Université de Strasbourg
UFR Mathématiques et Informatique

Licence 3 Informatique
Année 2018/2019

Réseaux et Protocoles

Manuel API socket en langage C

1 Introduction

Le but de ce polycopié est de donner les bases nécessaires pour commencer le TP1 (petits programmes utilisant des connexions entre machines).

Formellement, un(e) socket est un point de communication bidirectionnel par lequel un processus pourra émettre ou recevoir des informations. Moins formellement, au lieu de parler de la communication entre processus, parlons de la communication entre personnes par courrier. Cette communication nécessite que les personnes disposent d'une boîte aux lettres. Une boîte aux lettres est un point de communication ayant une adresse connue du monde extérieur. Les sockets et les processus sont respectivement l'analogue de ces points de communications et des individus. Lorsque deux processus veulent s'échanger des données, il faut que chacun d'eux disposent d'au moins une socket. Les sockets ne sont certes pas le seul moyen de communication entre deux processus mais un grand nombre d'applications sont fondées sur les sockets.

Quand une communication entre deux sockets est établie, on ne se préoccupe pas de savoir comment est réalisée la liaison réelle. Si par exemple la communication se fait grâce à des machines intermédiaire, le programme utilisant les sockets l'ignore complètement. Les protocoles réseaux sont hiérarchisés en couche du plus bas niveau (transmission d'un signal sur un fil conducteur) vers le plus haut niveau (utilisation d'un programme comme un navigateur Internet). Ainsi dans le modèle OSI, on a 7 couches :

7 Application
6 Présentation
5 Session
4 Transport
3 Réseaux
2 Liaison
1 Physique

En pratique, dans les communications basées sur le protocole IP, ce modèle est simplifié :

— Application (Telnet, FTP, Navigateur Internet)
— Transport (TCP, UDP)
— Réseaux (IP)
— Liaison (Ethernet)
— Physique (Câble coaxial, Paires torsadées, fibre)

On situe les sockets au niveau de la couche 4 (Transport). Précisément, les sockets sont les points d'accès aux services de la couche transport. Enfin, en pratique on parle également d'encapsulation des données : les données d'origine sont encapsulées par le protocole applicatif (e.g. ajout de commandes), puis ce nouvel ensemble de données est encapsulé par la couche transport (on parle alors de segment pour TCP ou de datagramme pour UDP). Le segment (ou le datagramme) est lui-même encapsulé par la couche réseaux (on parle alors de paquet) et ce paquet est enfin encapsulé par

la couche liaison (pour Ethernet on parlera alors de trame). Enfin, la trame est envoyée sur le support physique. La figure 1 illustre ces différents niveaux d'encapsulation.

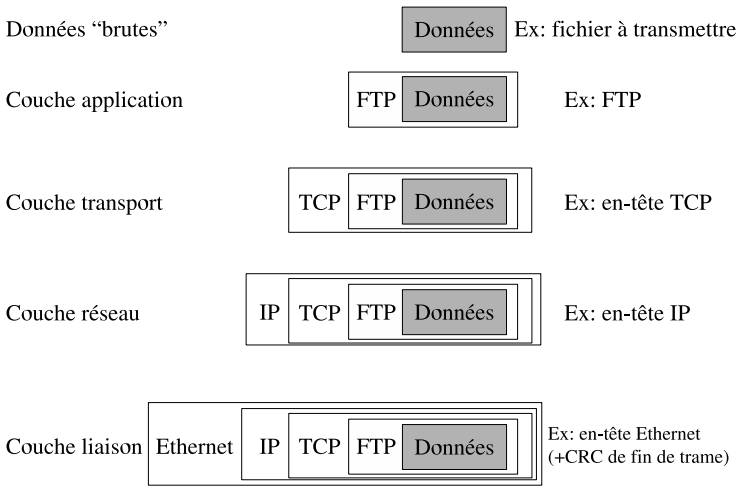

FIGURE 1 – Encapsulation des données

2 La primitive socket

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

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

2.1 Le domaine

Lors de la création d'une socket, il faut indiquer le domaine dans lequel elle se place. Cela définit le système d'adresse utilisé ainsi que le protocole de bas niveau.

