Documentation technique sur l'utilisation des Sockets

Pascal Sicard

Table des matières

1 INTRODUCTION

L'interface d'accès au réseau via la couche transport d'Internet que nous utiliserons s'appelle les Sockets.Cette interface *socket* a été développée dans la version BSD du système Unix (Berkeley Sofware Distribution) sous l'impulsion de l'*Advanced Research Project Agency* (ARPA) à l'université de Californie à Berkeley. Elle est de fait au cours des années devenue un standard de communication.

Le concept de socket : Une *socket* est un point d'accès à un service de communication qui peut être réalisé par différentes familles [1] de protocole. Il est créé dynamiquement par un processus. Lorsque l'on crée une socket, on doit préciser la famille de protocoles de communication que l'on désire utiliser, ainsi que la type de service que l'on veut obtenir.

2 ENVIRONNEMENT DE PROGRAMMATION

Pour vous simplifier le travail (programmation de la gestion des erreurs, des aides à la mise au point...), une librairie de primitives appelant les "vraies" primitives des **sockets** vous ai fournie (voir **fon.h**) .

Ainsi, chaque procédure à votre disposition :

- encapsule une primitive « Socket » en conservant la même sémantique : la procédure de nom ***h_ primitive()*** encapsule la procédure de nom ***primitive()*** *(sauf adr_socket).*
- réalise les manipulations de données fastidieuses (*adr_socket*).
- réalise des contrôles sur les paramètres et rend des messages d'erreur.
- permet d'obtenir avec une option de compilation (voir **makefile**) des traces à l'exécution des primitives.

3 PARAMETRES UTILISÉS DANS LES PROCEDURES

Ces paramètres apparaissent dans les différentes procédures. Pour chacun d'eux, nous donnons la signification, le type en C et les différentes valeurs qu'il peut prendre.

3.1 Le domaine d'utilisation *(int domaine)*

Il indique la famille de protocoles de communication : pour TCP/IP, il prend la valeur **AF_INET**.
Remarque : Cette valeur est valable pour IPV4, dans le cas d'IPV6 le domaine est **AF_INET6**.

1. On dit également domaine, on parle ainsi du domaine Internet, du domaine ISO etc...

3.2 Le mode de communication *(int mode)*

C'est un entier qui représente le type de protocole transport utilisé sur une socket :
- mode connecté : **SOCK_STREAM** (TCP : flot d'octets)
- mode non connecté : **SOCK_DGRAM** (UDP : datagrammes)

3.3 L'identificateur de socket *(int num_soc)*

L'ensemble des points d'accès aux diverses communications en cours pour un processus donné est accessible à travers un tableau de pointeurs de descripteurs. Chacun de ces descripteurs de Socket (voir la figure **??**) contient les différents paramètres propres à une communication : mode de communication, adresses IP source et destination, Numéro de port source et destination...

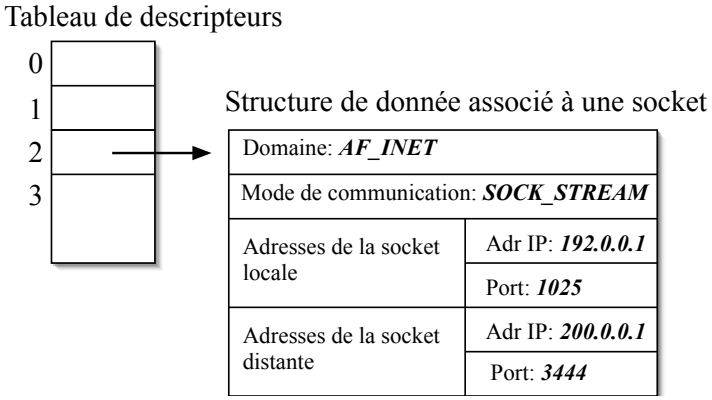

FIGURE 1 – Descripteur de Socket

L'indice dans ce tableau de descripteur est l'identificateur de socket.
Remarque : L'identificateur de socket de valeur nulle correspond au clavier et peut être utilisé de la même manière que toutes les autres sockets.

3.4 Les adresses socket *(struct sockaddr_in)*

Le couple d'adresses (@IP, numéro de port) permet d'identifier la socket au niveau réseau INTERNET. Ce couple sera appelé par la suite adresse socket. Ce couple (adresse IP, numéro de port) doit être rangé dans une structure *sockaddr_in* avant d'être passée en paramètre aux différentes procédures de l'interface (voir la figure **??**).

Remarque : Cette structure est valable pour IPV4, dans le cas de l'utilisation d'adresse IPV6 c'est la structure *sockaddr_in6* qui est utilisée.

Remarque : Vous n'aurez normalement pas à manipuler directement cette structure, une procédure d'instanciation est à votre disposition pour cela (voir ***adr_socket*** plus loin).

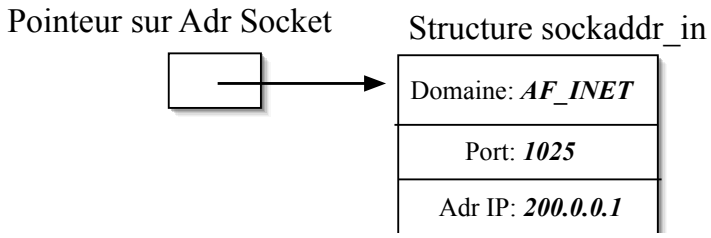

FIGURE 2 – Structure d'adresse d'une socket

4 LES PROCEDURES DE l'INTERFACE

4.1 Renseignement des adresses d'une socket : *adr_socket*

Cette procédure est spécifique à la boîte à outils, il n'y a pas de primitive UNIX équivalente. Elle permet d'affecter les différents champs d'un descripteur d'adresse de socket (**structure sockaddr_in**) avant que celui-ci soit passé à différentes procédures (bind , sendto , connect ...).

void ***adr_socket*** *(service, name, typesock, p_adr_socket)*

*char *service* ; /* nom du service lié au port socket à renseigner */

*char *name* ; /* chaine de caractère correspondant à une adresse IP, nom DNS ou figurant dans le fichier /etc/hosts, soit sous la forme décimal pointé (« 192.0.0.1 » */

int typesock ; /* type de socket (SOCK_STREAM ou SOCK_DGRAM */

*struct sockaddr_in **p_adr_socket* ; /* pointeur sur un pointeur sur la structure d'adresse à renseigner*/

Cette procédure renseigne les adresses de la socket (adresse IP, port) à partir :

- du nom du service : nom dans **/etc/services** ou numéro de port en décimal
- du nom : nom dans **/etc/hosts** ou nom DNS ou adresse IP en décimal pointé
- du type de la socket :SOCK_STREAM ou SOCK_DGRAM
- La structure pointée par **p_adr_socket* est allouée par la procédure *adr_socket*.

