

Outils de Communication Réseau sous Unix BSD 4.X

Philippe Durif

1999

Contents

1	Le concept d'Internet	1
1.1	Internet : un réseau de réseaux (format des adresses Internet)	1
1.2	IP : protocole de la couche réseau	2
1.3	UDP et TCP : protocoles de la couche transport	2
1.4	Internet et Unix BSD 4.X (i.e. les Suns)	2
1.4.1	La table des machines (hosts)	3
1.4.2	La table des réseaux (networks)	4
1.4.3	La table des services (/etc/services)	5
1.4.4	Manipulation des adresses Internet	5
1.4.5	Conversions machine/réseau	6
2	Adresses de la couche transport	7
2.1	Format général des adresses	7
2.2	Format des adresses du domaine Unix	8
2.3	Format des adresses du domaine Internet	8
2.4	Gestion des numéros de port	9
2.4.1	La table des services (/etc/services)	9
2.5	Fabriquer des adresses Internet	10
2.5.1	Utilitaires BSTRING(3)	12
3	Les sockets	13
3.1	Caractéristiques d'une socket	13
3.1.1	Domaines de communication d'une socket	13
3.1.2	Types de communication d'une socket	13
3.1.3	Protocole d'une socket	14
3.2	Primitives générales sur les sockets	14
3.2.1	Créer une socket : <code>socket()</code>	14
3.2.2	Détruire une socket : <code>close()</code>	15
3.2.3	Réduire les fonctionnalités d'une socket : <code>shutdown()</code>	15
3.2.4	Associer une adresse à une socket : <code>bind()</code>	15
3.2.5	Consulter l'adresse d'une socket : <code>getsockname()</code>	15
3.2.6	Exemples d'utilisations de <code>bind()</code>	16
3.2.7	Connexion de socket : <code>connect()</code>	17
3.2.8	Réception de message : <code>recv()</code>	17
3.2.9	Emission de message : <code>send()</code>	18
3.2.10	Lecture et écriture : <code>read()</code> , <code>write()</code>	18
3.2.11	Attente sélective : <code>select()</code>	18

3.3	Notion de client et de serveur	20
3.4	Autres aspects des sockets	22
4	Les sockets de type datagram (SOCK_DGRAM)	25
4.1	Règles générales d'utilisation des primitives	25
4.2	Exemples d'utilisation des sockets SOCK_DGRAM	26
4.3	Comment concevoir un nouveau service	29
5	Les sockets de type circuit virtuel (SOCK_STREAM)	33
5.1	Les primitives spécifiques aux serveurs en circuits virtuels	34
5.1.1	listen()	34
5.1.2	accept()	34
5.2	Un premier exemple	34
5.3	Utilisation des sockets SOCK_STREAM	36
5.4	Autres exemples	38
5.4.1	Un serveur de piles parallèle et multi-processus	38
6	Le mécanisme XDR (eXternal Data Representation)	41
6.1	Généralités sur le mécanisme XDR	41
6.2	Fonctions de gestion des flots XDR	41
6.2.1	fdopen() et fflush()	41
6.2.2	xdrstdio_create()	42
6.2.3	xdr_destroy()	42
6.3	Fonctions de transmission XDR	42
6.3.1	Fonctions de transmission sans allocation dynamique	43
6.3.1.1	xdr_void()	43
6.3.1.2	xdr_int()	43
6.3.1.3	xdr_opaque()	43
6.3.1.4	xdr_union()	43
6.3.2	Fonctions de transmission avec allocation dynamique éventuelle	44
6.3.2.1	xdr_string()	44
6.3.2.2	xdr_wrapstring()	44
6.3.2.3	xdr_array()	45
6.3.2.4	xdr_bytes()	45
6.3.2.5	xdr_reference()	45
6.3.2.6	xdr_pointer()	45
6.3.3	Libération de zone xdr_free()	46
6.4	Un exemple complet	47
6.5	Construction de nouvelles fonctions XDR	49
6.5.1	XDR de structures	49
6.5.2	XDR de tableau de taille variable	49
6.5.3	XDR de structures contenant un tableau	50
6.5.4	XDR de liste chaînée	50

7	Appel de procédure éloignée SunOS 4.x (RPC)	53
7.1	Rappel	53
7.2	Architecture des RPC de SunOS 4.x	53
7.2.1	Situation des RPC	53
7.2.2	Nommage des services RPC	53
7.2.3	Le processus portmap	54
7.2.4	L'enregistrement d'un serveur	55
7.2.5	Le mécanisme d'appel vu du client	55
7.2.6	Sémantique des appels	55
7.3	Primitives RPC de haut niveau	55
7.3.1	Le client	56
7.3.1.1	callrpc()	56
7.3.1.2	clnt_perrno()	56
7.3.2	Le serveur	56
7.3.2.1	registerrpc()	56
7.3.2.2	svc_run()	57
7.3.3	Exemple : un compte bancaire	57
7.4	Primitives RPC de bas niveau	58
7.4.1	Le client	58
7.4.1.1	clnt_create ()	59
7.4.1.2	clnt_control ()	59
7.4.1.3	clnt_pcreateerror ()	60
7.4.1.4	clnt_destroy()	60
7.4.1.5	clnt_call()	60
7.4.1.6	clnt_freeres()	60
7.4.1.7	clnt_perror()	60
7.4.2	Le serveur	60
7.4.2.1	svctcp_create()	61
7.4.2.2	svc_register()	61
7.4.2.3	Le dispatcher	61
7.4.3	Exemple : le compte bancaire	62
7.5	RPC non bloquant	63
7.6	Diffusion d'appel	64
7.7	Rétro appel	64
7.8	RPC sécurisés	64
7.8.1	Exemple : le compte bancaire sécurisé	64
8	Le générateur RPCGEN	67
8.1	Ecriture de l'interface en rpcgen	67
8.2	Ecriture des services	69
8.3	Ecriture d'un client	69
9	L'interface TLI	71
9.1	Architecture des TLI	71
9.1.1	Fournisseurs de transport ou TP (Transport Provider)	71
9.1.2	Ouverture d'un tep : t_open()	72
9.1.3	Publication d'un tep : t_bind()	72

9.1.4	Les événements : <code>t_look()</code>	72
9.1.5	Les états	73
9.1.6	Les structures de données : <code>t_alloc()</code>	76
9.2	Environnement de développement	77
9.3	Un exemple de communication en T_COTS	77

Chapter 1

Le concept d'Internet

1.1 Internet : un réseau de réseaux (format des adresses Internet)

L'idée de l'Internet (projet DARPA) est de fédérer différents réseaux hétérogènes dans un seul super-réseau ; en particulier, la technologie Internet ne fait pas disparaître les réseaux existant, elle s'appuie sur eux. Internet propose d'une part une technique d'adressage de réseaux et de machines, d'autre part une suite de protocoles.

Généralement les adresses réseaux ne désignent pas des machines mais plutôt des points d'accès au réseau, un point d'accès est, par exemple, une carte (carte Ethernet, carte Token Ring). Ainsi, une machine équipée de plusieurs cartes — une passerelle par exemple — aura plusieurs adresses. Par abus de langage, on parle souvent d'adresse de machine (host).

Lors de sa fabrication, chaque carte Ethernet est marquée d'un numéro unique de 48 bits. L'unicité de ce numéro fait qu'il peut représenter l'adresse de la carte, quel que soit l'environnement dans lequel elle est plongée.

En revanche, une carte Token Ring porte un numéro logiciel affecté par l'administrateur local du réseau. Une telle adresse n'est valable que dans ce réseau.

Dans Internet, une adresse de point d'accès est codée sur 32 bits et possède deux niveaux, les bits de poids forts permettent d'identifier le réseau parmi l'ensemble des réseaux connus par Internet, les bits de poids faible permettent de désigner un point d'accès particulier (une machine) dans ce réseau.

La gestion des adresses Internet est hiérarchique : le NIC (Network Information Control Center) distribue des fourchettes d'adresses réseau à des organismes officiels (pour la France c'est l'INRIA) qui se chargent d'allouer des adresses aux organismes demandeurs. La partie machine d'une adresse est gérée localement. Ce fonctionnement assure l'unicité d'une adresse Internet.

Internet propose trois formats d'adresse ce qui autorise une certaine souplesse vis à vis de la taille des réseaux physiques (voir figure 1.1).

Les protocoles de l'Internet sont nombreux, dont :

IP qui est le protocole réseau de base,

UDP et TCP sont les protocoles de transport construits sur IP,

ICMP protocole de contrôle (message d'erreur, régulation de flux, ...),

ARP (resp. RARP) spécifiques au réseau Ethernet et qui permet, connaissant l'adresse Internet (resp. Ethernet) d'une machine, d'obtenir son adresse Ethernet (resp. Internet).

Classe d'adresses	format	nombre	
		de réseaux	de points d'accès
A	0rrrrrrr mmmmmmm mmmmmmm mmmmmmm	128	16M
B	10rrrrrr rrrrrrr mmmmmmm mmmmmmm	16K	64K
C	110rrrrr rrrrrrr rrrrrrr mmmmmmm	2M	256

Figure 1.1: Les trois classes d'adresses Internet, les *r* désignent les bits d'adresse de réseau, les *m* ceux de point d'accès. La classe A correspond aux rares très grands réseaux, le réseau de Lille 1 est de classe B.

1.2 IP : protocole de la couche réseau

IP — Internet Protocol — est un protocole de commutation de paquet de point d'accès à point d'accès (couche réseau de l'ISO). Ses caractéristiques sont son manque de fiabilité (pas d'accusé de réception), l'ordre d'arrivée n'est pas forcément le même que l'ordre de départ, mais il y a préservation des frontières d'enregistrement. Typiquement un paquet IP possède une borne supérieure pour sa taille, son format est indiqué à la figure 1.2.

adresse Internet émetteur	adresse Internet destinataire	information utile
---------------------------	-------------------------------	-------------------

Figure 1.2: Format simplifié d'un paquet IP

1.3 UDP et TCP : protocoles de la couche transport

Les protocoles UDP — User Datagram Protocol — et TCP — Transport Control Protocol — se situent au dessus d'IP ; ils assurent tous deux la transmission d'information **d'application à application** et non plus de point d'accès à point d'accès ; plutôt que d'application on parlera de service ou encore de SAP¹. Plusieurs services pouvant tourner sur une même machine, on les distingue grâce à un **numéro de port** (sur Sun c'est un `u_short`). L'adresse d'un service est donc un couple adresse Internet, numéro de port.

UDP est une simple surcouche de IP (commutation de paquet) ; TCP assure une commutation de circuit virtuel avec les qualités qui en découlent (fiabilité, respect de l'ordre d'émission) mais il ne préserve pas les frontières d'enregistrement.

Les services officiels (well-known), comme FTP ou TELNET, portent des numéros de port alloués par un organisme centralisateur et sont inférieurs à `IPPORT_RESERVED`, constante définie dans `<netinet/in.h>`. Les ports au delà de `IPPORT_USERRESERVED` sont en principe utilisables librement.

1.4 Internet et Unix BSD 4.X (i.e. les Suns)

Le système Unix BSD 4.X des Suns intègre le concept Internet, le réseau physique étant la plupart du temps Ethernet. Chaque Sun possède donc deux adresses par point d'accès : une adresse Ethernet (pour les communications de la couche liaison de données : 802.3) et une adresse Internet.

¹selon la terminologie ISO : Service Access Point.

Les outils de développement réseau BSD 4.X proposent donc une approche Internet qui se matérialise dans la notion de **socket** donnant accès aux protocoles UDP et TCP.

D'autre part, les Macs du labo sont sur un réseau AppleTalk qui ne connaît pas Internet, il est cependant interconnecté avec Ethernet par une passerelle FastPath qui simule un adressage Internet des différents Macs.

Pour l'administration du réseau on dispose de plusieurs tables qui mettent en relation des noms externes symboliques (comme **homel**, **ftpd**, ...) et des noms internes (adresses Internet, numéro de port, ...). Les tables qui nous intéressent sont celles des machines, des réseaux et des services. Pour chacune de ces tables on dispose d'un interface shell et d'un interface programme.

1.4.1 La table des machines (hosts)

Elle met en relation les noms symboliques des machines et leurs adresses Internet.

l'interface shell Il propose le fichier **/etc/hosts** et la commande **ypcat hosts** qui utilise le NIS² ; cette dernière forme est plus sûre, en effet le fichier **/etc/hosts** local n'est pas forcément à jour. Chaque ligne correspond à une machine et contient de gauche à droite l'adresse Internet complète (4 champs décimaux séparés par des points), le nom symbolique officiel puis une liste éventuellement vide des noms "alias". Voici les formats d'adresses correspondant aux trois classes A, B et C :

Classe	Format
A	$r_1.s_3.s_2.s_1$
B	$r_2.r_1.s_2.s_1$
C	$r_3.r_2.r_1.s_1$

avec r_i et s_i compris entre 0 et 255.

l'interface programme Il propose les primitives de GETHOSTENT(3N) qui utilise la structure

```
struct hostent {
    char *h_name;
        /* official name of host */
    char **h_aliases;
        /* alias list, terminee par NULL */
    int h_addrtype;
        /* address type i.e. AF_INET */
    int h_length;
        /* length of address : 4 octets pour Internet */
    char **h_addr_list;
        /* liste des addresses, pour Internet la premiere */
        /* est la bonne et est en format reseau. */
        /* ATTENTION!!! il faut y acceder avec : */
        /* (unsigned long **) p->h_addr_list */
#define h_addr h_addr_list[0]
        /* address, for backward compatiblity */
};
```

²Network Information System : base de donnée accessible par toutes les machines du réseau

Les primitives sont :

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

struct hostent
    *gethostent (void),
    *gethostbyname (char *i_name);
```

Elles renvoient chacune un pointeur sur une structure `hostent` qui contient les champs d'une ligne du fichier `/etc/hosts`. `gethostent()` permet un parcours séquentiel des entrées et renvoie `NULL` après la dernière.

Attention : le pointeur renvoyé repère une zone statique dont le contenu est écrasé à chaque appel.

Les adresses (32 bits) sont fournies dans le format du réseau (cf `BYTEORDER`), pas dans celui de la machine.

Exemple tiré de la commande `ypcat hosts`

```
127.0.0.1      localhost
134.206.1.1    citil # CYBER 962-32
...
134.206.10.65 brigant
134.206.10.66 gordon
...
134.206.11.247 mac27
...
134.206.12.1  homel ms_mailhost
134.206.12.10 ms9
```

`localhost` représente la machine sur laquelle tourne le processus (à cause de 127 qui est un numéro spécial).

`homel` est le nom "officiel", il a un *alias* : `ms_mailhost`. 134 (1000 0110b) correspond à des adresses de classe B, l'adresse réseau de `homel` est donc 134.206, c'est en fait l'adresse, tout à fait officielle, du réseau `lilnet` (le réseau de LILLE 1). Il est possible de gérer localement des sous-réseaux, par exemple le premier octet de l'adresse machine correspond à l'adresse de sous-réseau : 10 désigne le réseau Ethernet du laboratoire, 11 est le réseau `Appletalk` des Macs, 12 est le réseau Ethernet de l'enseignement.

1.4.2 La table des réseaux (networks)

Elle met en relation les noms symboliques de réseau et leurs adresses Internet.

l'interface shell Il propose le fichier `/etc/networks` et la commande `ypcat networks`. Chaque ligne correspond à un réseau et contient de gauche à droite le nom symbolique officiel, la partie réseau de l'adresse Internet (au plus 3 champs décimaux séparés par des points), puis une liste éventuellement vide des noms "alias".

l'interface programme Il propose les primitives de `GETNETENT(3N)` qui utilisent la structure :

```

struct netent {
    char *n_name;
        /* official name of net */
    char **n_aliases;
        /* alias list, terminee par zero */
    int n_addrtype;
        /* net number type i.e. AF_INET */
    long n_net;
        /* net number en format interne et cadre */
        /* dans les octets de poids faibles */
};

```

Les primitives sont :

```

#include <netdb.h>

struct netent
    *getnetent (void),
    *getnetbyname (char *name);

```

qui renvoient un pointeur sur une structure statique correspondant à une ligne de la table des réseaux.

Exemple tiré de ypcat networks

```

arpanet      10      arpa
ucb-ether    46      ucbether
loopback     127
reunir       128.201
lilnet       134.206 localnet
irisnet      131.254

```

Moralité : les stations du M3 sont bien des sites du réseau lilnet !

1.4.3 La table des services (/etc/services)

Elle est présentée dans le chapitre suivant.

1.4.4 Manipulation des adresses Internet

La manipulation des adresses Internet est facilitée par les macros suivantes définies dans <netinet/in.h> :

```

/* Definitions of bits in internet address integers. */

#define IN_CLASSA(i) (((long)(i) & 0x80000000) == 0)
#define IN_CLASSA_NET 0xff000000
#define IN_CLASSA_NSHIFT 24
#define IN_CLASSA_HOST 0x00ffffff

#define IN_CLASSB(i) (((long)(i) & 0xc0000000) == 0x80000000)
#define IN_CLASSB_NET 0xffff0000
#define IN_CLASSB_NSHIFT 16
#define IN_CLASSB_HOST 0x0000ffff

#define IN_CLASSC(i) (((long)(i) & 0xe0000000) == 0xc0000000)
#define IN_CLASSC_NET 0xfffffff0
#define IN_CLASSC_NSHIFT 8
#define IN_CLASSC_HOST 0x000000ff

```

1.4.5 Conversions machine/réseau

Internet définit un ordre réseau standard des octets constituant les entiers 32 et 16 bits d'une adresse Internet ou d'un numéro de port. Cet ordre n'est pas forcément identique à celui de la machine. Les primitives suivantes effectuent les conversions nécessaires et sont à utiliser avec `gethostent` et `getservent`.

