Processus, suite : communication entre processus

LIF12-Systèmes d'Exploitation

Fabien Rico

Univ. Claude Bernard Lyon 1

30 janvier 2017

Jacques Bonneville	jacques.bonneville@univ-lyon1.fr	TP
Adil Khalfa	adil.khalfa@cc.in2p3.fr	TD + TP
Léo Le Taro	leo.le-taro@inria.fr	TD + TP
Fabien Rico	fabien.rico@univ-lyon1.fr	CM+ TD + TP

Échange de données

Un signal n'est pas suffisant, il faut être capable d'échanger des données.

- de tout type ;
- de taille quelconque ;
- de manière sécurisée ;
- de manière synchrone.

Première idée : pour communiquer on peut utiliser un fichier.
C'est une **mauvaise méthode** car c'est trop lent mais **"ça marche"**.

- Trop lent car on utilise le disque inutilement.
- On peut utiliser un fichier car ils sont partagés entre processus
- *2ème idée* utiliser des « fichiers spéciaux ».

Fichiers spéciaux

Vous avez déjà vu des fichiers spéciaux :

- la *sortie standard* reliée à l'écran du terminal, redirigée par >
- la *sortie d'erreur* reliée à l'écran du terminal, redirigée par 2>
- l'*entrée standard* reliée au clavier, redirigée par <

Généralement dans vos codes vous avez manipulé une notion de haut niveau le *flux* ou *stream* en C (stdout, stderr et stdin) ou C++ (cout, cerr et cin)

On peut utiliser une notion de plus bas niveau le *descripteur de fichier*

Descripteur de fichier

Définition (descripteurs de fichier)

- Ce sont des numéros qui identifient les fichiers ouverts par le processus.
- Ils sont conservés par le système pour éviter l'effacement de fichiers ouverts
- Sous linux on peut les retrouver dans /proc/<pid>/fd/

Les flux sont une structure de données qui encapsule les descripteurs de fichiers.

Trois descripteurs à retenir

- STDIN_FILENO ou 0 : l'*entrée standard*
- STDOUT_FILENO ou 1 : la *sortie standard*
- STDERR_FILENO ou 2 : la *sortie d'erreur*

Manipulation de fichiers en C

- création : `int creat(const char *pathname, mode_t mode)`
- destruction : `int unlink(const char *pathname)`
- ouverture : `int open(const char *pathname, int flags, mode_t mode)`
- fermeture : `int close(int fd)`
- informations : `int fstat(int fd, struct stat *buf)`
- lecture : `ssize_t read(int fd, void *buf, size_t count)`
- écriture : `ssize_t write(int fd, const void *buf, size_t count)`
- déplacement/navigation :
  `off_t lseek(int fildes, off_t offset, int whence);`

Question ?

Les systèmes Unix utilisent le même système de descripteur de fichiers pour les canaux de communications ou de vrais fichiers. Quel est l'intérêt ?

Pipes

Les « fichiers spéciaux »les plus simples que nous pouvons utiliser s'appellent des *tubes*, *canaux* ou *pipes*.

Définition (tube ou pipe)

Les tubes fournissent un canal de communication interprocessus unidirectionnel :

- Ils ont une extrémité d'écriture et une de lecture.
- Ils ont une taille limitée et peuvent être remplis.
- Ils n'ont pas de nom et doivent donc être partagés "dès" la création.

man pipe

- `int pipe(int descriptors [2]);`
- **pipe**() creates a pair of file descriptors and places them in the array pointed to by `descriptors`. `descriptors [0]` is for reading, `descriptors [1]` is for writing.
- On success, zero is returned. On error, -1 is returned.

Exemple (Échange de données)

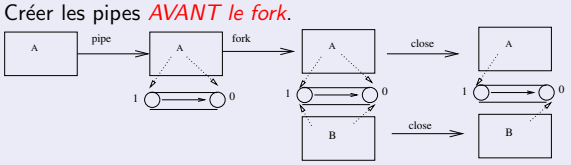
- A démarre.
- **A crée un pipe**
- A utilise **fork**() pour se dupliquer et créer B.