La structure **sockaddr_in** se présente comme suit :
struct sockaddr_in

{ *short sin_family ;* /* famille d'adresse : AF_INET*/
ushort sin_port ; /* numéro de port */
ulong sin_addr ; /* adresse de niveau 3 : IP*/
char sin_zero [8] ; /* inutilisé (mis à zéro) */
}

Elle est en fait la forme particulière d'un champs de la structure plus générale **sockaddr** qui est prévue pour différentes familles de protocole et d'adresse.

Remarque : Dans le cas d'un processus client, on peut laisser le choix de numéro de port au système (allocation dynamique) qui gère l'affectation de ces numéros, il faut alors donner la valeur « 0 » au numéro de port.

Remarque 2 : Dans le cas d'un processus serveur, le port du service ne doit pas changer afin d'être toujours connu par les clients potentiels. Il doit donc être défini une fois pour toute par le processus serveur. Dans ce cas la machine doit être accessible sur tous les réseaux auxquels elle est directement reliée (cas de plusieurs interfaces physiques) : la socket est donc définie pour l'ensemble des adresses IP de la machine serveur. Dans ce cas, l'adresse IP est définie de façon particulière. Le paramètre *name* doit être égal à la constante **NULL**. En mode DEBUG l'adresse affichée est alors **0.0.0.0**.

Remarque 3 : Si l'on veut récupérer le nom de la machine sur laquelle s'exécute le programme, on peut utiliser la procédure **gethostname** (voir plus loin sur l'utilisation de cette procédure). **Exemple** : gethostname (myname, lg) ;
adr_socket (" 0 ", myname, SOCK_DGRAM, &p_adr_socket) ;

La figure **??** donne l'affectation des différentes plages de numéros de port. On peut avoir la liste des connexions (adresses Internet et ports) ouvertes sur une machine par la commande **netstat -an**).

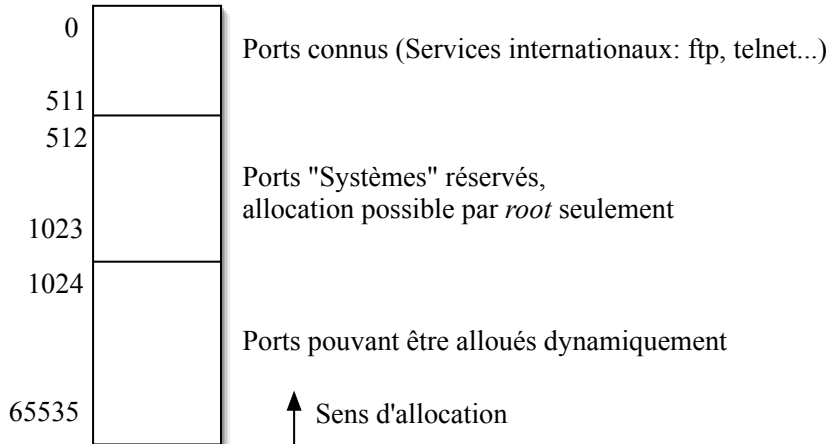

FIGURE 3 – Affectation des numéros de port

4.2 Création d'une socket : *h_socket*

int ***h_socket*** *(domaine, mode) int domaine* ; /* AF_INET */
int mode ; /* SOCK_STREAM ou SOCK_DGRAM */

Cette procédure crée une nouvelle structure de données de socket, et une nouvelle entrée dans la table des descripteurs, qui pointe sur cette structure.
Elle retourne l'entier qui est l'identificateur de la nouvelle socket. Les adresses de la socket locale ne sont pas instanciées. A ce stade, la socket n'est donc pas identifiée au niveau

du réseau. En particulier, les adresses de la socket distante ne sont pas renseignées : la communication n'est donc pas encore définie.

4.3 Association d'une socket à ses adresses : *h_bind*

void ***h_bind*** *(num_soc, p_adr_local)*

int num_soc ; /* Id de socket */

*struct sockaddr_in *p_adr_local* ; /* adresses (locales) de la socket (locale) */

Le **BIND** réalise l'instanciation des adresses **locales** dans le descripteur de socket dont l'identificateur est passé en paramètre.

4.4 Demande de connexion : *h_connect*

void ***h_connect*** *(num_soc, p_adr_distant) int num_soc* ; /* numero de socket */
*struct sockaddr_in *p_adr_distant* ; /* adresses de la socket distante */

Cette procédure est utilisée dans les applications clients qui fonctionnent en mode connecté. Elle réalise la connexion TCP entre la socket d'identificateur *num_sock* d'un processus client et une socket d'un processus serveur dont l'adresse est fournie par *p_adr_distant.*. Il faut donc que le serveur soit en attente de connexion. Elle effectue l'instanciation de l'adresse distante dans la socket locale avec les paramètres fournis.

Remarque : Dans le cas de TCP, l'instanciation de l'adresse locale peut être faite automatiquement avec l'adresse de la machine locale et le bind est donc inutile.