```

#include <sys/types.h>
#include <netinet/in.h>

netlong = htonl(hostlong);
u_long netlong, hostlong;

netshort = htons(hostshort);
u_short netshort, hostshort;

hostlong = ntohl(netlong);
u_long hostlong, netlong;

hostshort = ntohs(netshort);
u_short hostshort, netshort;

```

Exercice : écrire une commande `lh` (`listhost`) qui étant donné un nom de réseau affiche la liste des machines de ce réseau (idée, consulter la table des réseaux puis celle des machines).

Chapter 2

Adresses de la couche transport

On trouvera des informations utiles dans le manuel en ligne (`man 4` avec `intro`, `unix`, `inet`, `ip`, `tcp`, `udp`) ainsi que dans les fichiers d'inclusions mentionnés.

Un point d'accès (SAP) de la couche transport est ce qui permet à une application d'utiliser le réseau pour envoyer ou recevoir des données.

Pour que deux applications puissent échanger des données via la couche transport, chacune d'elle doit avoir accès à un SAP et être en mesure d'adresser le SAP de l'autre.

Le format d'une adresse de SAP dépend de la famille de protocole utilisée. Par exemple, pour les communications intra-Unix, une adresse de SAP est implantée comme une entrée dans le système de fichiers, par contre, dans Internet, une adresse de SAP est un couple (adresse Internet, numéro de port).

Un numéro de port est un entier qui identifie un SAP sur une machine pour un protocole particulier : `udp` et `tcp` ont chacun leur propre espace de numéros de port.

Le format d'adresse est orthogonal à l'API (Application Programming Interface) utilisée pour accéder à la couche transport. Par exemple, les adresses de SAP Internet sont implantées par la même structure C, qu'on utilise l'API des sockets (Unix BSD) ou bien l'API `tli` (Unix Système V).

Un point de transport (socket ou `tli`) n'est accessible de l'extérieur que si on lui a associé une adresse explicitement (`bind()` pour l'API socket) ou implicitement (dans le domaine internet et pour l'API socket, cette association est faite automatiquement lors d'un `connect()`, d'un `send()`, ...).

2.1 Format général des adresses

Le format général des adresses est décrit par la structure `sockaddr`¹ qui a pour seul rôle de permettre l'écriture des prototypes des primitives ayant une adresse en paramètre :

```
struct sockaddr {
    short sa_family;
        /* famille d'adresse (AF_INET, AF_UNIX,...) */
    char sa_data [14];
        /* contient l'adresse effective */
};
```

¹Le nom `sockaddr` semble indiquer qu'il s'agit d'adresse de socket. Il n'en est rien puisque ces même formats d'adresse sont aussi utilisables pour l'API `tli`. Cette ambiguïté est probablement due à des raisons historiques.

En fait, chaque famille de protocoles possède son propre format d'adresse. Le premier champs de la structure `sockaddr` permet justement de distinguer entre ces différents formats :

`AF_INET` pour la famille Internet,

`AF_UNIX` pour la famille Unix.

2.2 Format des adresses du domaine Unix

Dans le domaine Unix, une adresse est une entrée² dans le système de fichiers créée lors de l'exécution du `bind()`. Par exemple, le chemin `/users/graphix/durif/ma_socket`.

Dans `<sys/un.h>` on trouve le format de l'adresse Unix

```
struct sockaddr_un {
    short sun_family;
        /* AF_UNIX ou AF_UNSPEC pour casser */
        /* une pseudo-connexion datagram */
    char sun_path [108]; /* chemin */
};
```

2.3 Format des adresses du domaine Internet

Dans le domaine Internet, une adresse est un couple formé de l'adresse Internet de la machine, et du numéro de port. Un numéro de port est un identifiant unique dans le système et pour un protocole donné (un même numéro de port peut donc désigner une socket TCP et une socket UDP, ceci est le cas des services prédéfinis qui fonctionnent dans les deux protocoles comme `sunrpc`).

Dans `<netinet/in.h>` on trouve le format des adresses Internet.

²le type d'une entrée socket est `s`, celui des répertoires est `d`, celui des fichiers est `-`, ...

```

/* Internet address */

struct in_addr {
    union {
        struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;
        struct { u_short s_w1,s_w2; } S_un_w;
        u_long S_addr;
    } S_un;
#define s_addr S_un.S_addr /* should be used for all code */
};

struct sockaddr_in {
    short sin_family;
        /* AF_INET ou AF_UNSPEC pour casser */
        /* une pseudo-connexion datagram */
    u_short sin_port;
        /* port de la socket en format reseau */
    struct in_addr sin_addr;
        /* adresse internet de la machine en */
        /* format reseau (sin_addr.s_addr est u_long) */
    char sin_zero [8];
};

```

2.4 Gestion des numéros de port

La famille de protocole Internet définit un certain nombre de protocoles (ou applications) standards comme ftp, telnet et d'autres. D'autres protocoles (comme sunrpc) ne font pas partie de la famille Internet bien qu'étant largement diffusés.

Chacun de ces protocoles utilise un (ou plusieurs) SAP qui doivent être adressables. Pour cela, les SAP de ces protocoles sont affectés à des numéros de port bien connus qui doivent être identiques quel que soit le site qui les implémente.

La table des services répertorie l'ensemble de ces protocoles et les numéros de port qui leur ont officiellement été alloués.

2.4.1 La table des services (/etc/services)

Elle met en relation les noms des services officiels (well-known) avec les couples (numéros de port, protocole de transport) (principalement UDP et TCP). Les services sont soit spécifiques Unix comme rlogin, rsh, rcp, soit simplement Internet comme ftp, telnet ...

l'interface shell Il propose le fichier /etc/services ou ypcat services. Sur chaque ligne, on trouve le nom symbolique de service, son port officiel et le protocole dans lequel il est disponible.

l'interface programme Il propose les primitives de GETSERVENT(3N), la structure utilisée est la suivante

```

struct servent {
    char *s_name;
        /* official name of service */
    char **s_aliases;
        /* alias list terminée par NULL */
    int s_port;
        /* port service resides at : format reseau */
    char *s_proto;
        /* protocol to use "udp", "tcp" */
};

```

les primitives sont

```

#include <netdb.h>

struct servent
    *getservent (void),
    *getservbyname (char *i_name, char *i_proto);

```

Elles sont similaires à celles de GETHOSTENT mais pour le fichier `/etc/services`.

Si `i_proto` est NULL il n'est pas pris en compte dans la recherche, sinon c'est le nom du protocole souhaité ("udp", "tcp", ...).

Attention : le pointeur renvoyé repère une zone statique dont le contenu est écrasé à chaque appel.

Les numéros de port (32 bits) sont fournis dans le format du réseau (cf `ntoh` et `hton`), pas dans celui de la machine.

Exemple tiré de `ypcat services`

```

ftp      21/tcp
telnet   23/tcp
sunrpc   111/udp
sunrpc   111/tcp

```

`ftp` (File Transfer Protocol) est disponible sur le port 21 uniquement avec le protocole TCP. `sunrpc` (Remote Procedure Call) est disponible sur le port 111 dans les deux protocoles UDP et TCP : chaque protocole possède son propre ensemble de numéros de port.

2.5 Fabriquer des adresses Internet

La fabrication d'adresse Internet dépend du contexte :

- soit on veut associer une adresse à un SAP **local** pour le rendre accessible de l'extérieur,
- soit on veut fabriquer l'adresse d'un SAP **éloigné** pour dialoguer avec une application distante.

Dans ces deux cas il faut garnir les champs de la structure `sockaddr_in`, c'est à dire le numéro de port `sin_port` et l'adresse Internet `sin_addr`.

Voici un tableau résumant la manière de garnir le numéro de port `sin_port`.

		localisation du SAP	
		local	éloigné
nature du service	officiel	21	
		getservbyname ("ftp")	
	officieux statique dynamique	constante > 5000	
		0	serveur de nom

Dans le cas **officiel**, on utilisera de préférence le nom symbolique du service ("ftp" dans le tableau). Dans le cas **local** et **officieux**, on utilisera de préférence l'allocation dynamique du numéro de port par `bind()` en donnant 0 dans `sin_port` ce qui a l'avantage d'éviter tout conflit. L'inconvénient est qu'il faudra *publier* ce numéro.

Voici un tableau résumant la manière de garnir l'adresse internet `sin_addr`.

		localisation du SAP	
		local	éloigné
		<code>gethostname (moi)</code> <code>gethostbyname(moi)</code> <code>INADDR_ANY</code>	<code>gethostbyname(nom_serveur)</code>

Il est évidemment préférable d'interroger symboliquement la table des hosts pour obtenir l'adresse Internet d'une machine. Dans le cas où un serveur doit être accessible par toutes ses interfaces réseau, qui est probablement le plus fréquent, on donne la constante `INADDR_ANY` comme adresse machine.

Voici un utilitaire pour fabriquer une adresse du domaine Internet.

```
void makeAddress (struct sockaddr_in *o_pa, const char *host, int port)
{
    struct hostent *h ;

    bzero ((char *) o_pa, sizeof (struct sockaddr_in)) ;
    o_pa->sin_family = AF_INET ;
    o_pa->sin_port = htons (port) ;
    if ((h = gethostbyname (host)) == 0) {
        fprintf (stderr, "%s: machine inconnue\n", host) ;
        exit (1) ;
    } ;
    bcopy ((char *) h->h_addr, (char *) &o_pa->sin_addr, h->h_length) ;
}
```

Le paramètre `port` est supposé être en format machine. Remarquez l'utilisation de la macro `htons()` qui traduit la représentation machine de `port` en représentation réseau (Host TO Network Short).

Voici un utilitaire pour fabriquer l'adresse d'un SAP bien connu du domaine Internet.

```
void wellKnwonAddress (struct sockaddr_in *o_pa, const char *host,
                      const char *serv, const char *proto)
{
    struct servent *s ;

    if ((s = getservbyname (serv, proto)) == 0) {
        fprintf (stderr, "%s/%s: service inconnu\n", serv, proto) ;
    }
}
```

```
        exit (1) ;  
    } ;  
    makeAddress (o_pa, host, ntohs (s->s_port)) ;  
}
```

Exercice Ecrire un utilitaire similaire pour le domaine Unix.

2.5.1 Utilitaires BSTRING(3)

Deux utilitaires pour recopier et mettre à zéro des chaînes d'octets.

```
bcopy (const char *source, char *dest, int length) ;  
  
bzero (char *zone, int length) ;
```

Chapter 3

Les sockets

On trouvera des informations utiles dans le manuel en ligne (`man 4` avec `intro`, `unix`, `inet`, `ip`, `tcp`, `udp`) ainsi que dans les fichiers d'inclusions mentionnés.

Les **sockets** (prise ou SAP) sont l'incarnation Unix BSD 4.2 des couches réseau et transport de l'OSI. Elles autorisent l'utilisation de différentes familles de protocoles et en particulier ceux de l'Internet. Elles permettent une communication **inter-processus bidirectionnelle** soit sur une même machine soit à travers le réseau.

Toute communication met en jeu deux sockets qui doivent posséder les mêmes **caractéristiques**, en particulier elle doivent utiliser la même famille de protocole et le même protocole.

Un processus désigne par un **descripteur** (un simple entier) la socket locale qu'il a lui-même créée ou dont il a hérité lors d'un `fork()`. En revanche, pour désigner une socket éloignée, il doit spécifier son **adresse**.

3.1 Caractéristiques d'une socket

Les trois caractéristiques d'une socket sont son **domaine**, son **type** et son **protocole**. Des sockets destinées à communiquer doivent posséder les mêmes caractéristiques.

3.1.1 Domaines de communication d'une socket

Une socket peut être destinée à communiquer avec des sockets qui sont sur la même machine ou sur d'autres machines.

Domaine Unix (PF_UNIX) la socket ne peut communiquer qu'avec une socket située sur la même machine, on ne passe pas par le réseau (tout se passe en mémoire centrale).

Domaine Internet (PF_INET) la socket peut communiquer sur la même machine ou sur d'autres machines par l'intermédiaire du réseau.

A chaque domaine correspond, d'une part, un format d'adresse (voir chapitre précédent) et, d'autre part, une famille de protocoles, d'où le nom des constantes (PF comme Protocol Family). En particulier le domaine PF_INET correspond à la suite des protocoles de l'Internet (i.e. TCP/IP).

3.1.2 Types de communication d'une socket

Le type de communication fixe les caractéristiques logiques de la transmission.

datagram (SOCK_DGRAM) il n'y a pas de circuit virtuel entre les deux sockets et chaque paquet est envoyé indépendamment des autres, c'est à dire non fiable, non séquentiel mais préservation des frontières de paquet.

circuit virtuel (SOCK_STREAM) il y a d'abord établissement d'une connexion fixe, la communication fonctionne ensuite comme un pipe Unix, c'est à dire fiable, la séquence est préservée, pas de doublons mais il n'y a pas préservation des frontières d'enregistrement (par exemple, le processus émetteur peut envoyer 2 octets par 2 octets, et le récepteur peut lire 3 par 3).

raw (SOCK_RAW) utilise directement le protocole IP (réservé à root).

sequenced paquet (SOCK_SEQPACKET) cumule les avantages des circuits virtuels et la préservation des frontières.

Le type de communication permet de savoir quels protocoles du domaine sont utilisables pour implanter la communication désirée.

3.1.3 Protocole d'une socket

Le protocole permet de choisir le protocole dans l'ensemble des protocoles définis par le domaine et le type de communication.

Les différents protocoles existant sont répertoriés dans la table des protocoles (voir GETPROTOENT(3N)).

En général il n'existe qu'un protocole par domaine et par type ; d'autre part il existe un protocole par défaut déterminé par le domaine et le type de la socket (voir table 3.1).

protocole par défaut	SOCK_DGRAM	SOCK_STREAM	SOCK_SEQPACKET	SOCK_RAW
PF_INET	UDP	TCP	?	IP

Table 3.1: les protocoles par défaut.

On remarque qu'il n'y a pas de ligne pour le domaine PF_UNIX, en effet, la notion de protocole n'a pas de sens pour les sockets du domaine Unix.

3.2 Primitives générales sur les sockets

Les primitives qui suivent sont disponibles à la fois pour les sockets SOCK_DGRAM et SOCK_STREAM. En général, leur sémantique diffère suivant le type de la socket.

En général, une fonction renvoie une valeur positive ou nulle en cas de succès, une valeur négative en cas d'échec, dans ce cas on utilise `perror(char *)` pour imprimer un message approprié.

Sauf option particulière de socket, les primitives `connect()`, `recv()`, `send()`, `read()`, `write()` et `accept()` sont bloquantes.

L'usage des primitives décrites ci-après est illustré dans les deux chapitres qui suivent.

3.2.1 Créer une socket : `socket()`

C'est à la création d'une socket qu'on fixe ses trois caractéristiques.

```
#include <sys/types.h>
#include <sys/socket.h>

int socket (int domain, int type, int protocol);
```

crée une socket (i.e. alloue les ressources systèmes nécessaires) et renvoie son descripteur, ses caractéristiques sont fixées par les paramètres, en particulier on précisera 0 pour protocol afin d'obtenir le protocole par défaut. Exemple :

```
s = socket (PF_UNIX, SOCK_DGRAM, 0) ;
```

3.2.2 Détruire une socket : close()

```
close (int s);
```

ferme la socket désignée par `s` et restitue les ressources associées au système. On voit ici que la socket se comporte comme un simple descripteur de fichier; ceci sera vrai dans certains cas pour d'autres primitives et permettra de confondre la notion de socket et celle de fichier.

3.2.3 Réduire les fonctionnalités d'une socket : shutdown()

```
shutdown (int s, int how);
```

Réduit les fonctionnalités de `s` suivant la valeur de `how`.

how	0	1	2
fonction	écriture seule	lecture seule	plus rien

3.2.4 Associer une adresse à une socket : bind()

```
#include <sys/types.h>
#include <sys/socket.h>

bind (int s, struct sockaddr *i_saddr, int i_saddrlen);
```

Associe l'adresse `i_saddr` à la socket `s`. `i_saddrlen` est la taille de la structure `*i_saddr` effectivement passée. Contre toute apparence, tous les paramètres sont en entrée¹.

3.2.5 Consulter l'adresse d'une socket : getsockname()

```
getsockname (int s, struct sockaddr *o_saddr, int *io_saddrlen);
```

qui range dans `o_addr` l'adresse associée à `s`.

¹en cas d'ambiguïté possible, les noms de paramètres formels sont précédés de **i**, **io** ou **o** suivant qu'ils sont de mode **in**, **in out** ou **out**. Par exemple pour la primitive socket il n'y a pas d'ambiguïté : tous ses paramètres sont en entrée puisque le C ne possède que le passage par valeur. Une ambiguïté apparaît lorsqu'un paramètre est l'adresse d'une variable.

3.2.6 Exemples d'utilisations de bind()

La fabrication d'adresse du domaine Unix ne pose aucun problème et n'est pas illustrée.

Le code suivant pourrait être le début du démon telnet ; le `bind()` n'a aucune initiative à prendre, l'adresse étant totalement spécifiée.

```
{int s ;
  struct sockaddr_in snom ;

  s = socket (PF_INET, SOCK_STREAM, 0) ;
  {
    char hostname [MAXHOSTNAMELEN + 1] ;
    struct servent *serv = getservbyname ("telnet", "tcp") ;

    gethostname (hostname, MAXHOSTNAMELEN) ; /* nom machine locale */
    makeAddress (&snom, hostname, ntohs (serv->s_port)) ;
  }
  bind (s, (struct sockaddr *) &snom, sizeof snom) ;
  ...
```

Dans ce second exemple, c'est `bind()` qui alloue un numéro de port. Il faut ensuite le publier avec `printf()`.