— AF_UNIX reservé aux communications locales. Les adresses sont des chemins dans l'arborescence des fichiers.

— AF_INET utilisé par le protocole IPv4 (Internet Protocol version 4, l'actuel protocole Internet). Les adresses des machines sont des entiers sur 32 bits. On les représente souvent en base 256 sous la forme *n1.n2.n3.n4* . Par exemple l'adresse 130.79.7.13 est en fait représentée de manière interne $130*256^3+79*256^2+7*256+13$.

— AF_INET6 protocole IPv6 qui remplacera le protocole IPv4. Les adresses IPv6 sont codées sur 128 bits.

— D'autres tels que AF_APPLETALK, AF_IPX.

2.2 Le type de la socket

Il existe plusieurs types de socket, selon le mode d'utilisation :

— SOCK_DGRAM : la machine va recevoir et envoyer par paquets (appelés *datagrammes*). Ces paquets peuvent être envoyés vers n'importe quelle machine et reçus de n'importe quelle machine (sachant que l'autre machine doit aussi utiliser ce type de transmission). Il n'y a pas de fiabilité et les paquets peuvent être modifiés pendant la transmission (erreurs). Ils peuvent aussi être perdus ou dupliqués et l'ordre n'est pas préservé. Par contre les paquets ne sont pas fragmentés ni regroupés (*analogie avec le courrier*).

— SOCK_STREAM : la socket doit, avant toute transmission, se *connecter* à la machine distante. Les données ne sont *pas* regroupées en trames. Il y a un flot continu de données d'émission et un flot continu de réception. La fiabilité est maximale : ni erreur, ni perte, ni duplication. Par contre comme il n'y a pas de trame, il faut faire attention que la lecture des données à la réception devra peut-être se faire plusieurs fois, même si l'envoi s'est fait en une seule fois. Des messages hors bandes peuvent être envoyés (*analogie avec le téléphone*).

— SOCK_RAW : envoi de paquet pour les protocoles de bas niveau.

— SOCK_RDM : correspond à SOCK_DGRAM mais avec fiabilité (pas d'erreurs ni de pertes). Par contre, l'ordre n'est toujours pas préservé.

— SOCK_SEQPACKET : envoi de paquets mais en mode connecté, avec fiabilité et respect de l'ordre.

Les deux derniers types sont rarement implémentés. Le tableau 1 résume les types de socket et leurs propriétés.

	SOCK_DGRAM	SOCK_STREAM	SOCK_RDM	SOCK_SEQPACKET
Fiabilité	NON	OUI	OUI	OUI
Préservation de l'ordre	NON	OUI	NON	OUI
Non duplication	NON	OUI	OUI	OUI
Mode connecté	NON	OUI	NON	OUI
Préservation des limites des messages	OUI	NON	NON	OUI
Message hors-bande	NON	OUI	NON	NON

TABLE 1 – Types de socket et leurs propriétés

2.3 Le protocole

Ce champ identifie le protocole utilisé pour mettre en oeuvre la transmission. Si on met 0, c'est le protocole par défaut qui est utilisé. Comme la plupart du temps il n'y a qu'un seul protocole correspondant aux deux premiers arguments, c'est souvent ce qui est fait. Si le domaine est AF_INET, alors les deux principaux protocoles sont UDP et TCP :

— UDP : *User Datagram Protocol* pour SOCK_DGRAM
— TCP : *Transmission Control Protocol* pour SOCK_STREAM

Ces deux protocoles utilisent des numéros de port pour permettre plusieurs connexions depuis une machine. Chaque machine ayant une adresse IP a ainsi 65535 ports numérotés de 1 à 65535. En fait, par la carte réseau de la machine on peut ainsi avoir plusieurs connexions *simultanées* (on parle de multiplexage : faire passer plusieurs connexions sur une même ligne).

Certains ports sont réservés à des services particuliers : echo(7), telnet(23), ftp(21), ssh(22), http(80). Ce sont les ports côté serveur. Du côté client, le numéro de port n'est pas fixe. Le système prend en général un port libre supérieur à 1024.