A ferme l'écoute du pipe	**B ferme l'écriture du pipe**
A démarre les calculs.	**B écoute sur le pipe**
A envoie les résultats à B.	**B reçoit les résultats de A.**
A attend la fin de B	B finit les calculs.
	B affiche les résultats.
A traite la fin de B	B se termine.
A se termine	

Utilisation de **pipe**() et **fork**()

A et B doivent partager le même pipe

- Si on crée les pipes quand les processus sont séparés (après le **fork**), A et B vont créer 2 pipes différents (chacun avec 2 descripteurs de fichiers).
- Ça ne marchera pas.
- Créer les pipes *AVANT le fork*.



B ne peut pas lire les variables de A,

mais B est un clone de A !

Exemple :

À cause de sa création :

- Les processus qui peuvent utiliser un pipe doivent avoir un lien familiale.
- Chaque processus doit fermer le descripteur non utilisé
  - pour indiquer la direction du pipe, de A vers B ou le contraire,
  - le lecteur saura ainsi qu'il n'y a plus rien à lire lorsque le rédacteur fermera le *dernier* descripteur en écriture (échec de **read**)
  - rappel : *toujours* libérer les ressources non utilisées !

Exemple (Échange de données)

- A démarre.
- **A crée un pipe**
- A utilise **fork**() pour se dupliquer et créer B.

A ferme l'écoute du pipe	**B ferme l'écriture du pipe**
A démarre les calculs.	**B écoute sur le pipe**
A envoie les résultats à B.	**B reçoit les résultats de A.**
A attend la fin de B	B finit les calculs.
	B affiche les résultats.
A traite la fin de B	B se termine.
A se termine	

Exemple :

```
pid_t code;
int pipefd[2];

if (pipe(pipefd) == -1) {
  fprintf(stderr, "pipe : %s", strerror(errno));
  exit(EXIT_FAILURE);
}
code = fork();
if (code == -1) {/*gestion de l'erreur*/...}
if (code == 0) {
  close(pipefd[1]); /* Ferme l'extrémité d'écriture*/
  .../* Le fils lit dans le tube */
} else {
  close(pipefd[0]); /* Ferme l'extrémité de lecture*/
  .../* Le père écrit dans le tube*/
}
```

Exemple

Exemple (Échange de données)

- A démarre.
- A crée un pipe
- A utilise **fork**() pour se dupliquer et créer B.

- A ferme l'écoute du pipe
- A démarre les calculs.
- **A écrit sur le pipe**.
- A attend la fin de B
- A traite la fin de B
- A se termine

- B ferme l'écriture du pipe
- **B écoute sur le pipe**
- **B lit sur le pipe**
- B finit les calculs.
- B affiche les résultats.
- B se termine.

write

ssize_t **write**(int fd, const void *buf, size_t count);

- **write**() writes up to count bytes from the buffer pointed buf to the file referred to by the file descriptor fd.
- On success, the number of bytes written is returned (zero indicates nothing was written). On error, -1 is returned, and errno is set appropriately

read

ssize_t **read**(int fd, void *buf, size_t count);

- **read**() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf
- If count is zero, **read**() returns zero and has no other results. If count is greater than SSIZE_MAX, the result is unspecified.
- On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number.[...] On error, -1 is returned, and errno is set appropriately. In this case it is left unspecified whether the file position (if any) changes.

Résumé

read()/write()

- Ces deux fonctions lisent/écrivent un nombre fixe d'octets.
- Tout passe par buf un tableau qui doit être réservé.
- Par défaut **read** est bloquant et **write** non bloquant.
- Les données peuvent être de n'importe quel type mais
  - Il faut lire la même chose que ce qu'on a écrit
  - On envoie une zone mémoire, il faut donc que les données soient contiguës en mémoire
- Le pipe est en *mode octet*. Il n'y a pas préservation de la taille des messages. On peut par exemple envoyer une chaîne de caractères et la lire caractère par caractère.

Et alors, ça marche ?

```
if (pipe(pipefd) == -1) {...}
code = fork();
if (code < 0) {...}
if (code == 0) {     /* Le fils lit dans le tube */
  char buf;
  ...
  while ((res = read(pipefd[0], &buf, 1)) != 0) {
    fprintf(stdout,"J'ai lu %c\n",buf);
  }
  ...
} else {/* Le père écrit argv[1] dans le tube */

  const char* buf = "coucou";
  res = write(pipefd[1], buf, strlen(buf));
  ...
}
```

Pipe nommé

- Deux processus qui utilisent un pipe *doivent avoir un lien familial*
- Sinon que fait-on ?
- Il faut *donner un nom au pipe* pour que 2 processus sans rapport l'utilisent.
- Comme pour les fichiers.