```
{int s ;
  struct sockaddr_in snom ; int snom_taille = sizeof snom ;

  s = socket (PF_INET, SOCK_STREAM, 0) ;
  {
    char hostname [MAXHOSTNAMELEN + 1] ;

    gethostname (hostname, MAXHOSTNAMELEN) ; /* nom machine locale */
    makeAddress (&snom, hostname, 0) ;
  }
  bind (s, (struct sockaddr *) &snom, sizeof snom) ;
  getsockname (s, (struct sockaddr *) &snom, &snom_taille) ;
  printf ("port alloue : %d\n", ntohs (snom.sin_port)) ;
  ...
```

L'inconvénient majeur de cette technique est que le numéro de port publié est différent à chaque exécution, il faut alors donner le bon numéro aux futurs clients qui voudront se connecter à cette adresse. L'introduction d'un serveur de noms qui gère une table de correspondances nom symbolique, numéro de port, permet de supprimer ce problème : les différents intervenants de l'application n'ont plus qu'à connaître le nom symbolique invariable du port avec lequel ils veulent communiquer.

La carte `hosts` du NIS est un exemple de serveur de nom ; malheureusement, si vous n'êtes pas super utilisateur (`root`) vous ne pouvez pas ajouter de nouveaux services dans `/etc/services`.

Dans ce cas, on peut programmer soit même son propre serveur de numéro de port. Supposons que ce serveur existe, appelons le `snp` (comme Serveur de Numéro de Port). Voici une partie de l'interface qu'il propose :

```
/* interface destine aux clients */
```

```

unsigned int snp_consulter (char *nom, char *host) ;
/* renvoie le numero de port correspondant a nom sur host */

/* interface destine aux serveurs */

void snp_enregistrer (char *nom, unsigned int port) ;
/* enregistre la correspondance (nom.host_local, port) dans la table */

void snp_supprimer (char *nom) ;
/* supprime la correspondance (nom.host_local, port) de la table */

```

Lorsqu'un serveur alloue un numéro de port dynamiquement, il le publie auprès de `snp` en lui associant un nom symbolique. `snp` enregistre ce couple (nom symbolique, numéro de port) dans sa table. Lorsqu'un client désire contacter un serveur dont il connaît le nom symbolique, il interroge `snp` pour récupérer le numéro de port correspondant. On se servira de ces primitives dans la suite.

3.2.7 Connexion de socket : connect()

La connexion permet d'indiquer avec quel partenaire unique (socket éloignée) les échanges ultérieurs auront lieu ; les primitives `send()` et `write()` pourront dès lors être utilisées et c'est le partenaire qui en sera la cible.

```

#include <sys/types.h>
#include <sys/socket.h>

connect (int s, struct sockaddr *i_peer, int peerlen) ;

```

`s` est la socket locale, `i_peer` et `peerlen` forment l'adresse d'une socket éloignée.

Si `s` est de type `SOCK_STREAM` la connexion est obligatoire pour qu'une communication puisse avoir lieu. Le processus qui exécute ce `connect()` est considéré comme un client de la socket éloignée qui appartient au processus serveur. La demande de connexion d'un client n'aboutit que quand le serveur l'honore avec la primitive `accept()`, ceci correspond très précisément à l'établissement d'un circuit virtuel. Une fois créé, ce circuit ne peut être modifié, la seule possibilité est de détruire la socket ainsi connectée (`close()`).

Si `s` est de type `SOCK_DGRAM` il ne s'agit pas d'une connexion mais d'une facilité qui permet d'alléger les communications ultérieures ; les envois se feront sur `i_peer` et les réceptions seulement depuis `i_peer` ; `s` peut par la suite être connectée avec un autre partenaire par un nouvel appel à `connect()`, ou bien, simplement déconnectée si on spécifie `AF_UNSPEC` dans `i_peer->sa_family`.

3.2.8 Réception de message : recv()

```

#include <sys/types.h>
#include <sys/socket.h>

int recv (int s, char *o_buf, int lbuf, int flags) ;

int recvfrom (int s, char *o_buf, int lbuf, int flags,
              struct sockaddr *o_from, int *io_fromlen) ;

```

Ces primitives reçoivent sur `s` un message d'une autre socket. `recv()` ne peut fonctionner que si `s` est connectée (`connect()`). Le message est stocké dans le tampon `o_buf` de taille `lbuf`, la longueur effective du message est renvoyée par la primitive.

`flags` est 0 ou un "ou bits à bits" d'une ou plusieurs des valeurs suivantes

MSG_OOB qui spécifie qu'un message urgent doit être reçu ("out-of-band" data), seules les sockets `SOCK_STREAM` dans `INET` supportent cette option,

MSG_PEEK qui spécifie une réception non destructive.

Si `o_from` n'est pas le pointeur `NULL`, `o_from` et `io_fromlen` permettent de récupérer l'adresse de l'émetteur.

3.2.9 Emission de message : `send()`

```
#include <sys/types.h>
#include <sys/socket.h>

int send (int s, char *i_msg, int lmsg, int flags);

int sendto (int s, char *i_msg, int lmsg, int flags,
            struct sockaddr *i_to, int tolen);
```

Ces primitives envoient le message `i_msg`, `lmsg` sur la socket `s`. Pour `send()`, `s` doit être connectée (`connect()`).

`i_to` et `tolen` représentent l'adresse de la socket cible.

`flags` est 0 ou vaut `MSG_OOB` : le message est urgent ("out-of-band"), cette dernière possibilité n'existe que sur les sockets `SOCK_STREAM` et `PF_INET`.

3.2.10 Lecture et écriture : `read()`, `write()`

Le descripteur d'un ordre `read()` ou `write()` peut être une socket à condition qu'elle soit connectée.

Attention, `read()` renvoie le nombre d'octets lus qui peut être inférieur au nombre d'octets qu'on a demandé à lire, pour mener la lecture à son terme il est alors sensé de mettre le `read()` dans une boucle qui s'arrête dès qu'on a lu le nombre d'octets souhaité.

3.2.11 Attente sélective : `select()`

Cette primitive est générale aux descripteurs d'entrée sortie d'Unix : descripteurs de fichier (obtenus par `open()`), descripteurs de tube (obtenus par `pipe()`) et descripteurs de socket (obtenus par `socket()` et `accept()`).

Cette primitive est particulièrement utile lorsqu'on a plusieurs descripteurs ouverts et qu'on ne sait pas a priori sur lequel une lecture ou une écriture sera possible (on ne veut pas fixer un ordre a priori sur les ordres de lecture et/ou d'écriture).

Le `select()` et les macros associées permettent de connaître les descripteurs sur lesquels une opération d'entrée/sortie non bloquante est possible. Cette primitive est bloquante avec la possibilité de fixer un délai de garde (timeout).

```

#include <sys/types.h>
#include <sys/time.h>

int select (int width,
            fd_set *io_readfds, fd_set *io_writefds, fd_set *io_exceptfds,
            struct timeval *io_timeout);

FD_SET(fd,&fdset) /*positionne le descripteur fd dans le masque fdset*/
FD_CLR (fd, &fdset) /* raz du descripteur fd dans le masque fdset */
FD_ISSET (fd, &fdset) /* test du descripteur fd dans le masque fdset */
FD_ZERO (&fdset) /* raz du masque fdset */
int fd;
fd_set fdset;

```

Un masque de type `fd_set` représente un ensemble de descripteurs, les opérations sur un tel ensemble sont l'ajout et la suppression d'un descripteur (macros `FD_SET` et `FD_CLR`), le test d'appartenance (`FD_ISSET`) et l'initialisation à vide de l'ensemble (macro `FD_ZERO`).

En entrée, le masque `io_readfds` (resp. `io_writefds` et `io_exceptfds`), contient les descripteurs pour lesquels `select()` doit tester la possibilité d'une lecture non bloquante (resp. écriture et événement exceptionnel).

En sortie, le même masque `io_readfds` (resp. `io_writefds` et `io_exceptfds`), contient les descripteurs pour lesquels une lecture non bloquante est possible (resp. écriture et événement exceptionnel).

Un des trois masques peut être le pointeur `NULL` s'il est sans intérêt.

`width` indique le nombre de bits significatifs de chaque masque, sa valeur est donnée par `getdtablesize(2)`.

`io_timeout` donne un délai de garde au `select()`, si c'est le pointeur `NULL`, l'attente n'est pas limitée dans le temps.

`select()` renvoie le nombre total de descripteurs prêts, en particulier 0 si le délai de garde (`io_timeout`) a expiré. `select()` renvoie -1 en cas d'erreur, en particulier s'il a été interrompu par un signal.

Dans l'exemple suivant, on veut accéder au descripteur (`fd1` ou `fd2`) qui, le premier, dispose d'une donnée à lire :

```

...
while (1) {
    fd_set lecture ;

    FD_ZERO (&lecture) ;
    FD_SET (fd1, &lecture) ;
    FD_SET (fd2, &lecture) ;

    if (select (getdtablesize (), &lecture, NULL, NULL, NULL) >= 0) {

        if      (FD_ISSET (fd1, &lecture)) traiter_1 (fd1) ;
        else if (FD_ISSET (fd2, &lecture)) traiter_2 (fd2) ;
        else /* theoriquement impossible puisque timeout infini */

```

```

    } else if (errno == EINTR) {
        /* select a ete interrompu par un signal : reprendre la boucle */
    }
}

```

Exercice : On remarque que fd1 est favorisé par rapport à fd2, comment éviter ce favoritisme ?

3.3 Notion de client et de serveur

Conceptuellement, un **client** est une entité active qui prend des initiatives de son propre chef, par exemple il demande à un serveur de lui rendre un service ; en revanche, un **serveur** est une entité passive qui attend qu'on lui demande un service.

Techniquement, le serveur est le processus qui effectue un `bind()`, le client est celui qui exécute un `connect()`. Un serveur est souvent implanté comme un processus bouclant indéfiniment et pouvant répondre à de multiples clients. Un client connaît nécessairement le serveur qu'il désire utiliser, alors qu'un serveur ne peut prévoir a priori à quels clients il aura à faire.

En principe, le processus serveur doit être lancé avant qu'un client ne lui fasse une demande.

Un processus peut être à la fois serveur et client.

Voici un module `utilInet` qui permet la création de sockets destinées à des clients ou des serveurs dans le domaine Internet : tout d'abord `utilInet.h`

```

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

typedef int SOCKETI ; /* socket Internet */

SOCKETI mkClient (int type, char *rhost, char *nom) ;
SOCKETI mkServeur (int type, char *nom) ;
void delServeur (SOCKETI s, char *nom) ;

```

L'implémentation du module est dans `utilInet.c` :

```

#include "utilInet.h"
#include "Iaddress.h"
#include "snp.h"
static void test (int diag, char *mess)
{
    if (diag < 0) { perror (mess) ; exit (1) ; }
}

static void rendrePublicInet (SOCKETI s, char *nom)
{
    struct sockaddr_in nom_public ;
    int lg = sizeof nom_public ;
}

```

```

{
    char hostname [MAXHOSTNAMELEN + 1] ;

    gethostname (hostname, MAXHOSTNAMELEN) ; /* nom machine locale */
    makeAddress (&nom_public, hostname, 0) ;
}

test (bind (s, (struct sockaddr *) & nom_public, sizeof nom_public),
      "rendrePublicInet:bind") ;

/* publier le port alloué par bind() */
test (getsockname (s, (struct sockaddr *) &nom_public, &lg),
      "rendrePublicInet:getsockname") ;
snp_enregistrer (nom, nom_public.sinport) ;
/* ou, si pas de snp :
*   printf ("Numero de port alloué: %d\n", ntohs (nom_public.sin_port)) ;
*/
}

static void    connecterInet    (SOCKETI s, char *rhost, char *nom)
{
    struct sockaddr_in nom_public ;
    struct hostent *he ;

    /* creation du nom public de la socket partenaire */
    makeAddress (&nom_public, rhost, ntohs (snp_consulter (nom, rhost))) ;
    test (connect (s,(struct sockaddr*) &nom_public, sizeof nom_public),
          "connecterInet:connect") ;
}

SOCKETI mkClient    (int type, char *rhost, char *nom)
{
    SOCKETI s ;

    tester (s = socket (PF_INET, type, 0), "socket()") ;
    connecterInet (s, rhost, nom) ;
    return s ;
}

SOCKETI mkServeur (int type, char *nom)
{
    SOCKETI s ;

    tester (s = socket (PF_INET, type, 0), "socket()") ;
    rendrePublicInet (s, nom) ;
    return s ;
}

```

```
void    delServeur (SOCKETI s, char *nom)
{
    snp_supprimer (nom) ; close (s) ;
}
```

3.4 Autres aspects des sockets

fermeture en douceur On peut demander à ce que le close d'une socket soit retardé (*linger*) jusqu'à ce que tous les messages en émission aient effectivement été envoyés (ou bien jusqu'à ce qu'un délai de garde soit atteint).

```
struct linger {
    int  l_onoff; /* Linger active          */
    int  l_linger; /* How long to linger for (seconds) */
};

struct linger ling = {1, 10} ;

setsockopt (s, SOL_SOCKET, SO_LINGER, &ling, sizeof ling) ;
```

Le verbe anglais *linger* signifie s'attarder.

données urgentes (OOB ou Out Of Band) Possibles avec le `send()` et le `recv()` ; avec le flag `MSG_OOB` et seulement dans `PF_INET` et en `SOCK_STREAM` (c'est à dire avec `tcp`). Ce mécanisme permet de faire réaliser au récepteur des traitements asynchrones par rapport au flux séquentiel des données transmises. Le récepteur est averti de l'arrivée d'une donnée urgente (en fait un seul caractère) par le signal `SIGURG`. Pour recevoir ce signal le récepteur doit armer le signal (`signal()`) et demander à la socket de le déclencher lors de l'arrivée d'une donnée urgente. Voici le code de l'émetteur :

```
send (sock, "%", 1, MSG_OOB) ;
```

et le code du récepteur :

```
#include <signal.h>
#include <unistd.h>
#include <fcntl.h>

static int socket_a_tester1, socket_a_tester2 ;

static void handler_urgent (int sig)
{
    fd_set evenements_urgents ;
    char buf ;

    /* rearmer le signal */
    signal (sig, handler_urgent) ;

    FD_ZERO (&evenements_urgents) ;
    FD_SET (socket_a_tester1, &evenements_urgents) ;
```

```

    FD_SET (socket_a_tester2, &evenements_urgents) ;

/* determiner la socket */
select (getdtablesize(), 0, 0, &evenements_urgents, 0) ;

if (FD_ISSET (socket_a_tester1, &evenements_urgents)) {
    recv (socket_a_tester1, &buf, 1, MSG_OOB) ;
    ...
}
if (FD_ISSET (socket_a_tester2, &evenements_urgents)) {
    recv (socket_a_tester2, &buf, 1, MSG_OOB) ;
    ...
}
}

void main()
{
/* armer le signal */
    signal (SIGURG, handler_urgent) ;
    ...
/* pour recevoir le SIGURG */
    fcntl (socket_a_tester1, F_SETOWN, getpid()) ;
    fcntl (socket_a_tester2, F_SETOWN, getpid()) ;
    ...
}

```

sockets non bloquantes il s'agit d'une propriété générale aux descripteurs d'E/S, et donc disponible sur les sockets. On l'indique avec

```
fcntl (s, F_SETFL, FNDELAY | fcntl (s, F_GETFL, 0)) ;
```

les primitives `read()`, `write()`, `recv()`, `send()` et `accept()` ne sont plus bloquantes sur la socket `s` et renvoient une valeur particulière si l'opération était impossible.

diffusion (broadcasting) La diffusion n'est possible qu'en communication datagramme (`SOCK_DGRAM`) et si le processus appartient au super utilisateur, on rend une socket `s` diffusante avec :

```

{
    int on = 1 ;
    setsockopt (s, SOL_SOCKET, SO_BROADCAST, &on, sizeof on) ;
}

```


Chapter 4

Les sockets de type datagram (SOCK_DGRAM)

En type datagram les primitives ne supportent pas la distinction client/serveur (alors qu'en circuit virtuel cette asymétrie est tout à fait explicite). En fait la distinction éventuelle entre client et serveur est purement du domaine de l'application.

4.1 Règles générales d'utilisation des primitives

Une socket peut être dans un des trois états suivant :

anonyme c'est son état initial après sa création par `socket()`,

publique si elle est associée à une adresse mais n'est pas connectée,

connectée elle possède nécessairement une adresse et sa socket partenaire est connue du système.

Dans le domaine Internet exclusivement, les primitives `connect()` et `sendto()` ont pour effet de bord d'associer une adresse à la socket locale.

Les règles d'utilisations sont relativement intuitives : une socket peut recevoir si elle est **publique** ou **connectée**; elle peut émettre si elle est **connectée** ou bien si elle utilise `sendto()`. Voici un diagramme de transition décrivant l'évolution de l'état d'une socket. Une primitive est autorisée dans un certain état si le nouvel état est défini.

	anonyme	publique	connectée
<code>bind()</code>	publique		
<code>connect()</code>	connectée	connectée	connectée
<code>sendto()</code>	publique	publique	connectée
<code>recvfrom()</code>		publique	connectée
<code>recv()</code>			connectée
<code>read()</code>		publique	connectée
<code>send()</code>			connectée
<code>write()</code>			connectée

Par exemple, le `recv()` n'opère que sur une socket connectée et la socket ne change pas d'état et le `read()` ne peut se faire que sur une socket publique ou connectée, toute chose logique.

Attention : dans le domaine Unix, le `bind()` est obligatoire pour recevoir.

4.2 Exemples d'utilisation des sockets SOCK_DGRAM

Pour aérer les sources présentés ci-dessous, aucun traitement d'erreur sur le résultat des primitives n'est effectué. Il est cependant clair qu'un programme réel doit détecter et traiter (`perror()`) toute erreur due à l'exécution d'une primitive, ceci évite bien des ennuis en cas de bug.

Les deux premiers exemples présentent les sources d'un *client* qui envoie un message à un *serveur*, et du serveur correspondant qui reçoit ce message et l'envoie sur sa sortie standard, d'abord dans le domaine Unix puis dans le domaine Internet.

Le troisième exemple propose un service de "question/réponse" où le client envoie une question puis reçoit la réponse; le serveur reçoit la question, la pose à l'opérateur puis renvoie la réponse de celui-ci.

Exemple 1 émission/ réception en datagramme dans le domaine Unix.

Texte du processus émetteur :

```

1  /*
2  * emet.c
3  */
4  #include <sys/types.h>
5  #include <sys/socket.h>
6  #include <sys/un.h>
7  #include <stdio.h>
8
9  main (void)
10 {
11     int sock ;
12     struct sockaddr_un nom_public ;
13     char * buf = "un message vers le recepteur ....." ;
14
15     sock = socket (PF_UNIX, SOCK_DGRAM, 0) ;
16
17     /* creation du nom de la socket eloignee, puis connexion */
18     nom_public.sun_family = AF_UNIX ;
19     strcpy (nom_public.sun_path, "socket") ;
20     connect (sock, (struct sockaddr*) &nom_public, sizeof nom_public) ;
21
22     /* communication : strlen (buf) + 1 pour envoyer le '\0' terminateur */
23     write (sock, buf, strlen (buf) + 1) ;
24     close (sock) ;
25 }
```

On peut éviter la connexion en remplaçant les lignes 20 à 23 par :