Conclusion :

On utilise principalement deux lancements de socket :

```
socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP);
socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
```

3 Mise en oeuvre concrète en C

3.1 Descripteurs de fichiers et entrée/sortie

De façon générale, la primitive *socket* crée une socket et rend un entier appelé le *descripteur* de la socket. A partir de là, dans le processus qui a fait appel à cette primitive, la socket créée sera toujours désignée et identifiée par ce descripteur. En particulier, pour manipuler une socket, pour fixer son adresse, pour pouvoir émettre un message sur elle ou y lire un message, le processus devra désigner la socket par son desripteur.

Remaque :

Dans un système UNIX, le descripteur d'une socket est de même nature qu'un descripteur de fichier régulier, de terminal ou de tube. Ainsi, ces fichiers spéciaux peuvent souvent interagir avec leur environnement comme s'ils étaient des fichiers réguliers. Dans beaucoup de cas, ces descripteurs sont obtenus par la primitive *open* et, en général, on peut lire et écrire dans un tube, un terminal ou une socket, de la même façon qu'on lit et écrit dans un fichier, c'est-à-dire en utilisant les primitives *read* et *write* avec le descripteur du fichier, du terminal ou du tube.

Exemple :

La connexion sur des machines extérieures sont des cas particuliers d'entrées-sorties. Lire/Ecrire un fichier est aussi un processus d'entrée-sortie. Les fonctions utilisées sont :

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int open(const char *path, int oflag, /* mode_t mode */ ...);
ssize_t read(int desc, void *buf, size_t nbyte);
ssize_t write(int desc, const void *buf, size_t nbyte);
int close(int desc);
```

L'entier retourné par la primitive *open* est appelé *descripteur* du fichier. C'est grâce à cet entier que l'on peut effectuer toutes les opérations sur les fichiers.

Trois descripteurs sont déjà alloués :
- d=0 : entrée standard
- d=1 : sortie standard
- d=2 : sortie d'erreur

L'appel à la primitive *socket* est l'équivalent de *open* pour les connexions.

```
d = socket(AF_INET,SOCK_STREAM,0);

est equivalent de

d = open("fich1",O_RDONLY);
```

Question :

Ecrire un programme prenant en argument deux noms de fichiers et réalisant la copie du premier dans le deuxième.

3.2 Adresses

Les adresses sont gérées par la structure *sockaddr* :

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

struct sockaddr
{
  sa_family_t sa_family; /* address family */
  char sa_data[14];      /* up to 14 bytes of direct address */
}
```

L'ensemble des adresses est divisé en deux domaines : le domaine UNIX qui est un domaine *local* et le domaine IP, qui est le domaine *global*. On utilise une structure particulière d'adresse pour chaque domaine. Ces structures serviront à stocker les informations suffisantes pour attacher une identification à une socket.

3.2.1 Domaine UNIX

```
#include <sys/un.h>

struct sockadd_un
{
  sa_family_t sun_family; /* AF_UNIX */
  char sun_path[108];     /* path name */
};
```

3.2.2 Domaine IP

```
#include <netinet/in.h>

struct in_addr { uint32_t s_addr; };

typedef uint16_t in_port_t;

struct sockaddr_in
{
  sa_family_t sin_family; /* AF_INET */
  in_port_t sin_port;
  struct in_addr sin_addr;
  unsigned char sin_zero[8];
};
```

3.2.3 Autres structures et fonctions liées à l'adressage

La fonction *gethostbyname* permet de récupérer l'adresse IP d'une machine en fonction de son nom, dans une structure *hostent* :

```
#include <netdb.h>
```

```
struct hostent
{
  char *h_name;               /* official name of host */
  char **h_aliases;           /* alias list */
  int h_addrtype;             /* host address type */
  int h_length;               /* length of address */
  char **h_addr_list;         /* list of addresses from name server */
#define h_addr h_addr_list[0] /* address, for backward compability */
};

struct hostent *gethostbyname(const char *name);
```

Exemple d'utilisation :

```
struct hostent *he;
struct sockaddr_in addr;

addr.sin_family = AF_INET;
he = gethostbyname("ada.u-strasbg.fr");
if(he == NULL)
{
  /* traitement de l'erreur */
  perror("gethostbyname");
  return -1;
}
memcpy(&addr.sin_addr,he->h_addr,sizeof(addr.sin_addr));
addr.sin_port = htons(port);
```

Juste après l'appel de *socket*, on peut lui attacher une adresse. C'est une adresse de la machine locale.