4.5 Mise en état d'écoute : *h_listen*

void ***h_listen****(num_soc, nb_req_att) int num_soc* ; /* numero de socket */
int nb_req_att ; /* nombre maximum de requêtes en attente */

Cette procédure est utilisée en mode connecté pour mettre la socket du serveur en état d'écoute de demandes de connexion des clients, et elle est dite passive. Les demandes de connexion sont mises en file d'attente, et *nb_req_att* est le nombre maximum de requêtes qui peuvent être mises en attente de traitement.
Cette procédure n'est pas bloquante et les réceptions des demandes de connexion se feront en parallèle avec le reste du programme.

Remarque : il peut y avoir des réceptions de données côté serveur alors que la connexion n'a pas encore été acceptée au niveau de l'application (voir procédure **accept** plus loin).

4.6 Acceptation de connexion : *h_accept*

int ***h_accept*** *(num_soc, p_adr_client)*

int num_soc ; /* numero de socket */
*struct sockadrr_in *p_adr_client* ; /* adresses (port, adresse IP) du distant (client) */

Elle est utilisée en mode connecté par le processus serveur pour accepter une demande de connexion qui a été faite par un processus client. Elle retourne le descripteur d'une nouvelle socket qui sera utilisée par le processus serveur pour l'échange des données avec le processus client :

- la socket initiale du processus serveur est utilisée pour recevoir les demandes de connexion des processus clients.
- pour chaque connexion acceptée avec un processus client, une nouvelle socket est créée pour l'échange des données.

S'il n'y a aucune demande de connexion dans la file d'attente, " accept " est bloquant : il attend une demande de connexion.

4.7 Lecture de données en mode connecté : *h_reads*

int ***h_reads*** *(num_soc, tampon, nb_octets)*

int num_soc ; /* numero de socket */
*char *tampon* ; /* pointeur sur les données reçues par le processus */
int nb_octets ; /* nb octets du tampon */

Cette procédure est utilisée en mode connecté par les processus applicatifs pour récupérer les octets transmis par TCP au niveau du port de la socket.
Cette procédure effectue autant d'appels que nécessaire à la primitive UNIX " read " et retourne le nombre d'octets réellement lus. *nb_octets* indique le nombre d'octets que l'on veut recevoir dans la chaîne *tampon*. Si le buffer de réception du port de la socket est vide, le " read " est bloquant : il attend que le buffer soit remplis (par TCP) pour pouvoir réaliser la lecture demandée. La procédure **h_reads** effectuant plusieurs appels à *read* , elle est aussi bloquante tant que le nombre de caractères spécifié n'est pas arrivé.

4.8 Ecriture de données en mode connecté : *h_writes*

int ***h_writes*** *(num_soc, tampon, nb_octets)*

int num_soc ; /* numero de socket */

*char *tampon* ; /* pointeur sur les données émises par le processus */

int nb_octets ; /* nombre d'octets émis = nb octets du tampon */

nb_octets indique le nombre d'octets que l'on veut envoyer, contenus dans la chaîne *tampon* ;

Cette procédure est utilisée en mode connecté par les processus applicatifs pour envoyer des octets à travers une socket donnée. Cette procédure effectue autant d'appels que nécessaire à la primitive UNIX "*write*" et elle retourne le nombre d'octets effectivement écrits.
Si le buffer d'émission (géré par le système) du port de la socket est plein, le " write " est bloquant : il attend que le buffer se vide pour pouvoir réaliser l'écriture demandée. La procédure **h_writes** effectuant plusieurs appels à write peut de la même façon être bloquante.

Remarque : On peut échanger des données en incluant des options avec les primitives " *recv* " et " *send* ". " send " (respectivement " recv ") est identique à " write " (respectivement " read "), dans son fonctionnement, il a seulement un argument supplémentaire d'options, qui permet par exemple d'envoyer une donnée prioritaire.

4.9 Lecture en mode non connecté : *h_recvfrom*

int ***h_recvfrom*** *(num_soc, tampon, nb_octets, p_adr_distant)*

int num_soc ; /*numero de socket */
*char *tampon* ; /* pointeur sur les données reçues par le processus */
int nb_octets ; /* nombre d'octets reçus = nb octets du tampon */
*struct sockaddr_in *p_adr_distant* ; /* pointeur sur adresses socket distante */

Elle est utilisée en mode déconnecté, de la même façon que le " read ". Elle retourne le nombre d'octets effectivement reçus, ou un résultat négatif en cas d'erreur.
Contrairement au read qui fonctionne en mode connecté et donc pour une connexion donnée, cette procédure permet de connaître les adresses de la socket distante, pour lui répondre éventuellement. Pour cela elle instancie le paramètre *p_adr_distant*.

4.10 Ecriture en mode non connecté : *h_sendto*

int ***h_sendto*** *(num_soc, tampon, nb_octets, p_adr_distant) int num_soc* ; /* numero de socket */
*char *tampon* ; /* pointeur sur les données reçues par le processus */
int nb_octets ; /* nombre d'octets à envoyer = nb octets du tampon */
*struct sockaddr_in *p_adr_distant* ; /* pointeur sur adresses socket distante */

Elle est utilisée en mode déconnecté, de la même façon que le " write ". Elle retourne le nombre d'octets effectivement émis, ou un résultat négatif en cas d'erreur.
p_adr_distant indique les adresses de la socket distante vers laquelle les données sont émises : ces adresses socket doivent être renseignées avant le " sendto " . On peut utiliser la procédure **adr_socket** précédemment vue.

4.11 Désallocation d'une socket : *h_close*

void ***h_close****(num_soc)*

int num_soc ; /* numero de socket */

Cette fermeture en cas de mode connecté assure que les messages en attente d'émission seront correctement envoyés. Puis elle désalloue et supprime la srtucture de donnée associée à la socket. Cette procédure n'est pas bloquante, c'est à dire que l'utilisateur retrouve la main toute de suite (avant l'envoi d'éventuels messages en attente d'émission).
Au niveau du réseau une demande de déconnexion est envoyé à la couche transport distante, qui ferme la connexion dans un sens. Pour fermer une connexion TCP dans les deux sens il faut donc deux " close ", un à chaque extrémité de la connexion.
En pratique, une socket peut être partagée par plusieurs processus : dans ce cas, si n processus partagent la socket, il faut n " close " pour supprimer la socket. C'est le nième " close " qui réalise la désallocation de la socket.