```

sendto (sock, buf, strlen (buf) + 1, 0,
        (struct sockaddr *)&nom_public, sizeof nom_public) ;
```

Texte du processus récepteur :

```

/*
* recep.c
*/
```

```

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>

main (void)
{
    int sock ;
    struct sockaddr_un nom_public ;
    char buf [1024] ;

    sock = socket (PF_UNIX, SOCK_DGRAM, 0)

    /* creation du nom public de la socket, puis connexion */
    nom_public.sun_family = AF_UNIX ;
    strcpy (nom_public.sun_path, "socket") ;
    bind (sock, (struct sockaddr*) &nom_public, sizeof nom_public) ;

    /* communication: on recupere évidemment le caractere '\0' final */
    read (sock, buf, 1024) ;
    printf ("le receptrer a reçu: %s\n", buf) ;
    close (sock) ;
    unlink ("socket") ;
}

```

Le serveur détruit sa socket avant de se terminer.

Exemple 2 émission/réception en datagramme dans le domaine Internet.

Texte du processus émetteur :

```

/*
 * emet.c: emet host
 */
#include "utilInet.h"

main (int argc, char *argv[])
{
    SOCKETI sock ;
    char * buf = "un message vers le receptrer ....." ;

    if (argc != 2) {
        printf ("usage: emet host\n") ;
        exit (1) ;
    }

    sock = mkClient (SOCK_DGRAM, argv [1], "service");
    send (sock, buf, strlen (buf) + 1, 0) ;
    close (sock) ;
}

```

```
}

```

Texte du processus récepteur :

```
/*
 * recep.c
 */
#include "utilInet.h"

main (void)
{
    SOCKETI sock ;
    char buf [1024] ;

    sock = mkServeur (SOCK_DGRAM, "service") ;
    read (sock, buf, 1024) ;
    printf ("le recepteur a recu: %s\n", buf) ;
    delServeur (sock, "service") ;
}

```

Exemple 3 service de question/réponse dans le domaine Internet

Texte du client

```
/*
 * qr.c: qr noport question
 *
 * !!! REMARQUE: le serveur doit tourner sur "homel"
 *
 */
#include "utilInet.h"
#define HOTE_SERVEUR "homel"

main (int argc, char *argv []) ;
{
    SOCKETI sock ;
    char reponse [1024] ;

    if (argc != 3) {
        printf ("usage: qr question\n") ; exit (1) ;
    }
    sock = mkClient (SOCK_DGRAM, HOTE_SERVEUR, "qr");
    send (sock, argv [2], strlen (argv [2]) + 1, 0)
    recv (sock, reponse, 1024, 0) < 0) ;
    printf ("La reponse: %s\n", reponse) ;
    close (sock) ;
}

```

Texte du serveur

```
/*

```

```

* qrserveur.c: qrserveur
*
* mode d'emploi: doit etre lance en interactif et sur "homel"
*/
#include "utilInet.h"
#define HOTE_SERVEUR "homel"

main (void)
{SOCKETI sock ;

sock = mkServeur (SOCK_DGRAM, "qr") ;
while (1) {
    struct sockaddr_in client ;
    int lg = sizeof client ;
    char question [1024], reponse [1024] ;

    recvfrom (sock, question, 1024, 0, (struct sockaddr*) &client, &lg) ;
    printf ("Question: %s\nReponse? ", question) ;

    scanf ("%[^\\n]", reponse) ;
    sendto(sock, reponse, strlen(s)+1, 0, (struct sockaddr*) &client, lg) ;
}
}

```

4.3 Comment concevoir un nouveau service

Les deux exemples précédents sont particulièrement simples, il n'y quasiment pas de protocole! Considérons un exemple plus complexe. Le problème consiste à implanter un service de *mini conférence* : chaque machine peut posséder ce service. Les commandes destinées aux utilisateurs sont : **show host** qui affiche le dernier message stocké par la machine host, **over host message** qui écrase le dernier message de la machine host avec message.

La résolution d'un tel problème peut passer par plusieurs étapes

1. écrire les prototypes des souches (stub) d'accès au serveur ; les souches sont des sous-programmes dont le seul but est de faciliter l'utilisation du serveur en cachant les sockets (on obtient ici `conf.h`). Ces souches sont donc destinées exclusivement aux clients chaque souche correspond à un service.
2. déduire des services attendus un protocole de communication avec le serveur (ici le résultat est un fichier `confd.h`), le fichier `.h` obtenu est destiné à la fois au serveur et aux fonctions souches (voir plus loin).
3. écrire le code du serveur (ici `confd.c`), il s'agit d'obtenir un exécutable qui implante les services.
4. écrire les corps des souches (on obtient `conf.c`).
5. écrire le code des commandes clientes (ici `show.c` et `over.c`).

Voici ce que ça donne avec le problème de conférence :

1. les prototypes des souches sont dans `conf.h`

```

/*
 * conf.h
 */
void conf_show (char *rhost, char *o_texte) ;
void conf_over (char *rhost, char *texte) ;

```

2. le fichier confd.h contient le protocole clients/serveur

```

/*
 * confd.h
 */
#define LG_TEXTE 100
#define CONF_NAME "confd"

enum COMMANDE {
    CONF_SHOW, /* commande */
    CONF_OVER, /* commande */
    CONF_OK, /* compte-rendu */
    CONF_ERROR /* compte-rendu */
} ;

/* l'unique type de message transfere entre un client et le serveur */
struct conf_message {
    enum COMMANDE com ;
    char texte [LG_TEXTE] ;
} ;

/* protocole unique (inefficace mais simple):
 * 1) client >---- message (CONF_SHOW/CONF_OVER) ----> confd
 * 2) execution du service par confd
 * 3) cliend <--- message (CONF_OK/CONF_ERROR) -----< confd
 */

```

3. le fichier confd.c contient le code du serveur :

```

/*
 * confd.c
 */
#include "confd.h"

void main (void)
{
    int s = mkServeur (SOCK_DGRAM, CONF_NAME) ;
    char texte [LG_TEXTE] ;

    while (1) {
        struct conf_message m ;
        struct sockaddr_in client ;
        int lg_client = sizeof client ;

        recvfrom (s, &m, sizeof m, 0, &client, &lg_client) ;
    }
}

```

```

        switch (m.com) {
            case: CONF_OVER:
                strcpy (m.texte, texte) ; m.com = CONF_OK ; break ;
            case: CONF_SHOW:
                strcpy (texte, m.texte) ; m.com = CONF_OK ; break ;
            default:
                m.com = CONF_ERROR ; break ;
        }
        sendto (s, &m, sizeof m, 0, &client, sizeof client) ;
    }
}

```

4. les corps des souches sont dans `conf.c`

```

/*
 * conf.c
 */
#include "utilInet.h"
#include "confd.h"
#include "conf.h"
void conf_show (char *rhost, char *o_texte)
{
    struct conf_message m ;
    int s = mkClient (SOCK_DGRAM, rhost, CONF_NAME) ;

    m.com = CONF_SHOW ;
    write (s, &m, sizeof m) ;
    read (s, &m, sizeof m) ;
    strcpy (o_texte, m.texte) ;
    close (s) ;
}

void conf_over (char *rhost, char *texte)
{
    struct conf_message m ;
    int s = mkClient (SOCK_DGRAM, rhost, CONF_NAME) ;

    m.com = CONF_OVER ;
    strcpy (m.texte, texte) ;
    write (s, &m, sizeof m) ;
    read (s, &m, sizeof m) ;
    close (s) ;
}

```

5. enfin, le code de la commande `show`.

```

/*
 * show.c
 */
#include "conf.h"

```

```
void main (int argc, char *argv[])
{
    if (argc == 2) {
        char texte [2000] ;
        conf_show (argv [1], texte) ;
        printf ("%s\n", texte) ;
    }
}
```

On remarque que grâce aux stubs on écrit une commande client sans avoir à connaître l'existence des sockets.

Un certain nombre de fonctions de librairie utilisent le réseau sans le montrer à l'extérieur, par exemple `rnusers(host)` qui renvoie le nombre d'utilisateurs connectés sur la machine `host`, ou encore `callrpc()`.

Exercice : Notre serveur de nom `snp` du chapitre précédent est bien utile ! implantez-le en utilisant le même type d'architecture.

Chapter 5

Les sockets de type circuit virtuel (SOCK_STREAM)

Avec les sockets `SOCK_DGRAM`, les différents partenaires d'une communication sont sur un pied d'égalité, autrement dit, au niveau des primitives, rien ne permet de dire si un processus est un client ou un serveur. D'autre part, puisqu'il n'y a pas à proprement parler de connexion entre les partenaires, un processus peut, avec la même socket, dialoguer successivement avec plusieurs partenaires.

Avec les sockets `SOCK_STREAM`, on distingue clairement les serveurs des clients : la relation n'est pas symétrique. D'autre part, la liaison qui existe entre un client et son serveur est figée et ne peut être modifiée, c'est naturel puisqu'il s'agit d'une commutation de circuit.

La connexion client serveur (c'est à dire la mise en place du circuit virtuel entre les deux partenaires) s'établit de la manière suivante :

côté serveur il faut

1. créer une socket qui va recevoir les demandes de connexion des futurs clients — `socket()`,
2. lui donner une adresse — `bind()`,
3. indiquer qu'à partir de maintenant elle attend des demandes de connexion — `listen()`,
4. honorer les demandes de connexion des clients avec la primitive `accept()` dont le résultat est une *nouvelle* socket qui est effectivement connectée à celle du client.

côté client il faut

1. créer la socket — `socket()`,
2. la connecter à celle du serveur — `connect()`.

Il faut noter une différence importante du serveur stream par rapport au serveur datagramme : la socket créée initialement sert exclusivement à recevoir les demandes de connexions, c'est une autre socket, créée par `accept()`, qui va servir à la communication effective avec le client. Ceci permet à un serveur unique de servir plusieurs clients.

Une fois la connexion établie, il est possible de la spécialiser avec `shutdown()`, par exemple si on sait que les informations ne circulent que dans un seul sens. Le dialogue peut ensuite prendre place avec les primitives `send()`, `recv()`, `read()` et `write()`. Enfin les deux partenaires détruisent la connexion avec `close()`.

5.1 Les primitives spécifiques aux serveurs en circuits virtuels

5.1.1 listen()

```
listen (int s, int backlog) ;
```

Déclare `s` comme une socket de connexion. `s` doit être de type `SOCK_STREAM` ou `SOCK_SEQPACKET` et posséder une adresse (`bind()`), `backlog` indique le nombre maximum de demandes de connexion en attente (limité par la constante `SOMAXCONN` définie dans `sys/socket.h`). `listen()` doit être exécutée avant de pouvoir exécuter un `accept()` sur `s`.

Le `connect()` d'un client échoue si la file d'attente de la socket de connexion du serveur est saturée.

5.1.2 accept()

```
#include <sys/types.h>
#include <sys/socket.h>

int accept (int s, struct sockaddr *o_addr, int *io_addrlen) ;
```

`s` est une socket `SOCK_STREAM` ou `SOCK_SEQPACKET`, qui possède un nom public (`bind()`) et une file d'attente de demandes de connexion (`listen()`).

`accept()` extrait la première demande de connexion de la file d'attente de `s`, l'identité du demandeur est rangé dans `o_addr`, `io_addrlen`, une nouvelle socket est créée (résultat de `accept()`) qui est mise en connexion avec le demandeur. Ensuite, `s` reste disponible pour accepter les autres demandes de connexion.

Autant se faire tout de suite un petit utilitaire

```
int accepter (int sock)
{
    struct sockaddr_in *peer ;
    int lpeer = sizeof peer ;
    return accept (sock, (struct sockaddr *)& peer, &lpeer) ;
}
```

en effet l'identité du partenaire est rarement nécessaire.

5.2 Un premier exemple

Cet exemple reprend l'exemple émetteur/récepteur déjà développé en datagramme (sockets `SOCK_DGRAM`) et illustre l'utilisation du `listen()` et du `accept()` pour l'implantation d'un serveur.

Texte du processus émetteur (c'est le client) :

```
/*
 * emet.c: emet host noport
 */
```

```

#include "utilInet.h"

void envoyer (int s, char *buf, int lgbuf) /* lgbuf >= 1 */
{
    while (1) {
        int lgeff = send (s, buf, lgbuf, 0) ;

        if (lgeff == -1) {
            perror ("envoyer()") ; exit (1) ;
        }
        buf += lgeff ;
        lgbuf -= lgeff ;
        if (lgbuf == 0) break ;
    }
}

void main (int argc, char *argv []) ;
{
    SOCKETI sock ;
    char * buf = "un message vers le recepneur ....." ;

    if (argc != 3) { printf ("usage: emet host noport\n") ; exit (1) ; }

    sock = mkClient (SOCK_STREAM, argv [1], atoi (argv [2]));
    envoyer (sock, buf, strlen (buf) + 1) ;
    close (sock) ;
}

```

Texte du processus récepteur (c'est le serveur) :

```

/*
 * recep.c
 */
#include "utilInet.h"

void recevoir (int s, char *buf, int lgbuf)
{
    while (1) {
        int lgeff = read (s, buf, lgbuf) ;

        if (lgeff == -1) {
            perror ("recevoir()") ; exit (1) ;
        }
        if (lgeff != 0 && buf [lgeff-1] == 0) break ;
        lgbuf -= lgeff ; buf += lgeff ;
    }
}

```

```

void main (void)
{
    SOCKETI sockConn ;
    char buf [1024] ;

    sockConn = mkServeur (SOCK_STREAM) ;
    /* a partir d'ici notez la difference avec la version SOCK_DGRAM */
    listen (sockConn, 1) ;
    {
        SOCKETI s = accepter (sockConn) ;

        recevoir (s, buf, 1024) ;
        printf ("le recepteur a recu: %s\n", buf) ;
        close (s) ;
    }
    close (sockConn) ;
}

```

Exercice : adaptez le service de question/réponse pour qu'il fonctionne en circuit virtuel.

5.3 Utilisation des sockets SOCK_STREAM

Généralement, un serveur est un démon qui tourne indéfiniment. Un tel serveur peut servir plusieurs clients. On peut distinguer plusieurs types de serveurs suivant la manière dont les clients sont servis.

serveur séquentiel : les clients sont satisfaits les uns à la suite des autres. La boucle d'un tel serveur a typiquement la forme suivante :

```

while (1) {
    int s = accepter (sockConn) ;

    servir_client (s) ;
    close (s) ;
}

```

serveur parallèle : plusieurs clients client peuvent être servis simultanément. Le serveur peut être **mono-processus** (utilisation du `select()`) ou **multi-processus** (utilisation du `fork()`).

multi-processus c'est le plus simple à réaliser et aussi le plus coûteux, il a la forme suivante :

```

void recupere (void) { while (wait (0) != -1) ; }

void main (void)
{
    ...
    signal (SIGCHLD, recupere) ; /* pour eviter les zombies */
    while (1) {
        int s = accepter (sockConn) ;

        if (fork() == 0) {

```

```

        close (sockConn) ;
        servir_client (s) ;
        exit (0) ;
    }
    close (s) ;
}

```

mono-processus il utilise le `select()` pour déterminer le client à servir. Il doit donc gérer l'ensemble des clients en cours de connexion (autrement dit l'ensemble des sockets connectées), il a la forme suivante :

```

void main (void) {
    fd_set clients ;
    ...
    FD_ZERO (&clients)
    while (1) {
        fd_set lecture = clients ;
        FD_SET (sockConn, &lecture) ;

        if (select (getdtablesize (), &lecture, NULL, NULL, NULL) > 0) {
            if (FD_ISSET (sockConn, &lecture)) {
                int nouveau_client = accepter (sockConn) ;
                FD_SET (nouveau_client, &clients) ;
            } else {
                honorer_la_demande_de (un_client_pret (&lecture)) ;
            }
        }
    }
}

static int un_client_pret (fd_set *masque)
{
    unsigned fd = 0 ;

    while (! FD_ISSET (fd, masque)) fd++ ;
    return fd ;
}

```

On utilise deux ensembles de descripteurs : `lecture` qui est temporaire et ne sert que pour le `select()` et `clients` qui est permanent et permet de mémoriser l'ensemble des clients connectés. A nouveau, on remarque l'inéquité de la fonction `un_client_pret()`.

Fonctionnellement, la principale différence entre le mono et le multi-processus est la possibilité ou non de partager les données entre les clients.

Exercice : faire que le serveur parallèle mono-processus réponde de façon plus équitable à ses clients.

L'avantage évident d'un serveur parallèle est qu'un client n'est pas bloqué par ceux qui le précèdent.

Exercice : adaptez ces schémas de serveurs séquentiel, parallèle multi-processus et parallèle mono-processus au type de communication datagramme.

5.4 Autres exemples

Pour la conception des exemples suivants, on applique la méthode décrite précédemment pour les sockets datagramme : énumération des services, choix du protocole, implantation du serveur et des sous-programmes souches.

5.4.1 Un serveur de piles parallèle et multi-processus

Voici un serveur de piles d'entiers (on l'appelle `piled`), chaque client dispose de sa propre pile, en effet le serveur est parallèle multi-processus ; cet exemple illustre l'utilisation conjointe du `accept()` et du `fork()` et l'utilisation bidirectionnelle des sockets.

1. `piled.h`, le fichier d'interface du serveur

```
/*
 * piled.h: protocole du serveur de piles d'entiers
 */
#define NOM_PILE "piled"

enum COMMANDE {
    EMPILER,      /* un parametre, par de retour */
    DEPILER,     /* retour de l'ancien sommet */
    DETRUIRE,    /* pas de retour */
    PILEVIDE     /* retour d'un entier */
} ;
#define TMESS sizeof (int) /* un message est un simple entier ! */
```

2. `piled.c`, le code du serveur

```
/*
 * piled.c: le code du serveur
 */
#include "utilInet.h"
#include "piled.h"

#define TPILE 10

struct pile {
    int som ;
    int p [TPILE] ;
} ;

static void servir_un_client (SOCKETI sclient) ;
{
    struct pile p ;

    p.som = -1 ;
    while (1) {enum COMMANDE c ; int vide, sortir = 0 ;
        read (sclient, (char *) &c, sizeof c) ;
        switch (c) {
```