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

int bind(int s, const struct sockaddr *name, int namelen);
```

Si on veut spécifier toutes les adresses de la machine (cas d'une machine se situant sur plusieurs réseaux ou simplement si on ne connaît pas l'adresse de la machine) on peut utiliser INADDR_ANY (faire addr.sin_addr.s_addr = htonl(INADDR_ANY)). Si on n'utilise pas *bind*, alors à la première lecture/écriture, le système attribue l'adresse INADDR_ANY et un port libre (en général entre 1024 et 5000) à la socket.

3.3 Portabilité

Il existe deux façon de coder les entiers sur une suite d'octets. Par exemple, pour coder 1010200108=0x3c366e2c, on peut le coder par la suite d'octets [0x3c,0x36,0x6e,0x2c] (Big-Endian) ou [0x2c,0x6e,0x36,0x3c] (little-endian). Les (anciens) Macintoshs et les stations SUN sont Big-Endian alors que les PCs et ALPHA sont little-endian. Sur le réseau, les données sont Big-Endian. Donc pour être portable, on utilise les 4 fonctions :

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

uint32_t htonl(uint32_t hostlong);  /* host to network long  */
uint16_t htons(uint16_t hostshort); /* host to network short */
```

```
uint32_t ntohl(uint32_t netlong);   /* network to host long  */
uint16_t ntohs(uint16_t netshort);  /* network to host short */
```

Remarque :

La structure *hostent* retournée par la primitive *gethostbyname* est directement en format Big-Endian.

3.4 Transfert UDP

On utilise directement des envois et réceptions de datagrammes en spécifiant à chaque fois l'adresse destination.

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

ssize_t sendto(int s, const void *msg, size_t len, int flags,
       const struct sockaddr *to, int tolen);

ssize_t recvfrom(int s, void *buf, size_t len, int flags,
 struct sockaddr *from, int *fromlen);
```

Remarque :

On pourrait aussi utiliser la primitive *connect* et juste *send* et *recv* (on parle alors de pseudo connexion, voir page du manuel).

3.5 Connexions TCP

Ce mode de communication doit passer par une connexion entre le client et le serveur. Ensuite, lors d'envoi/réception de segments dans la socket, il n'est plus utile de spécifier l'adresse destination/source.

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

int listen(int s, int backlog);
int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
int connect(int s, const struct sockaddr *name, int namelen);
ssize_t read(int desc, void *buf, size_t nbyte);
ssize_t write(int desc, const void *buf, size_t nbyte);
ssize_t send(int s, const void *msg, size_t len, int flags);
ssize_t recv(int s, void *buf, size_t len, int flags);
```

L'argument *backlog* est le nombre de connexions en attente, au maximum SO_MAXCONN (5 en général, 128 sous linux ...).

On peut envoyer des messages HORS-BANDES qui sont prioritaires grâce aux commandes :

```
send(d, message, tmessage,  MSG_OOB);
recv(d, message, &tmessage, MSG_OOB);
```

La commande *netstat* permet de voir toutes les connexions TCP ouvertes (voir annexe).

4 Bloquant vs Non Bloquant

Normalement les primitives *read* ou *recvfrom* attendent qu'il y ait une donnée à lire pour finir leur exécution. Ceci peut être gênant si on veut par exemple aussi lire sur un autre descripteur. Supposons que l'on ait deux descripteurs, on peut par exemple avoir le code suivant :

```
while(1)
{
  nbuf = read(d1,buf1,NBUF);

  if(nbuf > 0)
  {
    /* traitement */
  }

  nbuf = read(d2,buf1,NBUF);

  if(nbuf > 0)
  {
    /* traitement */
  }
}
```

Question :

Quel est le problème ?

4.1 La primitive select

Elle permet d'attrendre sur plusieurs descripteurs en même temps.

