FIFO (PIPES) & SOCKETS

Guillaume Chanel

Remerciements à Jean-Luc Falcone

Septembre 2019

PIPES ET FIFO

RAPPEL SUR LES PIPES ET FIFO

Rappel:

```
$ ls -lh /dev | more
```

- Le noyau crée un canal de communication anonyme (**pipe** en anglais, **tube** en français) entre les processus `ls` et `more`
- Il est également possible de créer des tubes nommées (**named pipes** ou **FIFO**).

TUBES ANONYMES

On peut créer un canal de communication anonyme en utilisant:

```
int pipe(int fildes[2]);
```

- `filedes[0]` est un descripteur de fichier représentant la sortie du tube/pipe (i.e. on peu lire sur ce descripteur);
- `filedes[1]` est un descripteur de fichier représentant l'entrée du tube/pipe (i.e. on peu écrire sur ce descripteur);
- retourne -1 en cas d'erreur (voir `errno`);
- Pas d'accès aléatoire possible.

CONCEPTS GENERAUX DES PIPES

Les tubes et FIFO:

- Permettent une communication à haute vitesse entre deux processus sur la même machine.
- Ont deux extrémités: une ouverte en lecture et une ouverte en écriture.
- **Sont unidirectionels**: en conséquence du point précédent l'information ne transite que dans un sens.

- Sont **bloquants**:
  - L'ouverture d'une extrémité bloque jusqu'à l'ouverture de l'autre extrémité.
  - Permet d'établir des **Rendez-Vous**
  - Possibilité de **deadlocks** !

PIPE NOMMÉ: COMMANDE `mkfifo(1)`

On peut créer un FIFO avec la commande:

```
mkfifo [OPTION]... NOM...
```

- Où `NOM` est le nom du FIFO à créer
- Parmi les options on peut passer les permissions du FIFO par l'option `-m MODE`.

Exemple shell

```
$ mkfifo -m 0640 /tmp/fifo1
$ ls -lh /dev > /tmp/fifo1

$ more /tmp/fifo1  # Dans un autre shell
```

FONCTION POSIX `mkfifo(2)`

On peut créer un FIFO avec l'appel système:

```
int mkfifo(const char *pathname, mode_t mode);
```

- `pathname` est le nom du fichier à créer
- `mode` représente les permissions (modifiées `mode & ~umask`)
- Un FIFO peut être ouvert en lecture/écriture comme n'importe quel fichier (`open`/`read`) mais il faut veiller à respecter la directionalité du fifo
- Pas d'accès aléatoire possible.

EXEMPLE - PRODUCER

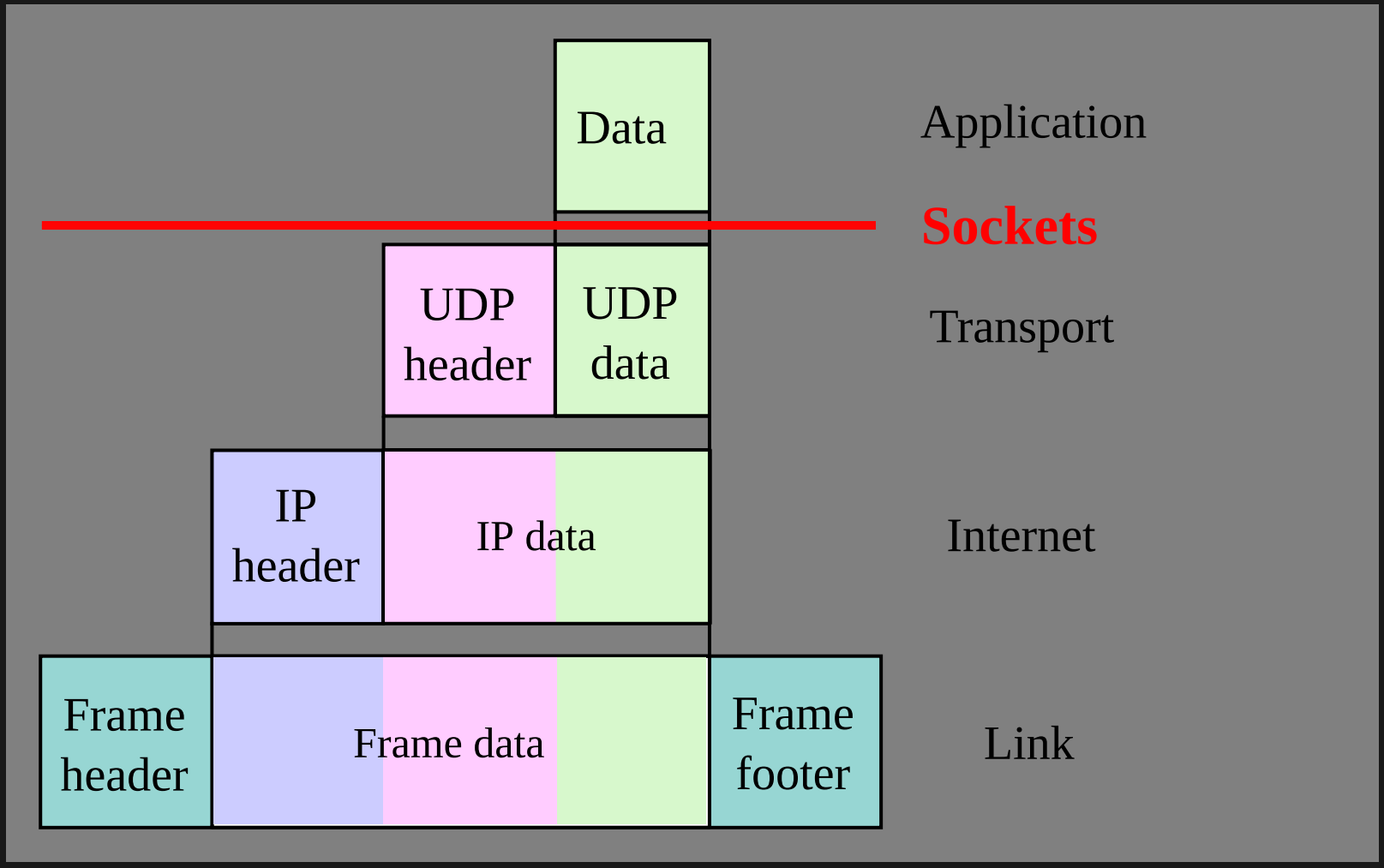
```
Include example there (see script)
```