```

    case EMPILER:
        read (sclient, (char *) &p.p [++p.som], sizeof (int)) ;
        break ;
    case DEPILER:
        write (sclient, (char *) &p.p [p.som--], sizeof (int)) ;
        break ;
    case DETRUIRE:
        sortir = 1 ;
        break ;
    case PILEVIDE:
        vide = p.som == -1 ;
        write (sclient, (char *) &vide, sizeof vide) ;
        break ;
    default:
        break ;
}
if (sortir) break ;
}
close (sclient) ;
}

main (void)
{
    SOCKETI sockConn ;

    sockConn = mkServeur (SOCK_STREAM, NOM_PILE) ;
    listen (sockConn, SOMAXCONN) ;
    while (1) {
        SOCKETI sclient = accepter (sockConn) ;

        if (fork () == 0) {
            close (sockConn) ;
            servir_un_client (sclient) ;
            exit (0) ;
        }
        close (sclient) ;
    }
}

```

Exercice 1 : écrire la partie souche du serveur (`pile.h` et `pile.c`),

Exercice 2 : adaptez ce serveur pour qu'il soit mono-processus, ainsi, les différents clients partageront une seule et même pile (ceci peut-il avoir un sens pratique ??)

Exercice 3 : adaptez ce serveur pour le type de communication datagramme.

Exercice : il est possible de rediriger l'entrée standard d'une commande sur une socket avec `dup2()`. Faites le pour utiliser la commande `more` pour afficher le contenu d'un fichier distant avec une commande du genre `plus host file`.

Chapter 6

Le mécanisme XDR (eXternal Data Representation)

On l'appelle aussi mécanisme d'empaquetage/déempaquetage, de marshalling ou tout simplement de transmission. Dans la suite, le terme transmission désigne à la fois l'émission et la réception.

Le but du mécanisme XDR est double :

- permettre à deux machines ayant des représentations internes de données différentes de se comprendre. En effet, que se passe-t-il si un Sun Sparc envoie un nombre flottant vers un Vax avec un bête `send()` ? Pour cela, le mécanisme XDR propose une représentation externe standard des données.
- permettre la transmission de structures de données évoluées (par exemple à base de pointeurs). Les primitives du type de `send()` et `recv()` ne permettent de transmettre que des séquences d'octets sans signification particulière.

L'accès au XDR se fait avec

```
#include <rpc/xdr.h>
```

6.1 Généralités sur le mécanisme XDR

L'objet central du mécanisme XDR est le flot XDR de type `XDR`. En fonctions des paramètres fixés lors de sa création, un flot XDR ne peut assurer qu'une fonctionnalité parmi les trois suivantes : l'émission, la réception ou la libération de mémoire. Cette dernière fonctionnalité semble étrange, elle est dûe au fait que, lors de la réception, un flot XDR peut être amené à allouer dynamiquement la zone de mémoire réceptrice ; il est alors nécessaire par la suite de libérer cette zone.

6.2 Fonctions de gestion des flots XDR

Avant d'être utilisable, un flot XDR doit être construit au dessus d'un `FILE *` qui lui-même est construit au dessus d'un descripteur (descripteur de fichier, de socket, de mémoire partagée, ...).

6.2.1 `fdopen()` et `fflush()`

Pour construire un `FILE *` au dessus d'une socket on utilise la fonction :

```
FILE *fdopen (int fd, char *mode /* "r", "w" ou "r+" */);
```

où `fd` est le descripteur de la socket.

Puisqu'un flot XDR utilise un `FILE *` qui possède un tampon local, il est parfois nécessaire de vider ce tampon explicitement afin de s'assurer qu'une donnée a bien été envoyée. Pour cela, on utilise la fonction

```
int fflush (FILE *f) ;
```

qui vide le tampon sur le descripteur associé.

6.2.2 xdrstdio_create()

```
void xdrstdio_create (XDR *o_xdrs, FILE *filep, enum xdr_op op) ;
```

créé dans `o_xdrs` un flot XDR. Les futures opérations de lecture (réception) ou d'écriture (émission) auront lieu sur le flot d'entrée sortie standard `filep`. Le paramètre `op` fixe le fonctionnement du flot :

XDR_ENCODE le flot fonctionne en émission, il traduit la représentation interne des objets dans leur représentation standard externe,

XDR_DECODE le flot fonctionne en réception, il traduit la représentation externe standard des objets dans leurs représentations interne. Il peut y avoir allocation dynamique de la zone de mémoire destinée à l'objet reçu. APRES ALLOCATION, CETTE ZONE EST AUTOMATIQUEMENT MISE A ZERO, ceci est indispensable lorsque l'objet contient lui-même des pointeurs qui doivent eux aussi être alloués.

XDR_FREE la fonction libère la mémoire précédemment allouée lors de la à un objet, `*pobjet` est considéré comme un paramètre donnée/résultat. Ceci permet de libérer la mémoire allouée lors d'un décodage antérieur.

Les constantes précédentes sont définies dans le fichier `rpc/xdr.h`.

6.2.3 xdr_destroy()

```
void xdr_destroy (XDR *xdrs) ;
```

vide le tampon du flot d'entrée sortie standard sous-jacent (`fflush()`) puis détruit le flot XDR lui-même. Attention, un `fclose()` reste nécessaire pour fermer le flot d'entrée sortie standard, ainsi qu'un `close()` pour le descripteur.

6.3 Fonctions de transmission XDR

Ce sont les fonctions de transmission où simplement fonctions XDR qui assurent l'encodage/décodage d'objets sur un flots XDR.

Ces fonctions renvoient vrai si la transmission s'est correctement déroulée et faux si une erreur s'est produite.

Certaines de ces fonctions ont un paramètre qui est lui-même une fonction XDR, le type de ce paramètre est `xdrproc_t` qui est défini par :

```
typedef bool_t (*xdrproc_t) (XDR *, void *) ;
```

Puisqu'une unique fonction XDR sert à la fois pour l'émission et la réception, l'objet qu'elle transmet doit toujours être un pointeur :

```
bool_t xdr_TYPE_OBJET (XDR *x, TYPE_OBJET *po) ;
```

Les fonctions de transmission se répartissent en deux groupes : celles qui ne font jamais aucune allocation dynamique pour l'objet à recevoir et celles qui en font éventuellement une.

6.3.1 Fonctions de transmission sans allocation dynamique

Ces fonctions de transmissions ne font pas d'allocation dynamique car elles s'appliquent à des données de taille fixe et connue par avance.

6.3.1.1 xdr_void()

```
bool_t xdr_void(void) ;
```

cette fonction indique l'absence de paramètre : elle ne transmet rien.

6.3.1.2 xdr_int()

```
bool_t xdr_int(XDR *xdrs, int *ip) ;
```

une fonction de ce genre existe pour chacun des types de base de C (char, short, long, float, double, bool_t, u_int, ...).

6.3.1.3 xdr_opaque()

```
bool_t xdr_opaque (XDR *xdrs, char *cp, u_int cnt) ;
```

transmet *telle quelle* une donnée opaque désignée par cp de longueur cnt. Elle est utilisée par un programme qui a besoin de stocker une information mais qui ne la consulte jamais lui-même, par exemple les handles de fichiers NFS sont des informations opaques pour les clients NFS.

6.3.1.4 xdr_union()

```
#define NULL_xdrproc_t ((xdrproc_t)0)
struct xdr_discrim {
    int value ;
    xdrproc_t proc ;
};

bool_t xdr_union(XDR *xdrs,
    int *dscmp,
    char *unp,
    struct xdr_discrim *choices,
    bool_t (*defaultarm) () /* may equal NULL */
) ;
```

pour traduire des unions avec discriminant, `dscmp` repère le discriminant, `unp` l'union, `choices` est une table dont chaque entrée contient une valeur de discriminant et le pointeur de fonction XDR correspondant, la dernière entrée sert de sentinelle : son pointeur de fonction est `NULL` ; la fonction `XDR_defaultarm` sera utilisée si la valeur pointée par `dscmp` n'est pas trouvée dans `choices`. Par exemple :

```
enum type {ENTIER, FLOTTANT} ;
struct exemple {
    enum type t ;
    union {
        int eval ;
        float fval ;
    } val ;
}

struct xdr_discrim choices [] = {
    {ENTIER, xdr_int      },
    {FLOTTANT, xdr_float  },
    {0,        NULL_xdrproc_t}
} ;

bool_t xdr_exemple (XDR *xdrs, struct exemple *e)
{
    return xdr_union (xdrs, &e->t, (char *) &e->val, choices, NULL) ;
}
```

6.3.2 Fonctions de transmission avec allocation dynamique éventuelle

Ces fonctions de transmissions s'applique à des données de taille variables (comme des listes chaînées) ou non connues par avance.

Il est alors possible de leur demander, du côté récepteur, d'allouer dynamiquement les données reçues. Cette allocation dynamique n'a lieu que si le pointeur sur la zone est nul ; la zone allouée est immédiatement mise à zéro avant que son contenu ne soit reçu (ce qui autorise la transmission de données récursives).

6.3.2.1 `xdr_string()`

```
typedef char *string ;
bool_t xdr_string (XDR *xdrs, string *ps, u_int maxsize) ;
```

transmet une chaîne de caractères C pas plus longue que `maxsize`.

6.3.2.2 `xdr_wrapstring()`

```
typedef char *wrapstring ;
bool_t xdr_wrapstring (XDR *xdrs, wrapstring *pws) ;
```

comme `xdr_string()` mais sans limitation sur la taille de la chaîne.

6.3.2.3 xdr_array()

```

typedef bool_t (*xdrproc_t)(XDR *, caddr_t *);

typedef elem array [];
bool_t xdr_array(XDR *xdrs,
    array *pa, u_int *effsize, u_int maxsize,
    u_int elemsize, xdrproc_t xdr_elem
);

```

`effsize` indique le nombre effectif d'éléments du tableau `*pa` et doit être inférieur à `maxsize`, `xdr_elem` est l'XDR à appliquer à chacun des éléments du tableau, la taille d'un élément est `elemsize`. Remarquons que `elemsize` n'est pas transmis puisque c'est une information spécifique à la machine locale.

6.3.2.4 xdr_bytes()

```

typedef char bytes [];
bool_t xdr_bytes(XDR *xdrs, bytes *pb, u_int *effsize, u_int maxsize);

```

comme `xdr_array()` sauf que cette fois le type des éléments est fixé : c'est l'octet (`char`), il n'est donc pas nécessaire de spécifier la taille d'un élément ni la fonction XDR correspondante.

6.3.2.5 xdr_reference()

Pour transmettre un pointeur qui doit être non nul lors de l'émission.

```

typedef objet *POINTEUR_NON_NULL;
bool_t xdr_reference(XDR *xdrs,
    POINTEUR_NON_NULL *pp,
    u_int objetsize,
    xdrproc_t xdr_objet
);

```

transmet, avec la fonction XDR `xdr_objet`, l'objet désigné par le pointeur `*pp`. ATTENTION, lors de l'encodage, cette primitive ne reconnaît pas le pointeur NULL, c'est à dire que la zone repérée par `*pp` sera effectivement transmise même si elle est à l'adresse 0. Lors du décodage, `*pp` peut être NULL, dans ce cas, la primitive alloue dynamiquement l'objet, `objetsize` donne alors la taille de zone mémoire à allouer.

6.3.2.6 xdr_pointer()

Pour transmettre des données récursives : un pointeur nul est correctement interprété.

```
typedef objet *POINTEUR ;
bool_t xdr_pointer(XDR *xdrs,
    POINTEUR *pp,
    u_int objsize,
    xdrproc_t xdr_objet
);
```

comme `xdr_reference()` sauf qu'elle traite correctement le cas où `*pp` est NULL en émission, et peut donc servir à transmettre des structures récursives.

6.3.3 Libération de zone `xdr_free()`

```
typedef ... objet ;
void xdr_free (xdrproc_t xdr_objet, objet *po) ;
```

cette fonction applique l'XDR `xdr_objet` passée en paramètre pour libérer l'objet repéré par le pointeur `*po` ; ATTENTION, `xdr_objet` doit être une fonction à deux paramètres XDR* et objet*. Par exemple on peut libérer une chaîne de la manière suivante :

```
char *s = NULL ;

xdr_wrapstring (&xdrs, &s) ; /* reception avec allocation dans s */
...
xdr_free (xdr_wrapstring, &s) ;
...
```

Exercice corrigé : écrire le source supposé de `xdr_array()`.

```
#define XDR(xdr_fun, objet) \
{ \
    if (! (* xdr_fun) (xdrs, (objet))) return FALSE ; \
}

bool_t xdr_array_suppose (XDR *xdrs,
    char **ppo,
    u_int *sizep, u_int maxsize, u_int elsize,
    xdrproc_t elproc)
{
    char *ptr ;
    unsigned cpt ;

    XDR (xdr_u_int, sizep) ;
    if (*sizep > maxsize) return FALSE ;

    if (xdrs->x_op == XDR_DECODE && *ppo == 0) {
        *ppo = calloc (*sizep, elsize) ; /* la zone est mise a 0 */
    }
    for (ptr = *ppo, cpt = 0 ; cpt < *sizep ; ptr += elsize, cpt++) {
        XDR (elproc, ptr) ;
    }
}
```

```

    }
    if (xdrs->x_op == XDR_FREE) free (*ppo) ;
    return TRUE ;
}

```

Exercice : écrire le code de `xdr_pointer()`.

```

bool_t xdr_pointer(XDR *xdrs,
    char **ppo,
    u_int osize,
    xdrproc_t xdr_o)
{
    bool_t present ;

    switch (xdrs->x_op) {
        case XDR_ENCODE :
            present = *ppo != 0 ;
            XDR (xdr_bool, &present) ;
            if (present) XDR (xdr_o, *ppo) ;
            break ;
        case XDR_DECODE :
            XDR (xdr_bool, &present) ;
            if (present) {
                if (*ppo == 0) {
                    *ppo = malloc (osize) ;
                    bzero (*ppo, osize) ;
                }
                XDR (xdr_o, *ppo) ;
            } else *ppo = 0 ;
            break ;
        case XDR_FREE :
            if (*ppo != 0) {
                XDR (xdr_o, *ppo) ;
                free (*ppo) ;
                *ppo = 0 ;
            }
            break ;
    }
    return TRUE ;
}

```

6.4 Un exemple complet

Toujours le même de service d'impression d'un message à l'écran.

```

/*
 * exdr.c: emetteur avec XDR (usage : exdr rhost)
 */

```

```

#include <stdio.h>
#include <rpc/xdr.h>

#include "utilInet.h"

void main (int argc, char *argv[])
{
    int sock ;
    char *b = "un message ... vers recepateur" ;
    XDR x ;
    FILE *f ;

    if (argc != 2) { fprintf (stderr, "usage : exdr rhost\n") ; exit (1) ; }

    sock = mkClient (SOCK_DGRAM , argv [1], "impression") ;
    shutdown (sock, 0) ;
    xdrstdio_create (&x, fdopen (sock, "w"), XDR_ENCODE) ;
    xdr_wrapstring (&x, &b) ;
    xdr_destroy (&x) ; /* effectue le fflush() */
    fclose (f) ;
    close (sock) ;
}

/*
 * rxdr.c: recepateur avec XDR (usage : rxdr)
 */
#include <stdio.h>
#include <rpc/xdr.h>

#include "utilInet.h"

void main (void)
{
    int sock = mkServeur (SOCK_DGRAM, "impression") ;
    XDR x ;
    FILE *f ;

    shutdown (sock, 1) ;
    xdrstdio_create (&x, fdopen (sock, "r"), XDR_DECODE) ;
    while (1) {
        char *b = NULL ;

        xdr_wrapstring (&x, &b) ;
        fprintf (stderr, "le recepateur a recu: %s\n", b) ;
        xdr_free (xdr_wrapstring, &b) ;
        sleep (2) ;
    }
    xdr_destroy (&x) ;
}

```


6.5.3 XDR de structures contenant un tableau

Un exemple un peu plus compliqué à cause du tableau `p` qui appartient à la structure : pour transmettre des piles

```
struct pile {
    int nb ;
    complexe p [20] ;
} ;
```

La solution suivante est en fait erronée, puisque si on utilise un `xdr_free (xdr_pile, ...)`, le tableau qui est **dans** la structure sera libéré!