4.12 Fermeture d'une socket : *h_shutdown*

void ***h_shutdown****(num_soc, sens)*

int num_soc ; / n° de socket */*
int sens ; / sens dans lequel se fait la fermeture :*

*0 : entrées fermées, 1 : sorties fermées, 2 : entrées et sorties fermées */*

Cette procédure permet de moduler la fermeture complète d'une socket. Elle ferme la socket seulement dans le sens indiqué par *sens*. La fermeture de la socket est brutale, il y a perte immédiate des données en attente d'émission ou de réception dans le buffer concerné.
Elle permet par contre de rendre une socket mono-directionnelle et de l'utiliser comme flot d'entrée ou comme flot de sortie.

5 Algorithmes de programmation CLIENT-SERVEUR

5.1 Modes de communication

Comme nous l'avons vu, il existe deux modes de communication client/serveur :

- le mode connecté : protocole de transport TCP
- le mode non connecté : protocole de transport UDP

Le mode connecté sera utilisé pour des échanges de données fiables (mise à jour d'une base de données par exemple) et pour des échanges de type "flux d'octets" (échanges de gros fichiers par exemple).
Le mode déconnecté est utilisé pour des échanges de messages courts, dont le contenu peut être facilement contrôlé par les processus applicatifs avant d'être traité. Il s'agit

d'échanges de données non sensibles aux pertes, que l'on peut facilement répéter, où les erreurs non détectées ne génèrent pas d'incohérences, ni de catastrophes.

5.2 Types de serveurs

Deux types de serveur pourront être utilisés :

- le serveur itératif : Le processus applicatif traite lui-même toutes les requêtes, les unes après les autres. Ce type de serveur présente de forts risques d'attente par les clients en cas de traitement long. On ne peut pas imaginer par exemple un serveur WEB de ce type.
- le serveur parallèle, ou à accès concurrent : Il sous-traite à des processus fils (lancés en parallèle) pour chacune des requêtes.

5.3 Serveur itératif

Voici les algorithmes de bases écrits à partir des procédures de la boite à outils dans le cas d'un seul client avec un traitement itératif du serveur.

5.3.1 Mode connecté (TCP)

La figure **??** donne le canevas des algorithmes du client et du serveur itératif en mode connecté. Côté client, le *bind* de la socket locale est optionnel, si il n'est pas fait, un numéro de port libre est affecté à la socket locale.

5.3.2 Mode déconnecté (UDP)

La figure **??** donne le canevas des algorithmes du client et du serveur itératif en mode déconnecté. Côté client, le *bind* de la socket locale est aussi optionnel.

5.4 Serveur parallèle

5.4.1 Mode connecté (TCP)

La figure **??** donne le canevas des algorithmes d'un client et d'un serveur parallèle en mode connecté.

5.4.2 Mode déconnecté (UDP)

Les requêtes de différents clients arrivant sur la même socket, la distinction des clients et les traitements possibles peuvent compliquer les algorithmes. L'équivalent du ***Accept*** qui renvoie une nouvelle socket n'existe pas en mode déconnecté. On pourra reproduire

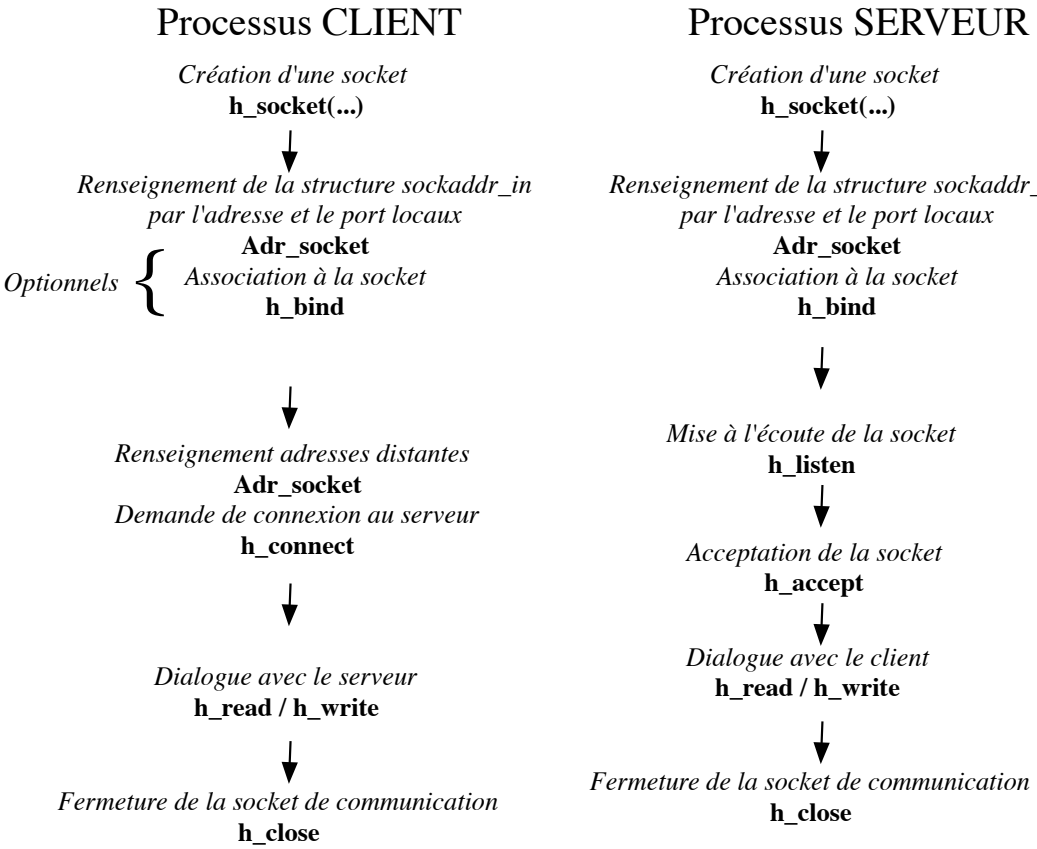

FIGURE 4 – Algorithmes client et serveur en mode connecté (TCP)