Définition (Tube nommé)

Un *tube nommé* ou *FIFO* est un tube (pipe) qui a un nom dans le système de fichiers.

- Il peut être géré comme un fichier
  - nom, droits.
  - ouverture (**open**), fermeture (**close**),
  - lecture (**read**), écriture (**write**)
- C'est un tube, une fois ouvert, cela s'utilise comme un pipe.

Utilisation

- Il faut créer le pipe (ou utiliser un pipe existant)
  `int mkfifo(const char *pathname, mode_t mode);`
  - pathname : est le nom du « fichier ».
  - mode : représente les droit d'accès (comme sous unix)
- Chaque processus doit l'ouvrir
  `int open(const char *pathname, int flags );`
  - pathname : le nom
  - flags : information d'ouverture notamment O_RDONLY pour la lecture et O_WRONLY pour l'écriture.
- S'il a été créé, ne pas oublier de le supprimer à la fin
  `int unlink(const char *pathname);`

Le lecteur

```
if (mkfifo("/tmp/fifo.plop", 0644)==-1) {
  fprintf(stderr, "probleme fifo %s : %s\n",
          argv[1], strerror(errno));
  exit(1);
 }

fd = open("/tmp/fifo.plop", O_RDONLY);
if (fd <= 0) { //erreur
  ...}
while ((res = read(fd, &buf, 1)) != 0) {
    fprintf(stdout, "Je lit %c\n", buf);
 }

...
close(fd);
if (unlink("/tmp/fifo.plop")==-1) {  //erreur
...}
```

Le rédacteur

```
fd = open("/tmp/fifo.plop", O_WRONLY);
if (fd <= 0) {
  fprintf(stderr, "ouverture du fifo %s : %s\n",
          argv[1], strerror(errno));
  exit(1);
 }

res = write(fd, "coucou", strlen("coucou"));
if (res == -1) {//erreur
  ...}
fprintf(stdout, "Fin de la liaison\n");
close(fd);
```

À travers le réseau ?

- Les tubes permettent de communiquer entre processus d'une même machine.
- Mais avec les processus distants ?
- Que faut-il de plus ?
  - Un nom valable sur le réseau.
  - Un protocole de transfert.

Pour étendre la notion de tube et son utilisation, on a défini les *sockets*

Socket

Définition (Socket)

La socket (ou prise) est une notion qui étend celle de tube. De la même manière, la socket permet de définir un canal de communication entre deux processus, mais :

- elle permet l'utilisation du réseau,
- elle permet de choisir différents protocoles de communication.

Deux questions se posent pour la communication :

- Mode connecté ou non :
  - la communication est maintenue (comme pour le téléphone)
  - refaite entre chaque message (comme la poste)
- Mode paquet ou flux :
  - les frontières de messages sont conservées.
  - tous les messages sont fusionnés.

Sockets réseaux

Les *sockets* permettent la communication entre processus locaux comme les *tubes*, mais leur intérêt est surtout d'utiliser le réseau.
On peut alors utiliser les modes :

- SOCK_STREAM : connecté (TCP) et flux.
- SOCK_DGRAM : non connecté (UDP) donc sans garantie sur l'arrivée et l'ordre des paquets, mais préservant les paquets.
- SOCK_SEQPACKET : connecté et préservant les paquets.

Le dernier mode n'est que rarement implémenté dans les langages de bas niveau donc inutilisable pour des raisons de portabilité.

Nous étudierons uniquement le mode SOCK_STREAM

Mode connecté

En mode connecté, il y a deux acteurs :

- Le *serveur* qui attend la connexion.
- Le *client* qui initie la connexion.

Afin de connecter un programme à distance, il faut 2 informations :

- un code qui permet de trouver la machine : *l'adresse IP*,
- un code qui permet au système de la machine de retrouver la socket : *le port*.

En général, les services (programmes) les plus connus ont un port identique sur toutes les machines (port réservé) par exemple :

- http : 80, https : 443
- ssh : 22
- mysql : 3306
- ...