```
bool_t xdr_pile (XDR *xdrs, struct pile *p)
{
    complexe *pc = p->p ; /* xdr_array() a besoin d'un DOUBLE pointeur */

    /* on ne transmet que les "cases pleines" de la pile */
    return xdr_array (xdrs, &pc, &p->nb, 20, sizeof (complexe), xdr_complexe) ;
}
```

Une solution raisonnable consiste à ne pas utiliser `xdr_array()`. On écrit directement une boucle sur le tableau :

```
bool_t xdr_pile (XDR *xdrs, struct pile *p)
{
    int i ;

    /* on ne transmet que les "cases pleines" de la pile */
    if (! xdr_int (xdrs, &p->nb)) return 0 ;
    for (i = 0 ; i < p->nb ; i++) {
        if (! xdr_complexe (xdrs, &p->p [i])) return 0 ;
    }
    return 1 ;
}
```

6.5.4 XDR de liste chaînée

Pour transmettre une liste chaînée, le plus simple est d'écrire une XDR récursive, par exemple :

```
typedef struct elem *liste ;
bool_t xdr_liste (XDR *xdrs, liste *pl) ;

typedef struct elem {
    complexe c ;
    liste s ;
} elem ;

bool_t xdr_elem (XDR *xdrs, elem *pe)
{
    return xdr_complexe (xdrs, &pe->c) && xdr_liste (xdrs, &pe->s) ;
}
```

```

bool_t xdr_liste (XDR *xdrs, liste *pl)
{
/* rappelons-nous que si on est en decodage et que *pl est nul
 * xdr_pointer() alloue la zone receptrice puis la met a zero
 * ce qui assure que les elements suivant seront correctement alloues.
 */
    return xdr_pointer (xdrs, pl, sizeof (elem), xdr_elem) ;
}

```

Le malaise vient ici de l'exploration récursive de la liste qui peut être longue. L'algorithme non récursif ci-dessous utilise le double pointeur `ps` qui repère toujours le champ `s` du dernier élément transmis (encodé, décodé ou libéré) :

```

typedef struct elem *liste ;
bool_t xdr_liste (XDR *xdrs, liste *pl) ;

typedef struct elem {
    complexe c ;
    liste s ;
} elem ;

bool_t xdr_elem (XDR *xdrs, elem *pe) ;
{
    return xdr_complexe (xdrs, &pe->c) ; /* on ne transmet plus le pointeur */
}

bool_t xdr_liste (XDR *xdrs, liste *p)
{
    while (1) {
        if (xdrs->x_op == XDR_ENCODE || xdrs->x_op == XDR_DECODE) {
            if (! xdr_pointer (xdrs, p, sizeof (elem), xdr_elem))
                return FALSE ;
            if (*p == 0) return TRUE ;
            p = & (*p)->s ;
        } else if (xdrs->x_op == XDR_FREE) {
            if (*p != 0) {
                liste old = *p ;
                *p = (*p)->s ;
                if (! xdr_pointer (xdrs, &old, sizeof (elem), xdr_elem))
                    return FALSE ;
            } else return TRUE ;
        }
    }
}

```

Exercice les paramètres traditionnels `int argc` et `char *argv[]` de toute fonction `main()` représentent la commande dont on a demandé l'exécution au système. Ceci peut être une manière de mémoriser une commande. Ecrire les structures de données permettant de représenter un historique

de commandes, et les fonctions XDR associées.

Chapter 7

Appel de procédure éloignée SunOS 4.x (RPC)

7.1 Rappel

Les appels de procédures éloignées, ou RPC (Remote Procedure Call), permettent la communication interprocessus en utilisant un outil bien connu de la programmation structurée : le sous-programme.

Les deux processus ainsi mis en communication peuvent tourner sur une même machine ou sur deux machines reliées par un réseau.

En général, le processus qui fait l'appel de procédure est appelé **client**, le processus qui exécute le corps de la procédure est appelé **serveur**, une procédure fournie par un serveur est appelée **service**. Un serveur peut fournir plusieurs services.

Un appel RPC se déroule comme suit : l'objet donnée est convoyé par le réseau du client au serveur, le service construit l'objet résultat puis le renvoie vers le client.

Ces objets sont transmis avec le mécanisme XDR.

En règle générale, les appels de procédure sont donc synchrones.

7.2 Architecture des RPC de SunOS 4.x

7.2.1 Situation des RPC

Le fonctionnement des RPC est basé sur les sockets : les passages de paramètres utilisent des sockets connectées entre le client et le serveur. Les RPC correspondent à la couche session du modèle de l'OSI.

7.2.2 Nommage des services RPC

Sous SunOS 4.x, les services sont désignés par les cinq informations suivantes :

- nom de la machine supportant le serveur,
- nom du serveur (ou programme),
- version du programme : plusieurs versions d'un même serveur peuvent coexister sans ambiguïté,
- service demandé (un serveur propose plusieurs services),
- protocole utilisé par la (les) socket(s) sous-jacente(s).

Les noms de serveur sont en fait des numéros, les serveurs officiels comme NFS (Network File System) possèdent des numéros qu'il ne faut pas réutiliser. Voici une table résumant les 4 tranches de numéros disponibles.

limites de la tranche	usage
0x0000_0000 à 0x1FFF_FFFF	serveurs officiels enregistrés par Sun
0x2000_0000 à 0x3FFF_FFFF	libre
0x4000_0000 à 0x5FFF_FFFF	réservé aux serveurs transitoires
0x6000_0000 à 0xFFFF_FFFF	réservé par Sun

Les serveurs *transitoires* utilisent des numéros alloués dynamiquement et ne peuvent donc être connus qu'à l'exécution. Par exemple, si un serveur doit appeler une procédure de son client (il s'agit d'un rétro appel), le client demande un numéro de programme transitoire puis le communique à son serveur, celui-ci peut ensuite appeler le client en utilisant ce numéro.

Les numéros de des serveurs officiels se trouvent dans les fichiers du répertoire `/usr/include/rpcsvc`. Par exemple on trouve dans `rusers.h` :

```
#define RUSERSPROG 100002
```

qui est le numéro du serveur RUSERS, et dans `nfs_proto.x` écrit en `rpcgen` :

```
program NFS_PROGRAM {
    version NFS_VERSION {
        void NFSPROC_NULL(void) = 0 ;
        ...
    } = 2;
} = 100003;
```

ce qui indique que le programme NFS possède le numéro 100003, il n'y a que la deuxième version (NFS_VERSION) qui propose, entre autres, le service NFSPROC_NULL de numéro 0.

Un tel service de numéro 0 sans paramètre et ne retournant aucune valeur devrait en principe être fourni par tout serveur ; il permet de tester la présence du serveur et est utilisé par la commande `rpcinfo`.

Un serveur RPC peut être disponible avec un des protocoles TCP et UDP ou les deux.

7.2.3 Le processus portmap

Chaque machine susceptible de proposer des serveurs RPC doit supporter le démon portmap qui gère la table des serveurs RPC disponibles sur la machine. Une entrée de cette table a la forme suivante :

```
[numero de programme, numero de version, protocole, numero de port]
```

La commande `rpcinfo -p` imprime le contenu du portmap local.

Le portmap fournit des services permettant l'utilisation de sa table :

- créer une entrée (i.e. enregistrer un nouveau serveur),
- détruire une entrée (i.e. effacer un serveur enregistré),
- consultation d'une ou plusieurs entrées.

Le portmap est un serveur bien connu dont le numéro de port figure dans le fichier `/etc/services` (`sunrpc` dont le numéro de port est 111) pour les deux protocoles TCP et UDP ; il est donc accessible par tout processus.

Le portmap ne gère pas les numéros de service : ce sont les serveurs eux-mêmes qui s'en chargent.

Le portmap est indispensable pour qu'un client puisse localiser le serveur dont il a besoin.

7.2.4 L'enregistrement d'un serveur

Pour qu'un serveur soit callable il doit s'enregistrer (cf les fonctions `registerrpc()`, `svc_register()`, `pmap_set()`). Cette opération consiste d'une part à demander au portmap local de créer une nouvelle entrée et d'autre part à mémoriser localement la correspondance entre un numéro de service et l'adresse de la fonction qui l'implémente ou bien l'adresse de la fonction *dispatcher* suivant le niveau de primitive utilisé.

Avant de se terminer, un serveur devrait l'indiquer à son portmap (cf les fonctions `svc_unregister()`, `pmap_unset()`).

7.2.5 Le mécanisme d'appel vu du client

Côté client, un appel de service se passe en deux temps :

1. consultation du portmap de la machine distante censée supporter le serveur pour récupérer le numéro de port du serveur (cf les fonctions `clnt_create()`, `clnttcp_create()`, `clntudp_create()` et `pmap_getport()`),
2. il est ensuite possible de faire plusieurs demandes de services au serveur (cf les fonctions `clnt_call()`).

La fonction de haut niveau `callrpc()` enchaîne de façon transparente ces deux étapes.

7.2.6 Sémantique des appels

La **sémantique** d'un appel RPC se rapporte au nombre d'exécutions effectives d'un service lors de son appel. Celle-ci varie en fonction de la fiabilité du protocole de communication utilisé (en l'occurrence **udp** ou **tcp**).

En `tcp`, la communication est fiable. Si l'appel réussit, le client est sûr que le serveur a exécuté exactement une fois le service. Si l'appel échoue, il peut y avoir plusieurs causes, le serveur était indisponible et il n'y a eu aucune exécution, ou bien il y a eu déclenchement du délai de garde côté client, dans ce cas il est possible que le service ait été effectivement honoré par le serveur trop lent (sémantique 0 ou 1).

En `udp` la communication n'est pas fiable. Un appel est abandonné par le client, avec diagnostic d'erreur, si aucun résultat n'est arrivé au terme d'un *délai de garde total*. Pendant ce délai, la demande de service est relancée à chaque expiration d'une *période de relance* si aucun résultat n'est arrivé. Ainsi, un seul appel peut donner lieu à plusieurs exécutions du service (par exemple si le serveur est très lent à répondre). le nombre maximum d'exécutions du service est environ le rapport du délai de garde sur la période. Ceci signifie que si un appel réussit le client est assuré que le service a été exécuté **au moins** une fois, si l'appel échoue il se peut que le service ait été exécuté zéro, une ou plusieurs fois !

La sémantique d'appel pose problème dès que l'exécution d'un service est amené à modifier l'état du serveur (dans ce cas il vaut mieux utiliser `tcp`).

7.3 Primitives RPC de haut niveau

Les primitives de haut niveau sont simples à utiliser mais peu flexibles. Cette couche ne fonctionne qu'avec le protocole UDP, ce qui limite la taille des paramètres après encodage à environ 8K. Ceci a aussi pour conséquence que la sémantique des appels sera zéro, une ou plusieurs fois.

Son attrait principal est sa grande simplicité de mise en œuvre.

7.3.1 Le client

Un client dispose des deux primitives

7.3.1.1 callrpc()

```
enum clnt_stat callrpc (
    char *host,
    u_long prognum, u_long versnum, u_long procnum,
    xdrproc_t inproc, char *in,
    xdrproc_t outproc, char *out
);
/* renvoie RPC_SUCCESS (cf rpc/clnt.h) si OK */
```

qui effectue l'appel distant du service `procnum` de la version `versnum` du programme `prognum` sur la machine `host`.

`in` est un pointeur sur l'objet donnée qui sera envoyé avec l'XDR `inproc`. `out` est un pointeur sur l'objet résultat qui sera reçu avec l'XDR `outproc`.

7.3.1.2 clnt_perrno()

```
void clnt_perrno (enum clnt_stat stat);
```

qui imprime un message d'erreur approprié en cas d'échec de `callrpc()`.

7.3.2 Le serveur

Un serveur dispose de :

7.3.2.1 registerrpc()

```
registerrpc (
    u_long prognum, u_long versnum, u_long procnum,
    char *(*procname) (),
    xdrproc_t inproc, xdrproc_t outproc
);
```

où `procname` est le code de la procédure `procnum` de la version `versnum` du programme `prognum`, et `inproc` et `outproc` sont les fonctions XDR de l'objet donnée et de l'objet résultat. Voici le prototype général de `procname` :

```
TYPE_RESULTAT *procname (TYPE_DONNEE *);
```

Cette fonction doit être utilisée autant de fois qu'il y a de services dans le serveur. et de :

7.3.2.2 `svc_run()`

```
svc_run ();
```

implémente la boucle sans fin du serveur : elle attend une demande de service, tente de la satisfaire puis attend la demande suivante...

7.3.3 Exemple : un compte bancaire

Il s'agit d'un serveur de compte (bancaire ?) proposant une opération élémentaire et la consultation du compte. On a intérêt à se donner un fichier d'entête commun au serveur et au client :

```
/*
 * interf.c: interface commun au compte et a ses clients
 */
#include <stdio.h>
#include <rpc/rpc.h>

#define PROG_NUM 0x20000000 /* numero du programme: le premier non officiel */
#define VERS_NUM 1          /* version du programme */

#define NULL_PROC 0

/* liste des operations disponibles sur un compte */
#define OPE_NUM 1
#define SEE_NUM 2
```

Le code du serveur est alors le suivant :

```
/*
 * compte.c: Un compte sur lequel on fait des operations ou qu'on consulte
 */
#include "interfc.h"

long * ope (int *val) ;          /* les deux services */
long * consult (void) ;         /* proposes */

void main (void)
{
    if (registerrpc
        (PROG_NUM, VERS_NUM, OPE_NUM, ope, xdr_int, xdr_long) == -1
        ||registerrpc
        (PROG_NUM,VERS_NUM,SEE_NUM,consult,xdr_void,xdr_long)==-1){
        perror ("registerrpc") ; exit (1) ;
    }
    svc_run () ;
}

static long compte = 0 ; /* contient la valeur courante du compte */
```

```

long *ope (int *val)
{
    compte += *val ;
    return &compte ;
}
long *consult (void)
{
    return &compte ;
}

```

on remarque que les fonction `ope()` et `consult()` qui implantent les services ont comme paramètre un *pointeur* sur l'objet `in` et renvoient aussi un *pointeur* sur l'objet résultat.

Voici le code d'un client possible (celui-ci suppose que le serveur tourne sur la machine `homel`) :

```

/*
 * ope.c: Un client du serveur de compte
 *
 * usage : ope valeur
 */
#include "interfc.h"

void main (int argc, char *argv []) ;
{int val, stat ;
 long compte ;

    val = atoi (argv [1]) ;
    stat = callrpc ("homel", PROG_NUM, VERS_NUM, OPE_NUM,
                   xdr_int, &val,
                   xdr_long, &compte) ;
    if (stat != RPC_SUCCESS) clnt_perrno (stat) ;
    else printf ("valeur du compte: %ld\n", compte) ;
}

```

7.4 Primitives RPC de bas niveau

Elles sont plus complexes que les primitives de haut niveau, mais aussi plus riches en fonctionnalités :

- choix du protocole de transport (UDP ou TCP),
- contrôle des délais de garde,
- maîtrise de l'allocation/libération mémoire par les fonctions XDR,
- possibilité d'identification des clients.
- efficacité accrue du fait que la consultation du portmap n'a lieu qu'une seule fois pour plusieurs appels de service.

7.4.1 Le client

Les structures de données :

```

CLIENT * ;
enum clnt_stat ;
struct timeval { /* sys/time.h */
    long tv_sec ;
        /* seconds */
    long tv_usec ;
        /* and microseconds */
};

```

Le type CLIENT permet de conserver la localisation du serveur après consultation du portmap. Les primitives de gestion des clients, en particulier, les primitives de création renvoient le pointeur nul s'il y a eu une erreur, d'autre part, sur un client UDP ne peuvent circuler que des paramètres de taille inférieure à 8K :

7.4.1.1 clnt_create ()

```

CLIENT * clnt_create (
    char *host,
    u_long prognum,
    u_long versnum,
    char *protocol /* "udp" ou "tcp" */);

CLIENT * clnttcp_create (
    struct sockaddr_in *addr,
        /* addr->sin_port == 0 ==> consultation du portmap distant */
    u_long prognum,
    u_long versnum,
    int *sockp, /* *sockp == RPC_ANYSOCK ==> creation d'une socket */
    u_int sendsz, u_int recvsz /* 0, 0 ==> valeurs par défaut */);

CLIENT * clntudp_create(
    struct sockaddr_in *addr,
        /* addr->sin_port == 0 ==> consultation du portmap distant */
    u_long prognum,
    u_long versnum,
    struct timeval periode_de_relance,
    int *sockp); /* *sockp == RPC_ANYSOCK ==> creation d'une socket */

```

Ces trois primitives interrogent le portmap éloigné et construisent la structure CLIENT.

7.4.1.2 clnt_control ()

```

void clnt_control (CLIENT *clnt, const u_int req, char *info);

```

Permet, par exemple, de changer la période de relance :

```

struct timeval retry = {1, 0} ;
clnt_control (c, CLSET_RETRY_TIMEOUT, &retry) ;

```

Ceci n'est possible qu'en datagramme.

7.4.1.3 clnt_pcreateerror ()

```
void clnt_pcreateerror (char *str);
```

imprime un message d'erreur approprié après une erreur d'interrogation du portmap.

7.4.1.4 clnt_destroy()

```
void clnt_destroy (CLIENT *clnt);
```

Pour détruire un client.

7.4.1.5 clnt_call()

La primitive d'appel de service est

```
enum clnt_stat clnt_call (
    CLIENT *clnt;
    u_long procnum;
    xdrproc_t inproc,
    char *in,
    xdrproc_t outproc;
    char *out;
    struct timeval delai_de_garde_total);
```

Dans le protocole UDP, le nombre d'essais d'un appel éloigné avant de considérer qu'il y a échec est en gros de `delai_de_garde_total/periode_de_relance`.

En TCP, il n'y a qu'un délai de garde total puisque le transport est fiable.

7.4.1.6 clnt_freeres()

```
bool_t clnt_freeres (CLIENT *clnt, xdrproc_t outproc, char *out);
```

Pour restituer la mémoire allouée dynamiquement par l'XDR de décodage `outproc`.

7.4.1.7 clnt_perror()

```
void clnt_perror (CLIENT *clnt, char *str);
```

Pour imprimer un message lors d'une erreur d'appel.

7.4.2 Le serveur

Structure de donnée d'un serveur dans `<rpc/svc.h>` :

```
SVCXPRT
```

7.4.2.1 svctcp_create()

Création d'un serveur :

```
SVCXPRT * svctcp_create (
    int sock, /* RPC_ANYSOCK ==> creation d'une socket */
    u_int sendsz, recvsz /* 0, 0 ==> valeurs par défaut */);

SVCXPRT * svcudp_bufcreate (
    int sock, /* RPC_ANYSOCK ==> creation d'une socket */
    u_int sendsz, recvsz /* PAS de valeurs par défaut */);

SVCXPRT * svcudp_create (RPC_ANYSOCK);

void svc_destroy (SVCXPRT *xpirt);
```

7.4.2.2 svc_register()

Enregistrement du *dispatcher* d'un serveur :

```
bool_t svc_register (
    SVCXPRT *xpirt,
    u_long prognum, versnum,
    void (*dispatcher) (struct svc_req *requete, SVCXPRT *),
    u_long protocol /* IPPROTO_UDP ou bien IPPROTO_TCP */);

void svc_unregister (u_long prognum, u_long versnum);
```

enregistre le *dispatcher* correspondant à la version *versnum* du programme *prognum*.

7.4.2.3 Le dispatcher

Le *dispatcher* est une routine spécifique à l'application, son rôle est d'aiguiller les appels des clients sur les routines effectives de service.

Le prototype d'un dispatcher doit être

```
void dispatcher (struct svc_req *req, SVCXPRT *X);
```

Le dispatcher doit :

1. identifier le service demandé grâce à `req->rq_proc`, qui est le numéro de service demandé.
2. récupérer l'objet donnée dans `in` avec :

```
bool_t svc_getargs (SVCXPRT *xpirt, xdrproc_t inproc, char *in);
```

3. exécuter le code de service qui fabrique l'objet résultat `out`,
4. renvoyer l'objet résultat avec :