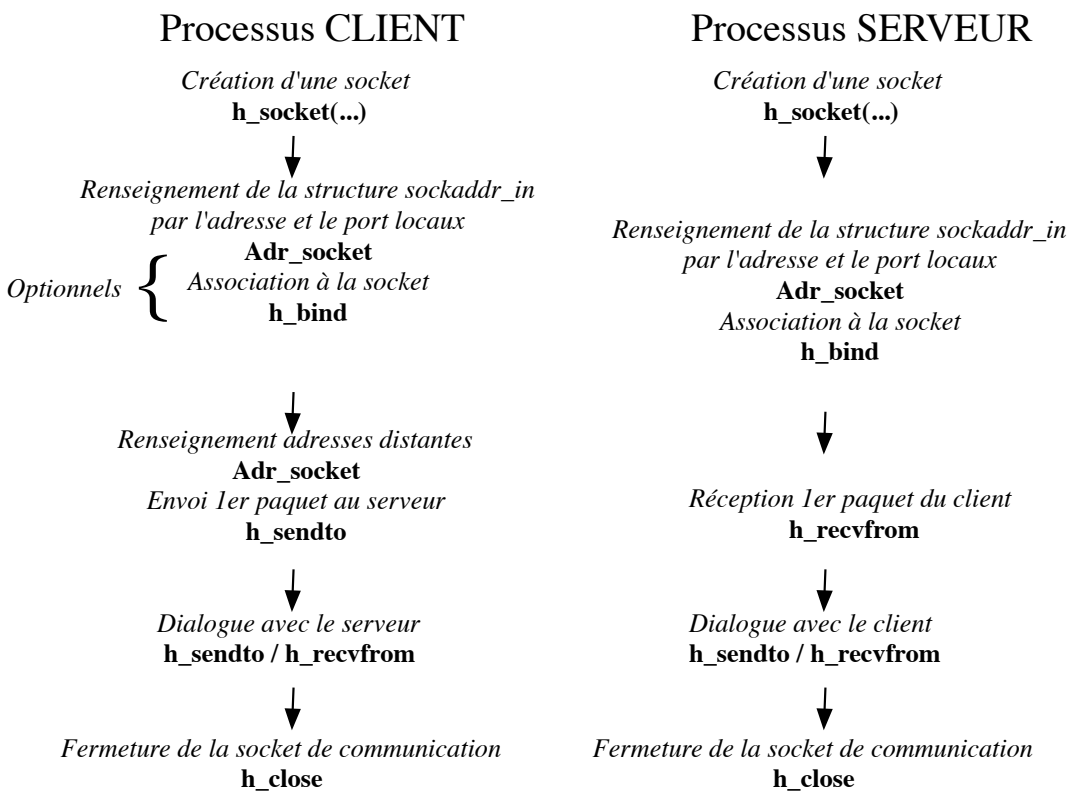

FIGURE 5 – Algorithmes client et serveur en mode déconnecté (UDP)