EXEMPLE - CONSUMER

```
Include example there (see script)
```

SOCKETS

ARCHITECTURE RÉSEAU



SOCKETS

- Abstraction d'un canal de communication à travers le réseau
- Descripteur de fichier:
  - Ecriture avec `read()`
  - Lecture avec `write()`
  - Pas d'accès aléatoire possible
- Modèle client-serveur

DOMAINE D'ADRESSAGE

- Un socket est peut-être lié à une adresse.
- Il existe deux domaines d'adressages:

internet domain	communication réseau (`AF_INET`)
unix domain	communication locale (`AF_UNIX`)

TYPES DE SOCKETS

Il existe plusieur types de sockets dont:

par flot	avec connection, par exemple TCP (SOCK_STREAM)
par datagramme	sans connection, par exemple UPD (SOCK_DGRAM)
brut	sans protocol de transport (SOCK_RAW)

Nous couvrirons surtout le premier type.

ADRESSES INTERNET

ADRESSAGE INTERNET

L'adresse est composée d'une adresse IP et d'un numéro de port (16bits).

Structures IPv4 (man 7 ip)

```
struct sockaddr_in {
    sa_family_t     sin_family; /* famille: AF_INET */
    in_port_t       sin_port;   /*port: big-endian */
    struct in_addr  sin_addr;   /* adresse internet */
};

struct in_addr {
    uint32_t        s_addr;     /*adresse ip: big-endian
}
```

OBTENIR UNE ADRESSE IP (`inet_pton`)

La fonction suivante permet d'obtenir une adresse IP valide:

```
int inet_pton(int af, const char *src, void *dst);
```

`af` Famille d'adresse soit `AF_INET`, soit `AF_INET6`

`src` la représentation de l'adresse (par exemple `192.168.1.1`).

`dst` un pointeur vers une structure `in_addr` ou `in6_addr` à initialiser.

Retourne:

`1` succès

`0` adresse non-valide

`-1` famille non-valide

OBTENIR LA REPRÉSENTATION D'UNE ADRESSE IP (`inet_ntop`)

La fonction suivante permet d'obtenir la représentation d'une adresse IP:

```
const char *inet_ntop(int af, const void *src, char *dst, socklen_t size);
```

`af`	Famille d'adresse soit `AF_INET`, soit `AF_INET6`
`src`	un pointeur vers une structure `in_addr` ou `in6_addr` initialisée.
`dst`	un pointeur vers un buffer (pour obtenir la représentation).
`size`	la taille du buffer

Retourne `NULL` en cas d'erreur (cf `errno`)

OBTENIR UN NUMÉRO DE PORT VALIDE (`htons`)

On peut convertir un entier, en un numéro de port valide grâce à:

```
uint16_t htons(uint16_t hostshort);
```

Le résultat est dans le bon byte-order (*Big-Endian*).