```
bool_t svc_sendreply (SVCXPRT *xpirt, xdrproc_t outproc, char *out);
```

5. libérer l'objet donnée :

```
bool_t svc_freeargs (SVCXPRT *xpirt, xdrproc_t obfproc, char *obj) ;
```

De plus, on dispose de :

```
void svcerr_noproc (SVCXPRT *xpirt) ; /* service inexistant */
```

si le service n'existe pas, et de :

```
void svcerr_decode (SVCXPRT *xpirt) ;
```

si `svc_getargs()` échoue.

Le prototype d'une fonction de service est :

```
TYPE_RESULTAT *service (TYPE_DONNEE *donnee) ;
```

7.4.3 Exemple : le compte bancaire

Il s'agit toujours du compte bancaire : programmons le serveur à un niveau plus fin (le fichier `interfc.h` est inchangé) :

```
/*
 * compte.c: Un compte sur lequel on fait des operations ou qu'on consulte
 */
#include "interfc.h"

void dispatcher (struct svc_req *rqstp, SVCXPRT *transp) ;

void main (void)
{SVCXPRT *transp ;

  transp = svcudp_create (RPC_ANYSOCK) ;
  if (transp == NULL) {
    fprintf (stderr, "creation du serveur UDP impossible\n") ;
    exit (1) ;
  }
  pmap_unset (PROG_NUM, VERS_NUM) ;
  if (! svc_register (transp, PROG_NUM, VERS_NUM, dispatcher, IPPROTO_UDP)) {
    fprintf (stderr, "enregistrement du serveur impossible\n") ;
    exit (1) ;
  }
  svc_run () ;
}

static long compte = 0 ; /* contient la valeur courante du compte */

long *ope (int *val)
{
  compte += *val ;
}
```

```

    return &compte ;
}

void dispatcher (struct svc_req *rqstp, SVCXPRT *transp)
{
    switch (rqstp->rq_proc) {
    case NULL_PROC: /* appelee pour verifier la presence du serveur */
        if (! svc_sendreply (transp, xdr_void, NULL))
            fprintf (stderr, "renvoi au client impossible\n") ;
        return ;
    case SEE_NUM:
        if (! svc_sendreply (transp, xdr_long, &compte))
            fprintf (stderr, "renvoi au client impossible\n") ;
        return ;
    case OPE_NUM:
        {int v ;
         if (! svc_getargs (transp, xdr_int, &v)) {
             svcerr_decode (transp) ;
             return ;
         }
         if (! svc_sendreply (transp, xdr_long, ope (&v)))
             fprintf (stderr, "renvoi au client impossible\n") ;
         svc_freeargs (transp, xdr_int, &v) ;
        }
        return ;
    default:
        svcerr_noproc (transp) ;
        return ;
    }
}

```

Remarque : on a réutilisé la fonction `ope()` telle que définie dans l'exemple illustrant les RPC de haut niveau.

Cette fois on aurait pu utiliser le protocole TCP plutôt que UDP, (ou même les deux, il faut créer deux structures `SVCXPRT` et enregistrer le dispatcher sur les deux). La procédure `dispatcher` consulte elle-même la demande puis l'aiguille sur le bon service ; le cas `NULL_PROC` permet à un client de tester la présence d'un serveur (tous les serveurs officiels implémentent ce service minimum avec le numéro 0).

La version haut niveau du client peut être utilisée telle quelle avec ce nouveau serveur.

Exercice : programmer le client du compte bancaire avec les primitives de bas niveau.

7.5 RPC non bloquant

Certains services peuvent être rendus non bloquants, voici les conditions qui autorisent ce fonctionnement :

- protocole fiable (TCP),
- le serveur n'exécute pas `svc_sendreply()` en fin de service,

- côté client, la routine de décodage des résultats `outproc` doit être NULL et le délai de garde doit être nul.

Cette technique permet d'augmenter les performances lors de transferts volumineux. Pour resynchroniser le client et son serveur il est toujours possible d'appeler un service bloquant.

Exercice illustrer ce gain en performance pour transmettre un gros fichier à un serveur par bloc de 1024 (faites des mesures d'abord en bloquant puis en non bloquant).

7.6 Diffusion d'appel

Il est possible de diffuser un appel de service à tous les réseaux de diffusion connectés localement avec la primitive :

```
enum clnt_stat clnt_broadcast(
    u_long prognum,
    u_long versnum,
    u_long procnum,
    xdrproc_t inproc, char *in,
    xdrproc_t outproc, char *out,
    resultproc_t eachresult);

typedef int (*resultproc_t) (char *out, struct sockaddr_in *addr);
```

Le seul protocole utilisable est UDP et la taille des requêtes doit être inférieure à 1500 octets. La primitive `eachresult()` est appelée pour chaque réponse reçue. Le `out` de `eachresult` désigne ce résultat et est le même que le `out` de `clnt_broadcast`. `addr` est l'adresse de la machine qui répond. `eachresult()` doit renvoyer vrai pour arrêter la diffusion.

Par défaut, `clnt_broadcast()` utilise le système d'authentification Unix.

Exercice écrire un programme qui imprime les noms des 5 premières machines qui proposent un serveur donné.

Exercice écrire une commande qui imprime les noms des machines sur lesquelles est connecté un utilisateur donné (voir `rusers(3)`).

7.7 Rétro appel

Un client peut s'allouer dynamiquement un numéro transitoire de programme auprès de son portmap local grâce à `pmap_set()`. Il le communique à son serveur qui peut ensuite lui demander des services.

7.8 RPC sécurisés

Les RPC permettent à un serveur d'identifier les usagers qui lui demandent un service.

7.8.1 Exemple : le compte bancaire sécurisé

Le but est d'assurer que le client qui manipule le compte est le propriétaire du compte bancaire (i.e. du serveur), en fait seule l'opération `ope` est ici sécurisée.

Voici le code du serveur :

```

int c_est_moi (struct svc_req *rq)
{
    if (rq->rq_cred.oa_flavor == AUTH_UNIX) {
        return ((struct authunix_parms *) rq->rq_clntcred)->aup_uid == getuid () ;
    } else return 0 ;
}

void dispatcher (struct svc_req *rqstp, SVCXPRT *transp)
{
    switch (rqstp->rq_proc) {
        ...
        case SEE_NUM: /* operation non securisee */
            if (! svc_sendreply (transp, xdr_long, &compte))
                fprintf (stderr, "renvoi au client impossible\n") ;
            return ;
        case OPE_NUM: /* operation securisee */
            if (c_est_moi (rqstp)) {int v ;
                if (! svc_getargs (transp, xdr_int, &v)) {
                    svcerr_decode (transp) ;
                    return ;
                }
                if (! svc_sendreply (transp, xdr_long, ope (&v)))
                    fprintf (stderr, "renvoi au client impossible\n") ;
                svc_freeargs (transp, xdr_int, &v) ;
            } else {
                svcerr_weakauth (transp) ;
            }
            return ;
        ...
    }
}

```

De son côté, le client doit s'identifier avant d'effectuer un appel :

```

#include "interfc.h"

main (int argc, char *argv [])
{
    CLIENT * c = clnt_create ("homel", PROG_NUM, VERS_NUM, "udp") ;
    int v = atoi (argv[1]) ;
    long compte ;
    struct timeval delai ;

    delai.tv_sec = 20 ; delai.tv_usec = 0 ;

    if (c == NULL) {
        clnt_pcreateerror ("serveur de compte") ; exit (1) ;
    }
}

```

```

    }

    /* detruire l'information d'identification par default (rien) */
    auth_destroy (c->cl_auth) ;

    /* creer l'information d'identification */
    c->cl_auth = authunix_create_default () ;

    if (clnt_call (c, OPE_NUM, xdr_int, &v, xdr_long, &compte, delai)
        == RPC_SUCCESS) {
        printf ("compte = %ld\n", compte) ;
    } else {
        clnt_perror (c, "ope") ;
    }

    /* detruire l'information d'identification */
    auth_destroy (c->cl_auth) ;

    clnt_destroy (c) ;
}

```

Exercice : examiner la doc de `authunix_create_default()` et `authunix_create()` pour shunter facilement la protection `AUTH_UNIX`.

Exercice : utilisez `clnt_control()` dans un client du compte bancaire et la primitive `sleep(3V)` dans le serveur pour mettre en évidence les sémantiques d'appel : en `udp`, un seul appel peut donner lieu à plusieurs exécutions par le serveur, en `tcp`, l'exécution peut être effective même si le client n'a pas eu de réponse (atteinte du délai de garde chez le client).

Chapter 8

Le générateur RPCGEN

Rpcgen propose d'une part un langage de description d'interface pour l'utilisation des appels de procédure éloignée (RPC) et d'autre part, un compilateur de ce langage permettant d'obtenir les sources C des procédures souches destinées aux clients et un squelette du serveur défini par l'interface.

En particulier le langage permet la description dans une syntaxe proche de celle du C des différentes structures de données correspondant aux paramètres des procédures éloignées, et celle de la liste des services fournis par le serveur.

Il reste alors au programmeur à écrire le code des procédures du serveur et le code d'un ou de plusieurs clients éventuel utilisant les souches fournies par rpcgen.

Ces intérêts sont les suivant :

- produire un interface commun client/serveur,
- fabrication automatique du source C des fonctions XDR d'après la description des structures de données,
- fabrication automatique du squelette du serveur avec introduction du service 0 permettant de tester la présence du serveur,
- fabrication en C des "souches" d'appel du client (stub),
- possibilité de modifier après coup les sources C obtenus.

La fabrication d'une application peut se résumer à trois activités :

1. conception de l'interface public du serveur qui aboutit à la rédaction d'un fichier `.x` en langage rpcgen,
2. implantation des fonctions du serveur,
3. écriture du code des clients.

Cette démarche est illustrée dans la suite sur un annuaire.

8.1 Ecriture de l'interface en rpcgen

Voici le source rpcgen du fichier `annuaire.x` :

```
/*  
 * annuaire.x  
 */  
  
const MAX = 100 ;
```

```

struct ENTREE {
    string nom <MAX> ;
    int tel [4] ;
} ;

enum STATUT {EXISTE, INEXISTANT} ;

union REPONSE switch (STATUT err) {
    case EXISTE :
        ENTREE e ;
    case INEXISTANT :
        void ;
} ;

program ANN_PROG {
    version ANN_VERS {
        REPONSE get (string) = 1 ;
        void set (ENTREE) = 2 ;
    } = 1 ;
} = 99 ;

```

après la commande

```
rpcgen annuaire.x
```

on obtient les fichiers suivants :

annuaire.h contient la traduction en C des structures de données, les constantes symboliques homonymes des noms de service et les prototypes des fonctions de service correspondant aux déclarations du fichier `.x`, voici un extrait de ce fichier :

```

/* Please do not edit this file. It was generated using rpcgen. */
...
#define MAX 100

struct ENTREE {char *nom ; int tel[4];};
typedef struct ENTREE ENTREE;

enum STATUT {EXISTE = 0, INEXISTANT = 1,};
typedef enum STATUT STATUT;

struct REPONSE {
    STATUT err;
    union {
        ENTREE e;
    } REPONSE_u;
};
typedef struct REPONSE REPONSE;

#define ANN_PROG ((u_long)99)
#define ANN_VERS ((u_long)1)
...

```

annuaire_clnt.c contient le code des souches destinées aux programmes clients, et dont voici les prototypes

```
REPONSE * get_1 (char **argp, CLIENT *clnt) ;
```

```
void * set_1 (ENTREE *argp, CLIENT *clnt) ;
```

Le "1" après le nom de fonction correspond au numéro de version du serveur.

annuaire_svc.c contient le squelette du serveur disponible en udp et tcp dont il reste à écrire les services en respectant les prototypes :

```
REPONSE * get_1 (char **, struct svc_req *) ;
```

```
void * set_1 (ENTREE *, struct svc_req *) ;
```

Le "1" après le nom de fonction correspond au numéro de version du serveur.

annuaire_xdr.c contient le code des fonctions XDR de chacun des types définis.

Par la suite, le seul fichier qu'il est vraiment intéressant de consulter est celui des structures de données **annuaire.h**.

8.2 Ecriture des services

On peut ranger dans **services.c** le code des différents services, par exemple :

```
void * set_1 (ENTREE *abonne, struct svc_req *inutile_ici)
{
    /* ranger le nouvel abonne dans une table interne */
    ...
    return 0 ;
}
```

Puis on obtient un serveur exécutable par

```
acc -o sv      annuaire_svc.c  services.c  annuaire_xdr.c
```

8.3 Ecriture d'un client

Les souches d'accès au serveur ont un paramètre de type **CLIENT** qu'il faut fabriquer au préalable :

```
/* cl.c */
#include "annuaire.h"

main (int argc, char *argv[])
{
    CLIENT *cl = clnt_create ("homel", ANN_PROG, ANN_VERS, "udp") ;
    REPONSE *rep ;

    rep = get_1 (argv [1], cl) ;
```

```
switch (rep->err) {
  case EXISTE:
    printf ("%s : ", rep->REPONSE_u.e.nom) ;
    for (i = 0 ; i < 4 ; i++) {
      printf ("%d ", rep->REPONSE_u.e.tel [i]) ;
    }
    break ;
  case INEXISTANT:
    printf ("%s: n'est pas abonne\n", argv [1]) ;
}
clnt_freeres (cl, xdr_REPONSE, rep) ;
clnt_destroy (cl) ;
}
```

Remarque, il est toujours de la responsabilité du programme client de libérer les structures de données éventuellement allouées par les fonctions XDR lors du décodage.

Puis on obtient l'exécutable du client par

```
acc -o cl cl.c annuaire_clnt.c annuaire_xdr.c
```

Chapter 9

L'interface TLI

L'interface TLI (Transport Layer Interface) se situe au même niveau que les sockets. Il correspond au standard ISO 8072 et se veut indépendant de tout protocole ou toute famille de protocoles de la couche transport. Il peut donc fonctionner au dessus de la famille TCP/IP mais aussi au dessus de tout autre famille.

A l'origine, il a été implanté dans Unix System 5 version 3. Il est émulé dans SunOS 4.2 et intégré, au dépend des sockets, dans SunOs 5.x (i.e. SVR4).

9.1 Architecture des TLI

Un utilisateur des TLI peut demander à utiliser les services d'un fournisseur de transport (**transport provider** ou **TP**) quelconque, par exemple `/dev/tcp`. L'entité manipulée est alors le point de transport (**transport endpoint** ou **tep**).

Par l'intermédiaire du tep, l'utilisateur a accès à un certain nombre de **requêtes** destinées au TP. D'autre part, des **événements** en provenance du TP sont consultables par l'utilisateur par la fonction `t_look()`.

9.1.1 Fournisseurs de transport ou TP (Transport Provider)

Un fournisseur de transport particulier est caractérisé par le style de communication qu'il autorise (voir la table 9.1 et `t_getinfo()`). Plusieurs fournisseurs de transport sont en général disponibles

T_CLTS	communication sans connexion (i.e. udp)
T_COTS	communication avec connexion (i.e. tcp) et déconnexion brutale
T_COTS_ORD	communication avec connexion et déconnexion sans perte de message (i.e. tcp sous SVR4)

Figure 9.1: Les différents styles de communication.

sur le système, ils sont désignés par une entrée dans le répertoire `/dev`, on trouve entre autres :

udp et **tcp** , qui sont des TP Internet,

ticlts, **ticots** et **ticotsord** qui proposent les différents styles de communication de la figure 9.1, mais uniquement localement à la machine : il n'y a pas de communication réseau.

9.1.2 Ouverture d'un tep : `t_open()`

Il est possible d'obtenir un tep pour un fournisseur de transport avec `t_open()` comme dans :

```
int tep = t_open ("/dev/tcp", O_RDWR, 0);
```

qui fournit un tep sur le TP `tcp`.

9.1.3 Publication d'un tep : `t_bind()`

Pour qu'un tep soit accessible de l'extérieur, il faut, comme pour les sockets, le publier en lui attachant une adresse. La primitive `t_bind()` effectue ce travail (`t_unbind()` en annule l'effet). Par exemple

```
t_bind (tep, 0, 0);
```

associe une adresse appropriée au tep, celle-ci n'étant pas explicitement indiquée par l'appelant (car le deuxième paramètre est 0) sera dynamiquement allouée par le TP.

Le format de l'adresse dépend, bien entendu, du TP utilisée ; ainsi, pour `tcp` ou `udp`, les adresses sont du type bien connu `struct sockaddr_in`.

9.1.4 Les événements : `t_look()`

Dès qu'un tep possède une adresse, des processus externes peuvent lui envoyer des requêtes (envoi de message, demande de connexion, ...). Le propriétaire du tep voit ces requêtes comme des événements, en général asynchrones avec son propre déroulement, qu'il doit détecter et, en principe, traiter. Voici quelques-uns de ces événements (voir `t_look()` pour la liste complète) :

T_DATA	des données sont arrivées
T_DISCONNECT	une déconnexion brutale est arrivée
T_ORDREL	une déconnexion en douceur est arrivée

On peut détecter (et traiter) ces événements soit de façon *synchrone* avec le programme en utilisant le fait qu'un événement provoque l'échec des primitives TLI, soit de façon *asynchrone* en utilisant le signal `SIGPOLL` qui peut être envoyé par le TP lorsqu'un événement se produit :

détection synchrone si une primitive TLI échoue (renvoie -1) on regarde si la variable globale `t_errno` contient la valeur `TLOOK`. Si c'est le cas alors un événement s'est produit.

détection asynchrone on arme le signal `SIGPOLL` et on demande au TP de déclencher `SIGPOLL` sur certains de ces événements (voir `streamio(7)` avec `I_SETSIG` comme valeur du paramètre `command` de la primitive `ioctl()`).

Lorsqu'un événement a été détecté par l'une ou l'autre technique, on peut utiliser `t_look(tep)` pour connaître sa nature et le traiter.

Voici un exemple de traitement synchrone d'événement dans le cas d'une déconnexion en douceur (communication `T_COTS_ORD`) alors que le processus est en attente de réception :