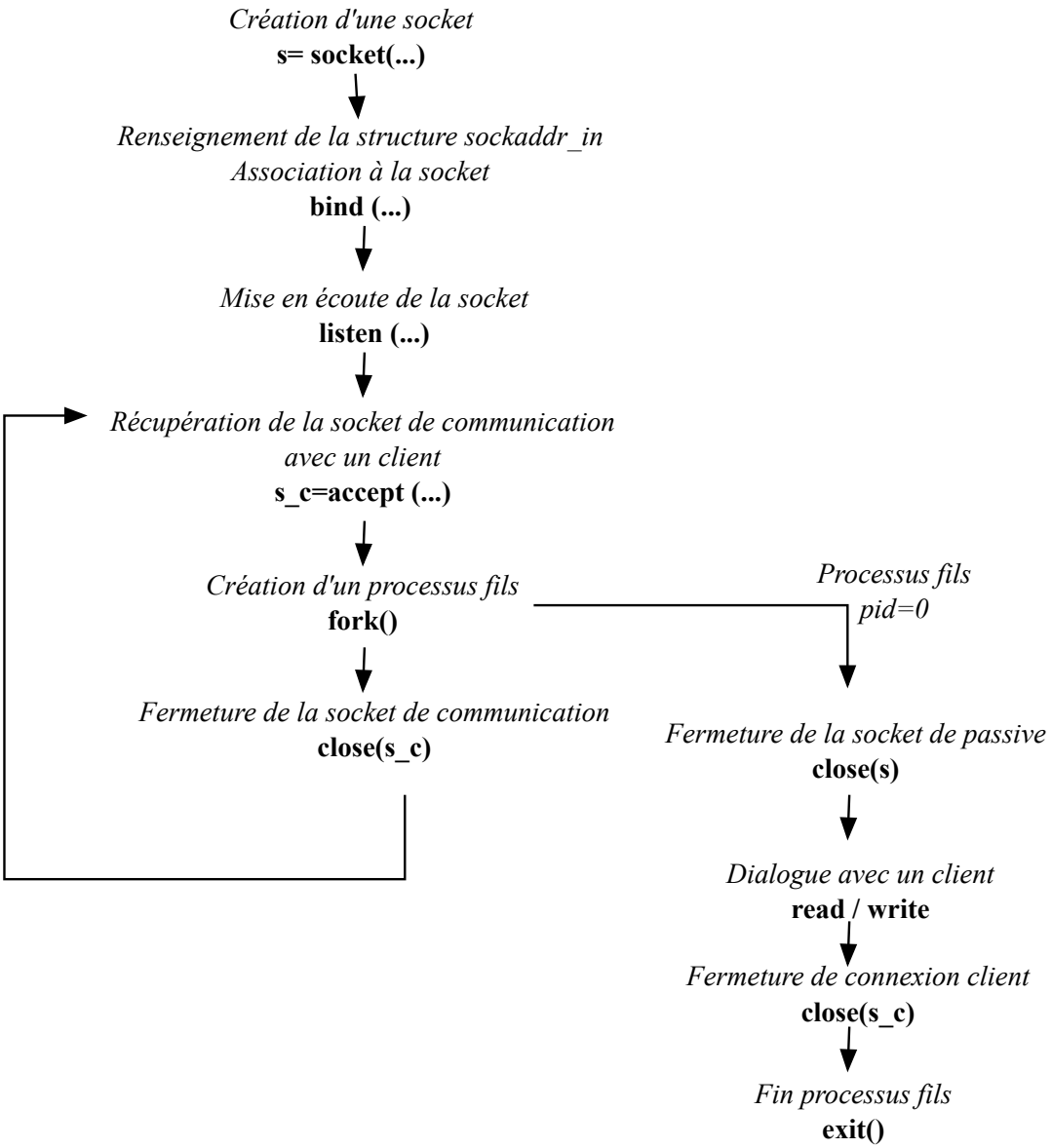

FIGURE 6 – Serveur parallèle en mode connecté (TCP)

ce mécanisme dans le serveur en envoyant dans la première réponse au client un numéro de port d'une nouvelle socket sur laquelle se fera le dialogue.
L'algorithme ressemble alors beaucoup au mode connecté.

5.5 Traitement de plusieurs sockets sans création de processus

Au lieu de lancer des processus fils s'occupant des dialogues avec chaque client, on peut gérer un ensemble de requête sur différentes sockets dans le même processus. Pour cela une primitive particulière (**select**) permet de se mettre en attente sur plusieurs sockets à la fois. La primitive est blocante tant qu'un évenment n'est pas arrivé sur au moins une des sockets de l'ensemble prédéfini de sockets.
Les primitives permettant de gérer un ensemble de sockets (qui sont assimilées à des descripteurs de fichiers) sont :

- **fd_set** : type C définissant un ensemble de descripteurs de fichier (ou de socket)
- **FD_ZERO (fd_set *set)) :**
  permet la remise à vide d'un ensemble de socket.
- **FD_SET (int idsocket, fd_set *set) :**
  ajoute l'identificateur de socket idsocket à l'ensemble set.
- **FD_CLR (int idsocket, fd_set *set) :**
  supprime l'identificateur de socket idsocket de l'ensemble set.
- **FD_ISSET (int idsocket, fd_set *set) :**
  retourne une valeur différente de 0 si l'identificateur de socket idsocket appartient à l'ensemble set.
- **int select (int maxfdpl, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout) :**
  Supprime de l'ensemble de socket ***readfds*** les sockets qui ne sont pas en attente de lecture (dont le buffer de lecture est vide).
- ***maxfdpl*** contient l'identificateur de socket maximum qui sera testé lors du ***select***.
- La fonction C **getdtablesize()** permet d'obtenir ce descripteur maximum.
  Les trois autres paramètres ne nous serviront pas ici et seront forcés à zéro.

Remarque :
l'utilisation de ces fonctions et types nécessite l'inclusion du fichier **<sys/types.h>**

Exemple d'utilisation :

```
main ()
{
fd_set set, setbis ;
int idsock1, idsock2, maxsock ;

FD_ZERO(&set) ;
maxsock= getdtablesize() ;

FD_SET(idsock1, &set) ; /* ajout de idsock1 à l'ensemble set */
FD_SET(idsock2, &set) ; /* ajout de idsock2 à l'ensemble set */
```

```
 bcopy ( (char*) &set, (char*) &setbis, sizeof(setbis)) ;
/* copie de l'ensemble set dans setbis */
select (maxsock,&set, 0, 0, 0) ;
if (FD_ISSET(idsock1, &set)) /* Test si idsock1 appartient à l'ensemble set */

 ...
if (FD_ISSET(idsock2, &set))
...
}
```

6 ANNEXES

6.1 Procédures secondaires relatives aux Sockets

6.1.1 Gestion des adresses IP

int gethostname (char *nom_hote, int longueur_nom)

Elle permet de récupérer le nom (alias de l'adresse internet) de la machine locale. Elle peut être utilisée pour renseigner l'adresse locale dans la structure **sockaddr_in** avant un *bind*.

Exemple :

char myname[MAXHOSTNAMELEN +1] ; /* MAXHOSTNAMELEN est une constante predefinie*/
gethostname(myname, MAXHOSTNAMELEN) ;

int getaddrinfo(const char *node, const char *service, const struct addrinfo *hints, struct addrinfo **res)

La fonction `getaddrinfo()` combine les possibilités offertes par les anciennes fonctions `gethostbyname` et `getservbyname` en une unique interface. Elle permet de trouver l'adresse ou les adresses IP (IPV4 et IPV6) associées à un nom soit DNS, soit donné dans le fichier `/etc/hosts` de la machine.