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

struct timeval
{
  time_t tv_sec;        /* seconds */
  suseconds_t tv_usec; /* and microseconds */
};

int select(int nfds, fd_set *readfds, fd_set *writefds,
   fd_set *errorfds, struct timeval *timeout);

void FD_SET(int fd, fd_set *fdset);
void FD_CLR(int fd, fd_set *fdset);
void FD_ISSET(int fd, fd_set *fdset);
void FD_ZERO(fd_set *fdset);
```

Le programme précédent devient :

```
fd_set readfds;
fd_set readfds2;
int nfds;
```

```
FD_ZERO(&readfds);
FD_SET(d1,&readfds);
FD_SET(d2,&readfds);


if(d1 > d2)
{
  nfds = d1 + 1;
}
else
{
  nfds = d2 + 1;
}

while(1)
{

  readfds2 = readfds;
  nr = select(nfds, &readfds2, 0, 0, 0);
  if(nr == -1)
  {
    /* traitement de l'erreur */
    perror("select");
  }
  else
  {
    if(FD_ISSET(d1,&readfds2))
    {
      nbuf = read(d1,buf,NBUF);
      /* traitement */
    }
    if(FD_ISSET(d2,&readfds2))
    {
      nbuf = read(d2,buf,NBUF);
      /* traitement */
    }
  }
}
```

4.2 Création d'un processus fils

Une autre méthode permettant d'attendre sur deux appels bloquants est de créer un processus fils (primitive *fork*). Ainsi, chaque processus (le père et le fils) pourront chacun attendre en parallèle sur un appel bloquant. Avec cette méthode, le programme précédent devient :

```
int pid;
int errno;
int status;

/* Crée un processus fils. */
pid = fork();
if(pid == -1)
```

```
{
  /* traitement de l'erreur */
  perror("fork");
  return -1;
}

while(1)
{
  if(pid == 0)
  {
    /* processus fils */
    nbuf = read(d1, buf, NBUF);
    /* traitement */
  }
  else
  {
    /* processus pere */
    nbuf = read(d2, buf, NBUF);
    /* traitement */
  }
}

...

/* Traitement de fin du processus père. */
/* Attend la terminaison du processus fils. */
errno = wait(&status);
if(errno == -1)
{
  perror("wait");
  return -1;
}
return status;
```

5 Annexes

5.1 Les pages de manuel

Les fonctions utilisées en réseaux sont dans les sections 2. Par exemple pour avoir le manuel de la primitive *send*, il faut utiliser *man 2 send*. Les fonctions de conversions se trouve dans la section 3 du manuel.

5.2 Le commande strace

Si on veut afficher les appels systèmes lancés par un programme, il faut l'exécuter en le faisant précéder de la commande *strace*.

5.3 La commande nstat

La commande *netstat* affiche toutes les connexions établies sur une machine :

```
[daurat@arthur2 daurat]$ netstat -a --inet
Connexions Internet actives (serveurs et etablies)

Proto Recv-Q Send-Q  Adresse locale   Adresse distante         Etat
tcp        0      0  arthur2:2123     blinis:3917
ESTABLISHED
tcp        0      0  *:2123           *:*                     LISTEN
tcp        1      0  arthur2:3549     lsiit.u-strasbg.fr:www
CLOSE_WAIT
tcp        0      0  arthur2:1023     dpt-info.u-strasbg.:ssh
ESTABLISHED
tcp        0      0  *:6000           *:*                     LISTEN
tcp        0      0  *:printer        *:*                     LISTEN
tcp        0      0  *:ssh            *:*                     LISTEN
udp        0      0  localhost:1071   localhost:2317
ESTABLISHED
udp        0      0  *:2317           *:*
udp        0      0  *:2000           *:*
```

5.4 La commande valgrind

Valgrind est un utilitaire proposant plusieurs outils permettant de profiler et de valider un programme. Parmi ces outils, le vérificateur de mémoire permet de contrôler la gestion de la mémoire d'un programme. Cet outil affiche notamment :

- Les accès en lecture et en écriture des zones mémoires qui ne sont pas allouées.
- Les accès en lecture à des zones mémoires qui ne sont pas initialisées.
- Les doubles désallocations de mémoire pour la même zone mémoire.
- Les zones de mémoire qui ne sont pas désallouées à la terminaison du programme (fuites de mémoires).

Pour valider la mémoire de votre programme, vous devez le compiler avec l'option *-g* avant de l'exécuter en le faisant précéder de la commande : *valgrind –tool=memcheck ./mon_programme mes_arguments*.

De plus, vous pouvez également lier *gdb* à la validation de *valgrind* avec la commande : *valgrind –tool=memcheck –db-attach=yes ./mon_programme mes_arguments*. Ainsi à chaque erreur, *valgrind* vous proposera de lancer *gdb* pour approfondir l'analyse de l'erreur.