```
...
if (t_rcv (tep, buf, lgbuf, 0) == -1) {
    if (t_errno == TLOOK && t_look (tep) == T_ORDREL) {
        t_rcvrel (tep) ;
        t_sndrel (tep) ;
    }
}
```

```

        goto fin ;
    } else {
        t_error ("t_rcv()") ; exit (1) ;
    }
} else { /* traitement du message reçu */
    ...
}
fin:

```

9.1.5 Les états

Le fonctionnement d'un tep et les primitives qui lui sont applicables sont parfaitement définis, d'une part, par le style de communication supportée par le TP sous-jacent, d'autre part, par une table de transitions qui définit l'évolution de l'état d'un tep en fonction des primitives qui lui sont appliquées. L'état d'un tep peut être examiné avec la primitive `t_getstate ()`. Les trois figures 9.2, 9.3 et 9.4, présentent une version simplifiée de cette table de transition pour chacun des trois styles de communication. Dans chacune de ces figures, les flèches hachurées représentent la transmission d'un événement depuis la primitive qui le déclenche jusqu'à la primitive qui le traite.

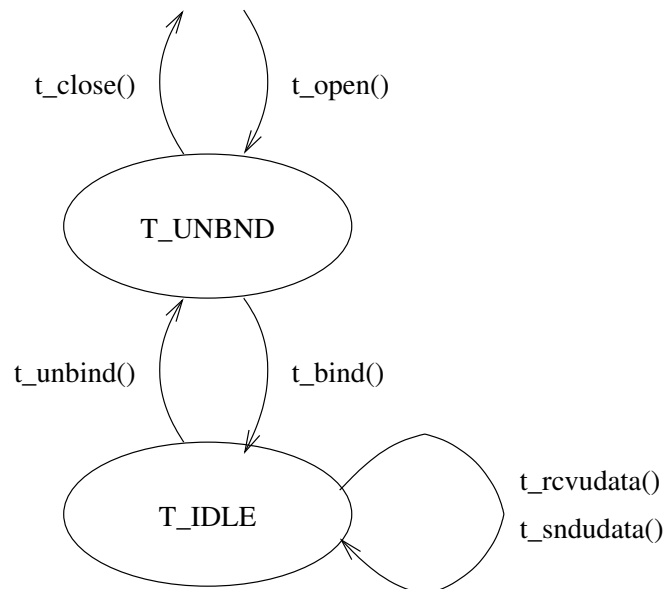


Figure 9.2: Enchaînement des états pour une communication sans connexion (T_CLTS).

Quand il n'y a pas de connexion (figure 9.2), rien ne distingue le serveur du client : ils ont tous deux les mêmes transitions.

Quand il y a connexion, le serveur peut communiquer soit sur le tep de connexion, soit sur un nouveau tep créé à cet effet et associé au tep du client (voir `t_accept()`). Dans les figures 9.3 et 9.4, c'est cette dernière possibilité qui est illustrée, c'est pourquoi le côté serveur se compose de deux graphes. On remarque que n'importe lequel des deux partenaires peut avoir l'initiative de la déconnexion — brutale ou ordonnée —, ceci est représenté par les *ou bien*.

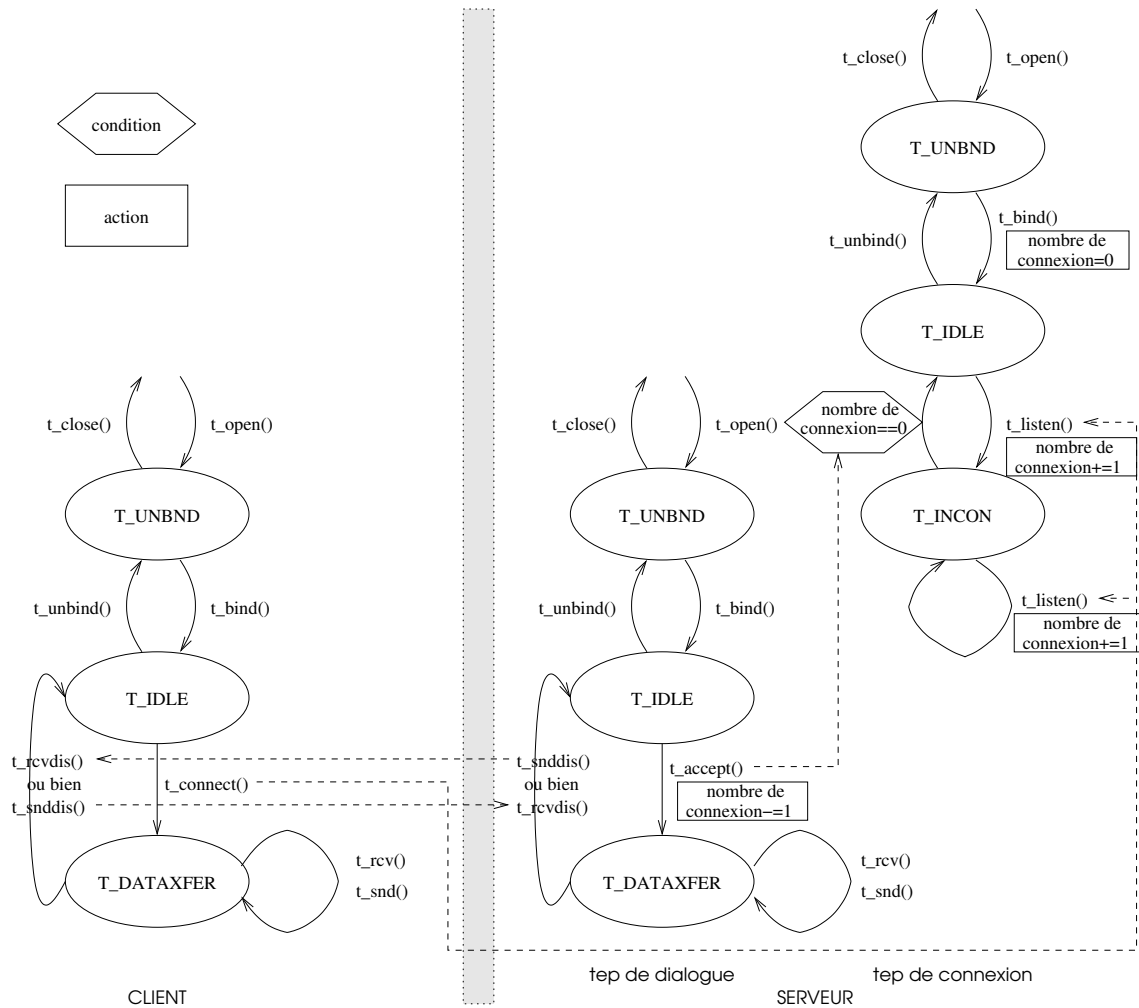


Figure 9.3: Enchaînement des états pour une communication avec connexion et déconnexion brutale (T_COTS). Au contraire des sockets, la primitive $t_listen()$ est bloquante jusqu'à réception d'une demande de connexion. La primitive $t_accept()$ permet d'honorer une demande de connexion et de fixer le **tep** de communication. $t_accept()$ doit donc être appelée après un $t_listen()$ réussi.

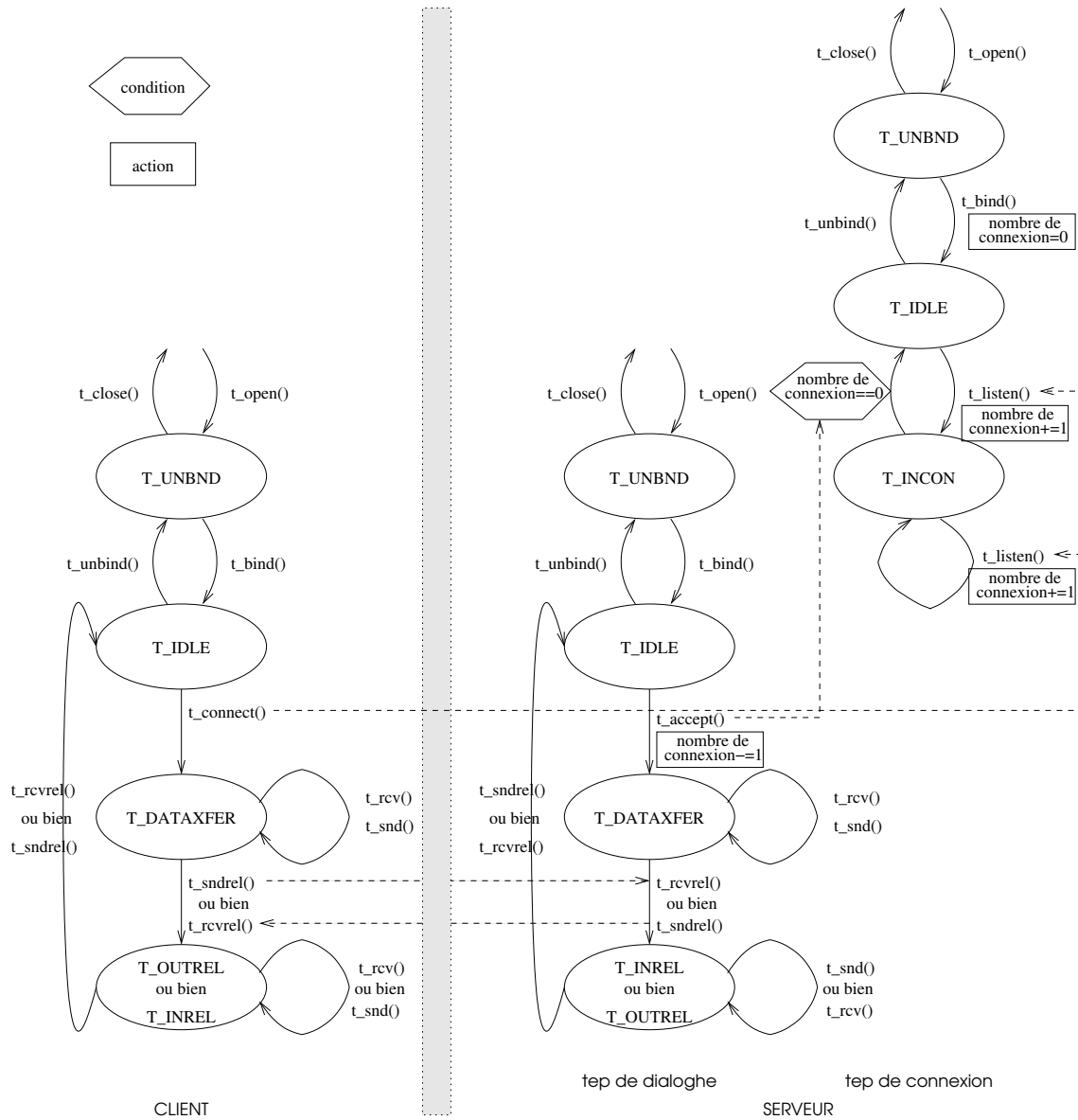


Figure 9.4: Enchaînement des états pour une communication avec connexion et déconnexion ordonnée (T_COTS_ORD). Au contraire des sockets, la primitive `t_listen()` est bloquante jusqu'à réception d'une demande de connexion. La primitive `t_accept()` permet d'honorer une demande de connexion et de fixer le tep de communication. `t_accept()` doit donc être appelée après un `t_listen()` réussi. On remarque que la déconnexion se fait en deux phases et qu'il est toujours possible de communiquer (dans un seul sens) entre ces deux phases : les réceptions sont possibles pour l'initiateur de la déconnexion, les émissions sont possibles pour celui qui a reçu la demande de déconnexion.

9.1.6 Les structures de données : t_alloc()

L'interface TLI doit pouvoir s'adapter à n'importe quel protocole, c'est pourquoi toute information est stockée dans un tableau flexible décrit par la structure `netbuf` :

```
struct netbuf {
    unsigned int maxlen; /* longueur max de buf */
    unsigned int len; /* longueur effective de buf */
    char *buf; /* la zone "flexible" */
};
```

`netbuf` est utilisée pour construire la plupart des données utilisées par les primitives TLI. Par exemple, le message envoyé par la primitive `t_sndudata()` lors d'une communication sans connexion (T_CLTS) a le format suivant :

```
struct t_unitdata {
    struct netbuf addr; /* adresse du destinataire */
    /*
     * (addr.buf est (struct sockaddr_in *) pour un TP udp
     */
    struct netbuf opt; /* options spécifiques au protocole */
    struct netbuf udata; /* message utile */
    /*
     * udata.maxlen == udata.len est la longueur de la
     * zone reperee par udata.buf
     */
};
```

La plupart des primitives TLI utilisent des structures de données spécifiques dont l'initialisation dépend du TP utilisé. La primitives `t_alloc()` alloue et initialise correctement ces dernières, la primitive `t_free()` permet de les libérer :

```
char * t_alloc (int tep, int struct_type, int fields);
int t_free (char *struct, int struct_type);
```

Le paramètre `struct_type` indique le type de structure à allouer ou à libérer, le paramètre `fields` indique les champs de la structure qu'il faut allouer et éventuellement initialiser (si `fields==T_ALL`, alors tous les champs doivent être initialisés).

Voici quelques-unes des valeurs de `struct_type` et les structures correspondantes (voir `t_alloc()` pour une liste complète) :

struct_type	structure associée
T_BIND	struct t_bind
T_CALL	struct t_call
T_DIS	struct t_dis
T_UNITDATA	struct t_unitdata
T_INFO	struct t_info

9.2 Environnement de développement

Pour utiliser TLI il faut inclure `tiuser.h` comme dans

```
#include <tiuser.h>
```

et compiler avec la bibliothèque `libnsl.a` comme dans

```
cc ... -lnsl
```

9.3 Un exemple de communication en T_COTS

Dans cet exemple, un client se connecte au serveur, puis lui envoie toutes les lignes lues sur l'entrée standard. Le serveur reçoit ces lignes et les imprime sur sa sortie standard. Le serveur sert un seul client puis se termine. C'est le client qui a l'initiative de la déconnexion. Le serveur détecte et traite de façon asynchrone la réception de et la demande de déconnexion.

Voici tout d'abord le module `tliInet.c` qui spécialise un certain nombre de primitives pour la famille TCP/IP et cache l'usage de `t_alloc()`.

```
#include "tliInet.h"
#include <netdb.h>
#include <netinet/in.h>

void td_bind_qlen (int tep, int qlen)
{
    struct t_bind
        *req = (struct t_bind *) t_alloc (tep, T_BIND, T_ALL),
        *rep = (struct t_bind *) t_alloc (tep, T_BIND, T_ALL) ;

    req->qlen = qlen ;
    t_bind (tep, req, rep) ;

    if (qlen) { /* c'est en principe un "serveur" sur lequel on se connecte */
        printf ("port = %d\n",
            ntohs (((struct sockaddr_in*) rep->addr.buf)->sin_port)) ;
    }
    t_free ((char *) req, T_BIND) ;
    t_free ((char *) rep, T_BIND) ;
}

struct t_call *td_listen (int listen_tep)
/* renvoie l'adresse du partenaire (peer), nécessaire pour le t_accept() */
{
    struct t_call *peer =
        (struct t_call *) t_alloc (listen_tep, T_CALL, T_ALL) ;

    t_listen (listen_tep, peer) ;
    return peer ;
}
```

```

}

int td_accept (int listen_tep, struct t_call *peer)
{
    int ntep = t_open ("/dev/tcp", O_RDWR, 0) ;

    t_bind (ntep, 0, 0) ;
    t_accept (listen_tep, ntep, peer) ;
    return ntep ;
}

void td_connect (int tep, const char *rhost, int port)
{
    struct t_call *sndcall = (struct t_call *) t_alloc (tep, T_CALL, T_ALL) ;

    {
        struct hostent *h = gethostbyname (rhost) ;
        struct sockaddr_in *a = (struct sockaddr_in *) sndcall->addr.buf ;

        bzero (a, sizeof (struct sockaddr_in)) ;
        a->sin_family = AF_INET ;
        a->sin_port = htons (port) ;
        bcopy ((char*) h->h_addr, (char*) &a->sin_addr, h->h_length);
    }
    sndcall->addr.len = sizeof (struct sockaddr_in) ;
    t_connect (tep, sndcall, 0) ;

    t_free ((char *) sndcall, T_CALL) ;
}

```

Voici le code du client

```

/* CLIENT, usage : cl port_du_serveur */
#include "tliInet.h"

main (int argc, char * argv[])
{
    int port = atoi (argv [1]) ;
    int tep = t_open("/dev/tcp", O_RDWR, 0) ;
    char buf [1024] ;

    t_bind (tep, 0, 0) ;
    td_connect (tep, "lifl", port) ;

    while (printf ("? "), gets (buf) != NULL) {
        t_snd (tep, buf, strlen (buf) + 1, 0) ;
    }
    t_snddis (tep, 0) ; t_unbind (tep) ; t_close (tep) ;
}

```

```

}
```

Voici le code du serveur

```

/* SERVEUR */
#include "tliInet.h"

#include <sys/types.h>
#include <stropts.h>
#include <signal.h>

static int listen_tep, comm_tep ;

void traite_evenement (int sig)
{
    signal (SIGPOLL, traite_evenement) ;
    switch (t_look (comm_tep)) {
        case T_DATA: {
            char buf [1024] ; int flags ;

            /* flags est necessaire mais pas utilise, voir le man */
            t_rcv (comm_tep, buf, sizeof buf, &flags) ;

            printf ("SERVEUR recu : %s\n", buf) ;
            break ;
        }
        case T_DISCONNECT:
            t_rcvdis (comm_tep, 0) ; t_unbind (comm_tep) ; t_close (comm_tep) ;
            t_unbind (listen_tep) ; t_close (listen_tep) ;
            exit (0) ;
            break ;
        default:
            t_error ("evenement non traite") ; exit (1) ;
    }
}

main (int argc, char * argv[])
{
    listen_tep = t_open("/dev/tcp", O_RDWR, 0) ;
    td_bind_qlen (listen_tep, 55) ;
    {
        comm_tep = td_accept (listen_tep, td_listen (listen_tep)) ;
        signal (SIGPOLL, traite_evenement) ;
        ioctl (comm_tep, I_SETSIG, S_INPUT) ;
        /* demande au TP d'envoyer SIGPOLL a chaque reception d'evenement */

        while (1) pause () ;
    }
}

```

}