Elle retourne une liste de structure `addrinfo` dans lesquelles on retrouve un pointeur sur la sructure `sockaddr` qui peut ensuite être utilisée pour les appels aux fonctions `bind, connect .....`
La structure `sockaddr` contient après l'appel une des adresses IP de `node` et le numéro de port donné par `service` (numéro de port ou nom associé dans `/etc/services`). La structure pointée par `hints` peut être remplie avant l'appel pour fixer la valeur de certains champs (par exemple le type de la socket dans `hints->ai_socktype`).

```
struct addrinfo {
    int              ai_flags;
    int              ai_family;
    int              ai_socktype;
    int              ai_protocol;
```

```
    size_t           ai_addrlen;
    struct sockaddr *ai_addr;
    char            *ai_canonname;
    struct addrinfo *ai_next;
};
```

Vous n'aurez pas à vous servir de cette fonction si vous utilisez la fonction `adr_socket` fournie dans `fon.c`.

const char *inet_ntop(int af, const void *src, char *dst, socklen_t size) ;
Permet de convertir une adresse IP issue de la structure *sockadr_in* en chaine de caractère sous forme "décimal pointé".
Exemple :
char str[INET_ADDRSTRLEN] ;
inet_ntop(AF_INET, &(adr_serv.sin_addr), str, INET_ADDRSTRLEN) ;

int inet_pton(int af, const char *src, void *dst) ; Permet de remplir la structure *sockadr_in* par une adresse donnée en "décimal pointé" (chaîne de caractère).
Exemple :
struct sockaddr_in adr_serv ;
inet_pton(AF_INET, "192.0.0.1", &(adr_serv.sin_addr)) ;

6.1.2 Récupération des adresses de la socket locale

int getsockname (int socket, struct sockaddr_in *name, int *namelen)
Elle permet de récupérer les adresses locales de la socket donnée en paramètre en cas d'allocation dynamique (1 <=> échec de la recherche).

ATTENTION : le paramètre **namelen** est un paramètre « donnée/résultat ».

6.1.3 Récupération des adresses de la socket distante

int getpeername (int socket, struct sockaddr_in *name, int *namelen)
Cette procédure permet de récupérer les adresses socket du processus distant connecté à la socket donnée en paramètre.

1 <=> échec de la recherche

6.1.4 Représentation des entiers

ATTENTION ! ! Les représentations des valeurs des entiers sur plusieurs octets diffèrent d'un ordinateur à l'autre (poids forts en premier ou en dernier). TCP/IP a choisi de passer les poids forts en dernier et chaque constructeur livre, avec sa version de système d'exploitation, les procédures de transformation de format dont l'emploi garantit la portabilité des applications.
Il existe quatre procédures qui permettent de passer de la représentation machine à la représentation réseau pour les différents types entiers (voir la figure **??**).

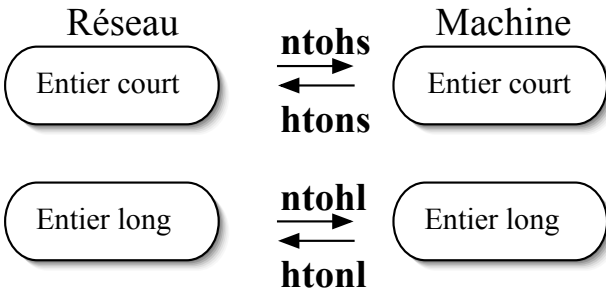

FIGURE 7 – Procédures de normalisation des formats d'entier

Par exemple, "htons " permet de passer de la machine (**h**ost) vers (**to**) le réseau (**n**etware) pour un entier court (**s**hort).

Ainsi par exemple, les deux octets correspondant au paramètre *s_port* de la procédure *getservbyname* sont dans l'ordre "réseau" et il faudra les remettre dans l'ordre "machine" avec la primitive *ntohs*.

6.2 Rappels concernant les créations de processus et leurs synchronisations dans UNIX

Pour créer un processus à partir d'un autre processus, on utilise la primitive ***fork()*** d'UNIX. Le processus créé est appelé " processus fils ", vis à vis du processus initial appelé " processus père ".
Le contexte de ce nouveau processus est créé par duplication du contexte du processus père, c'est à dire qu'il possède le même code exécutable et les mêmes variables (en particulier les descripteurs des sockets) dans une zone mémoire séparée. Son exécution démarre au point même où s'est réalisé le fork(). Le processus père se distingue du processus fils par son identifiant ou PID, dont la valeur est zéro (0) pour le processus fils.
Voici un exemple de code et son exécution dans le temps :

```
int pid; int i, j;

 i=0;
```

```
pid=fork();
  printf(‘‘i = %d’’, i)

  if (pid==0)  /* processus fils */
   { i=1; printf(" i = \%d ’’,i) ; }

  else  /* processus père */
   { i=3 ; printf(‘‘ i = \%d ’’,i) ; }

  j=i ;
 printf (" j = i = \%d ’’,j) ;
```

Les deux processus fonctionnent de façon indépendante. Si l'on souhaite à un moment donné, que le père attende la fin d'exécution du fils, il faudra le synchroniser par la primitive UNIX *wait()*.
Il faut bien avoir en tête qu'il se passe la même chose avec les descripteurs de sockets :

- le père crée un processus fils après " ACCEPT " : il possède à ce moment-là au moins 2 sockets : la socket passive et la socket de connexion au client
- le processus fils hérite donc des 2 sockets du père au moment du fork()

Il faut donc, après le fork() :

- dans le cas du fils : fermer la socket passive, parce qu'il ne s'en sert pas. Il conserve la socket de connexion au client, pour gérer les échanges de données avec celui-ci.
- dans le cas du père, fermer la socket de connexion au client, puisqu'il ne s'en sert pas.

Sans entrer dans les détails, signalons également que pour éviter les processus 'zombies', il faut que le processus serveur (père) ignore le signal SIGCHLD que lui fait le processus fils à sa mort : ce qui s'écrit en début de programme par *signal (SIGCHLD, SIG_IGN) ;*

6.3 Génération d'exécutable

Tous les fichiers de prototypage nécessaires sont inclus dans le fichier **fon.h** , qui contient aussi le prototype de toutes les fonctions décrites ci-dessus. Il faudra donc inclure ce fichier en en-tête des programmes application client et serveur (#include " fon.h "). Le corps des fonctions se trouve dans le fichier **fon.c**. Fichier qu'il est donc nécessaire de compiler et lier avec chacune des applications (voir le **makefile** qui vous est fourni).