CLIENTS

CLIENT TCP

Le fonctionement d'un client TCP est le suivant:

1. Crée un socket (`socket`)
2. Connecte un socket à un serveur (`connect`)
3. Lecture/Ecriture à parir du socket (`read/write`)
4. Ferme le socket (`close`)

CRÉER UN SOCKET (`socket`)

On utilise pour créer un socket, l'appel système:

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

domain famille d'adresse (`AF_INET`[1], `AF_INET6`[1], `AF_UNIX`, ...)

type type de communication (`SOCK_STREAM`, `SOCK_DGRAM`, `SOCK_RAW`, ...)

protocol protocol de transport, passer `0` pour UDP et TCP

Retourne, soit un descripteur de fichier, soit -1 (cf. `errno`).

1. ici PF_INET devrais être utilisé mais AF_INET est toléré et souvent utilisé à la place (voir le man)

INITIER UNE CONNECTION (`connect`)

L'appel système suivant permet d'initier une connection:

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

sockfd	descripteur de fichier du socket
addr	un pointeur vers une adresse
addrlen	la longueur de la structure

Retourne 0 en cas de succès, et -1 sinon (cf. `errno`).

EXEMPLE DE CLIENT TCP

```
struct sockaddr_in address;
memset( &address, 0, sizeof(address) );
inet_pton( AF_INET, "192.168.1.1", &(address.sin_addr) );
address.sin_family = AF_INET;
address.sin_port = htons(8080);

int sock = socket(AF_INET, SOCK_STREAM, 0);
connect(sock, (struct sockaddr *) &address, sizeof(address));
...
read(sock, ...)
write(sock, ...)
```

SERVEURS

SERVEUR TCP

Le fonctionement d'un serveur TCP est le suivant:

1. Crée un socket serveur (`socket`)
2. Attache le socket serveur à une adresse (`bind`)
3. Ecoute le début d'une connection (`listen`)
4. Accepte une connection et obtient un socket client (`accept`)
5. Lecture/Ecriture à parir du socket client (`read/write`)
6. Ferme le socket client (`close`)
7. Répète l'étape 4 si nécéssaire
8. Ferme le socket serveur (`close`)

CRÉER UN SOCKET (`socket`)

On utilise l'appel système `socket` pour créer un socket serveur (cf. *client*).

LIER UN SOCKET À UNE ADRESSE (`bind`)

On lie un socket à une adresse (interface locale) avec l'appel système:

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

sockfd	le descripteur du socket
addr	un pointeur vers l'adresse à lier.
addrlen	la taille de la structure d'adresse.

Retourne 0 en cas de succès, -1 sinon (cf. `errno`)

ECOUTER LES CONNECTION (`listen`)

L'appel système suivant, permet de marquer un socket comme étant **passif**, c'est à dire un socket permettant d'accepter des connections:

```c
int listen(int sockfd, int backlog);
```

sockfd le descripteur du socket

backlog taille maximum de la queue de connections en attente.

Retourne 0 en cas de succès, -1 sinon (cf. `errno`)

ACCEPTER LES CONNECTIONS (`accept`)

L'appel système suivant permet d'accepter une connection:

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

`sockfd`	Descripteur du socket (doit être passif).
`addr`	Structure garnie avec les informations du client.
`addrlen`	Longueur de la structure.

- Retourne un nouveau descripteur de fichier permettant de communiquer avec le client en cas de succès et 0 sinon (cf. `errno`)
- Bloque jusqu'à la prochaine connexion entrante ou en extrait une de la queue.

EXEMPLE DE SERVEUR TCP (1)

```
struct sockaddr_in address;
memset(&address, 0, sizeof(address));
address.sin_family = AF_INET;
address.sin_addr.s_addr = htonl(INADDR_ANY);
address.sin_port = htons(8080);

int serverSock = socket(AF_INET, SOCK_STREAM, 0);
bind( serverSock, (struct sockaddr *) &address, sizeof(address) );
listen(sock, 5);

while( 1 ) {
    struct sockaddr_in clientAddress;
    unsigned int clientLength = sizeof(clientAddress);
    int clientSock = accept(serverSock,
                            (struct sockaddr *) &clientAddress,
                            &clientLength);

    /* Lectures/Ecritures sure clientSock (read/write) */

    close( clientSock );
}
```

REMARQUE

- L'utilisation de `INADDR_ANY` permet de se lier à **toutes** les interfaces réseaux de la machine.
- On utilise la fonction `htonl` pour obtenir une adresse numérique valide:

```
address.sin_addr.s_addr = htonl(INADDR_ANY)
```

EXEMPLE: A FILE SERVER

- Common code: file_transmission.c and file_transmission.h
- Sever code: file_server.c
- Client code: file_client.c

PROBLÈMES

Dans l'exemple précédent, il faudrait:

- Avertir le client qui demande un fichier qui n'existe pas.
- Permettre au client d'obtenir la liste des fichiers disponibles.