```
==15640== Conditional jump or move depends on uninitialised value(s)
==15640==    at 0x402A0D0: pvh_dct_rle_compressed (pvh_dct.c:344)
==15640==    by 0x402D71D: pvh_subbanded_block_map_encode
(pvh_subbanded_block_map.c:235)
==15640==    by 0x4029A90: pvh_encode_next_frame (libpvh.c:65)
==15640==    by 0x8049663: video_decode (avcodec_sample_opengl.c:174)
==15640==    by 0x4228E46: glutMainLoopEvent (in /usr/lib/libglut.so.3.8.0)
==15640==    by 0x4229285: glutMainLoop (in /usr/lib/libglut.so.3.8.0)
==15640==    by 0x8049CE5: main (avcodec_sample_opengl.c:431)
==15640==
==15640== ---- Attach to debugger ? --- [Return/N/n/Y/y/C/c] ----
```

5.5 La commande gdb

Gdb est également un validateur de logiciel, permettant d'obtenir de plus amples informations sur l'état actuel du programme. Après l'avoir compilé avec l'option *-g*, démarrez la validation avec *gdb ./mon_programme*, faisant apparaître l'invite de commande de *gdb*. Dès lors, vous pouvez lancer votre programme avec la commande : *run mes_arguments*. Ainsi à chaque erreur, vous pouvez interroger *gdb* sur l'état de votre programme :

- La commande *where* affiche la pile d'appels de fonctions.
- La commande *info locals* affiche la valeur actuelle des variables de la fonction.

```
0x0402a0d0 in pvh_dct_rle_compressed (src=0x565ef48 "\v", dst=0x579a028 "v|",
    dst_length=589824, bit=0xbea214e5 "\001") at pvh_dct.c:344
344             if(length != tmp_length){
(gdb) where
#0  0x0402a0d0 in pvh_dct_rle_compressed (src=0x565ef48 "\v", dst=0x579a028
    "v|", dst_length=589824, bit=0xbea214e5 "\001") at pvh_dct.c:344
#1  0x0402d71e in pvh_subbanded_block_map_encode
    (pvh_subbanded_block_map=0x804a374, actual_block_map=0x804a364,
    old_block_map=0x804a36c, row_offset=0x804a360 "\003", buffer=0x579a028 "v|",
    buffer_len=589824, dct_block=0x565ef48 "\v",
    list_circ_selected_blocks=0x804a380) at pvh_subbanded_block_map.c:235
#2  0x04029a91 in pvh_encode_next_frame (pvh=0x804a360, buffer=0x579a028 "v|",
    buffer_len=589824, buffer_rgb=0x4cd7d88 '\b' <repeats 24 times>, '\a'
    <repeats 96 times>,
    "\006\006\006\005\005\005\005\005\005\006\006\006\b\b\b\t\t\t\t\t\t",
    '\a' <repeats 35 times>...) at libpvh.c:65
#3  0x08049664 in video_decode (dumm=0) at avcodec_sample_opengl.c:174
#4  0x04228e47 in glutMainLoopEvent () from /usr/lib/libglut.so.3
#5  0x04229286 in glutMainLoop () from /usr/lib/libglut.so.3
#6  0x08049ce6 in main (argc=2, argv=0xbea21834) at avcodec_sample_opengl.c:431
(gdb) info locals
i = 64 '@'
j = 2 '\002'
bit_j = 0 '\0'
k = 2 '\002'
bit_k = 7 '\a'
length = 63 '?'
mode = 0 '\0'
value = 0 '\0'
tmp = 62 '>'
(gdb) quit
```