Exemple de MAKEFILE :

OBJ1 = fon.o serveur.o
OBJ2 = fon.o client.o
fon.o : fon.h fon.c

gcc -DDEBUG -c fon.c

serveur.o : fon.h serveur.c

gcc -c serveur.c

client.o : fon.h client.c

gcc -c client.c

serveur : {OBJ1}

gcc {OBJ1} -o serveur -lsocket -lnsl

client : {OBJ2}

gcc {OBJ2} -o client -lsocket -lnsl

7 Résumé des procédures de la boîte à outils

Elles sont différenciées des primitives UNIX par un préfixe (**h_** pour highlevel). Notez bien pour les fonctions de lecture et écriture le **s** final qui indique clairement plusieurs appels de la primitive UNIX associée.
Des exemples sont donnés pour chaque primitive.

adr_socket

void adr_socket (char *service, char *name, int typesock, struct sockaddr_in **p_adr_socket) Renseigne les différents champs de la structure d'adresses pointée par ***p_adr_socket**, en fonction des paramètres **service**, **name**, **typesock**.
La structure est allouée par la procédure.

```
char *bip_bip = "2222"
int num_socket ;
struct sockaddr_in *padr_distante ;
adr_socket ( bip_bip, "mandelbrot.e.ujfgrenoble.fr ", SOCK_STREAM , &padr_distante) ;
```

h_socket

int h_socket (int domaine, int mode) Réserve un descripteur socket pour un mode de transmission et une famille de protocoles.

```
int num_socket ;
num_socket = h_socket ( AF_INET, SOCK_STREAM ) ;
```

h_bind

void h_bind (int num_soc, struct sockaddr_in *p_adr_local)
Définit un point d'accès (@IP, numéro de port) local pour la socket.

```
int num_socket ;
struct sockaddr_in adr_locale ;
h_bind (num_socket, &adr_locale ) ;
```

h_connect

void h_connect (int num_soc, struct sockaddr_in *p_adr_distant) Utilisée côté client, en mode connecté, ouvre une connexion entre la socket locale (num_sock) et la socket serveur distante (référencée par la structure sockaddr_in que pointe p_adr_distant).

```
int num_socket ;
struct sockaddr_in adr_serv ;
h_connect (num_socket, &adr_serv ) ;
```

h_listen

void h_listen (int num_soc, int nb_req_att) Utilisée côté serveur, en mode

connecté, place le processus en attente de connexion d'un client. Définit la taille de la file d'attente, c'est-à-dire le nombre maximum de requêtes client qui peuvent être stockées en attente de traitement.

```
int num_socket ;
int nb_requetes = 6 ;
h_listen (num_socket, nb_requetes ) ;
```

h_accept

int h_accept (int num_soc, struct sockaddr_in *p_adr_client) Utilisée côté serveur, en mode connecté. Accepte, sur la socket (qui était en écoute), la connexion d'un client et alloue à cette connexion une nouvelle socket qui supportera tous les échanges. Retourne l'identité du client (sockaddr_in que pointe p_adr_client) qui pourra être traitée par l'application, si aucun traitement n'est envisagé utiliser la constante prédéfinie *TOUT_CLIENT*.

```
int num_socket, sock-client ;
sock-client= h_accept (num_socket, TOUT_CLIENT ) ;
```

h_reads

int h_reads (int num_soc, char *tampon, int nb_octets) Transfert les octets du buffer réception socket vers le tampon du processus applicatif jusqu'à détection d'une fin de transfert ou atteinte de la capacité du buffer. Utilisé en mode connecté, elle est bloquante tant que le buffer de réception socket est vide. Retourne le nombre de caractères effectivement reçus.

```
int taille, num_socket ;
char *message ;
message = ( char * ) calloc ( taille, sizeof (char) ) ;
lg_message = h_reads (num_socket, message, taille ) ;
```

h_writes

int h_writes (int num_soc, char *tampon, int nb_octets) Utilisée en mode connecté. Bloquante tant que le buffer d'émission socket est plein. Transfert le nombre d'octets spécifié du tampon vers le buffer réception socket. Retourne le nombre de caractères effectivement émis.

```
int taille, lg_message, num_socket ;
char *message ;
message = ( char * ) calloc ( taille, sizeof (char) ) ;
h_writes (num_socket, message, lg_message ) ;
```

h_recvfrom

int h_recvfrom (int num_soc, char *tampon, int nb_octets, struct sockaddr_in

p_adr_distant) Utilisée en mode non connecté, bloquante tant que le buffer de réception de la socket est vide. Transfert les octets provenant d'une socket distante (affecte la structure sockaddr_in que pointe p_adr_distant) du buffer réception socket vers le tampon. Retourne le nombre de caractères effectivement reçus.

```
int taille, lg_message, num_socket ;
char *message ;
struct sockaddr_in adr_distante
message = ( char * ) calloc ( taille, sizeof (char) ) ; lg_message = h_recvfrom (num_socket,
message, taille, &adr_distante) ;
```

h_sendto

int h_sendto (int num_soc, char *tampon, int nb_octets, struct sockaddr_in *p_adr_distant) Utilisée en mode non connecté, elle est bloquante tant que le buffer d'émission socket est plein. Transfert le nombre d'octets spécifié du tampon vers le buffer réception socket à destination d'une socket distante spécifique (structure sockaddr_in que pointe p_adr_distant).
Retourne le nombre de caractères effectivement émis.

```
int taille, lg_message, num_socket ;
char *message ;
struct sockaddr_in adr_distante
message = ( char * ) calloc ( taille, sizeof (char) ) ;
h_sendto (num_socket, message, lg_message, &adr_distante) ;
```

h_close

void h_close (int num_soc) Libère le **descripteur socket** après avoir achevé les transactions en cours.

```
int num_socket ;
h_close ( num_socket )
```

h_shutdown

void h_shutdown (int num_soc, int controle) Met fin " violemment " à toute transaction sur une socket, y compris celles en attente, dans le ou les sens spécifiés.

```
#define FIN_RECEPTION 0
#define FIN_EMISSION 1
#define FIN_ECHANGES 2
int num_socket ;
h_shutdown ( num_socket, FIN_RECEPTION ) ;
```