- **Nécéssité de définir un protocole**

Attention

L'exemple précédent contient une faille de sécurité importante.

QUASI-UNIVERSELLES

- Les sockets sont implémentés au niveau de l'OS, par presque tous les OS.
- La grande majorité des languages de programation ont une librairie permettant d'utiliser les sockets.

- Les sockets permettent la communication entre processus écrits dans des langages différents.
- Les sockets permettent la communication entre OS différents.

SOCKETS EN PYTHON

```
#CLIENT
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("www.mcmillan-inc.com", 80))

#SERVEUR
serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serversocket.bind((socket.gethostname(), 80))
serversocket.listen(5)
```

SOCKETS EN JAVA

```
try {
    serverSocket = new ServerSocket(4444);
}
catch (IOException e) {
    System.out.println("Could not listen on port: 4444");
    System.exit(-1);
}
Socket clientSocket = null;
try {
    clientSocket = serverSocket.accept();
}
catch (IOException e) {
    System.out.println("Accept failed: 4444");
    System.exit(-1);
}
```

SOCKETS EN SCHEME

```
(define (id-server)
   (let ((socket (open-socket)))
    (display "Waiting on port ")
    (display (socket-port-number socket))
    (newline)
    (let loop ((next-id 0))
      (call-with-values
        (lambda ()
          (socket-accept socket))
         (lambda (in out)
          (display next-id out)
          (close-input-port in)
          (close-output-port out)
          (loop (+ next-id 1)))))))
```

AUTRES FONCTIONS UTILES

SEND/RECEIVE

On peut utiliser les appels systèmes suivant à la place de `read` et de `write` lorsqu'on utilise des sockets:

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

- Le paramètre `flags` permet de passer des paramètres supplémentaires pour contrôler finemement la transmission.
- `recv(sockfd, buf, len, 0)` est équivalent à `read(sockfd, buf, len)`
- `send(sockfd, buf, len, 0)` est équivalent à `write(sockfd, buf, len)`

SENDFILE (NON POSIX)

Sous Linux, on peut remplacer une paire `read/write` ou `send/recv`, par un appel à `sendfile` qui permet de rester dans l'espace du noyau:

```
ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count)
```

- `out_fd` est le descripteur ouvert en *écriture* (devait être un socket jusqu'à Linux 2.6.33)
- `in_fd` est le descripteur ouvert en *lecture* (ne peut pas être un socket)
- `offset` représente l'offset en lecture (peut être `NULL`)
- `count` taille à envoyer.

UDP

Différences avec TCP:

Client pas besoin d'établir une connection.

Serveur pas besoin d'écouter et d'accepter une connection.

Lectures/Ecritures

Comme on n'établit pas de connection, on utilisera les appels systèmes suivants:

`sendto` pour envoyer des données vers une adresse.

`recvfrom` pour recevoir des données depuis une adresse.

UNIX SOCKET

- Un socket Unix permet d'établir une communication locale entre deux processus au moyen d'un **inode**.
- Il faut passer le domaine `AF_UNIX` à l'appel système `socket`.
- L'adressage est différent, mais tout le reste est identique aux sockets internet.

Adresse Unix (man 7 unix)

```c
#define UNIX_PATH_MAX    108
struct sockaddr_un {
    sa_family_t sun_family;         /* Famille: AF_UNIX*/
    char sun_path[UNIX_PATH_MAX];  /* Chemin sur le systeme de fichiers */
};
```

RÉSOLUTION DES NOMS DE DOMAINE (`gethostbyname`)

Obsolète

Cette fonction est obsolète !!!

La fonction suivante permet de résoudre un nom de domaine:

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

Structure hostent

```
struct hostent {
    char  *h_name;      /* nom officiel */
    char **h_aliases;   /* tableau d'alias */
    int    h_addrtype; /* type: AF_INET ou AF_INET6 */
    int    h_length;   /* longueur de l'adresse */
    char **h_addr_list;/* tableau d'adresses */
};
```