Voir le fichier /etc/ services

Vue d'ensemble

L'interface des sockets a peu changé depuis sa création. Cela indique sa qualité et sa souplesse mais aussi sa difficulté d'utilisation.
On peut rapprocher son utilisation de celle d'une carte de téléphone prépayée :

- Vous êtes serveur (vous espérez des appels téléphoniques)
  - vous consultez les offres des différents opérateurs ;
  - vous choisissez le premier qui vous convient, l'achetez et faite en sorte d'apparaître sur l'annuaire ;
  - vous allumez le téléphone et attendez l'appel ;
  - un certain nombre de fois
    - vous répondez à un client
    - vous discutez avec lui ;
    - vous ou votre interlocuteur raccroche ;
  - vous jetez votre carte.
- Vous êtes client (vous souhaitez appeler)
  - vous consultez l'annuaire pour savoir quel(s) sont le(s) numéro(s) du serveur ou du service que vous voulez contacter ;
  - vous achetez la carte compatible avec le serveur (bof) ;
  - vous contactez le serveur ;
  - vous discutez ;
  - vous ou votre interlocuteur raccroche ;
  - vous jetez votre carte.

Les fonctions C

- Consulter les offres ou l'annuaire : **getaddrinfo**() ;
- acheter une carte compatible : **socket**() ;
- apparaître sur l'annuaire : **bind**() ;
- allumer pour attendre : **listen** () ;
- attendre puis répondre : **accept**() ;
- appeler le serveur : **connect**() ;
- raccrocher, se desinscrire, jeter la carte : **close**() ;

Consulter - I

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

- node : la machine demandée (nom ou adresse) ;
- service : le *service* (*http, ldap, rdp*) ou le numéro de port (*"80", "386", "3089"*) *attention c'est une chaîne de caractère* ;
- hints : le formulaire de requête (pour filtrer certaines offres) ;
- res : l'adresse d'un pointeur ou sera stocké le résultat.

Consulter - II

La demande est différente selon qu'on est :

- Serveur – on veut ouvrir un service particulier sur la machine locale

```
memset(&hints, 0, sizeof(struct addrinfo));
hints.ai_family = AF_UNSPEC;     /* IPv4 ou IPv6 */
hints.ai_socktype = SOCK_STREAM; /* socket flux connectée */
hints.ai_flags = AI_PASSIVE;     /* Les signifie que toutes les addresse
res = getaddrinfo(NULL, "80", &hints, &result);
if (res != 0) { // c'est une erreur
  fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(res));
  exit(1);
}
```

- Client – on veut contacter l'une des adresses (ipv4 ou ipv6) d'une machine dont le nom est connu

```
memset(&hints, 0, sizeof(struct addrinfo));
hints.ai_family = AF_UNSPEC;     /* IPv4 ou IPv6 */
hints.ai_socktype = SOCK_STREAM; /* socket flux connectée */
res = getaddrinfo("mamachine.univ-lyon1.fr", "http", &hints, &result);
if (res != 0) { // c'est une erreur
  fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(res));
  exit(1);
}
```

Création d'une socket (en détail)

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

- domaine : le domaine d'utilisation, AF_LOCAL pour un rôle proche du tube, AF_INET ou AF_INET6 pour internet IPV4 ou IPV6
- type : mode de communication (SOCK_STREAM, SOCK_DGRAM, SOCK_SEQPACKET,...)
- protocol : propocol utilisé pendant la communication (généralement il n'y en a qu'un possible ⇒ on met 0).

```
int s;
s = socket(AF_INET, SOCK_STREAM, 0);
if (s == -1) {
  perror("Erreur à la création de la socket");
  exit(1);
}
```

Obtention d'un nom (en détail)

Pour être contacté depuis internet, la socket doit avoir un nom, c'est à dire réserver un port sur l'une des adresses de la machine sur laquelle il se trouve.

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

- sockfd : la socket
- sockaddr : une structure de données décrivant l'adresse et le port.
- addrlen : la taille de sockaddr.

Connexion (en détail)

Le client doit contacter le serveur

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

- serv_addr : l'adresse et le port du serveur
- addrlen : la longueur de serv_addr

```
if(connect(s, (struct sockaddr *)&sin, sizeof(sin)) <0)
{
  perror("connect");
  exit(1);
}
```

Création, obtention d'un nom et connexion en pratique

Les fonctions **socket**, **bind** et **connect** sont des fonctions anciennes qui imposent de connaître le domaine utilisé (donc de faire la différence entre ipv4 et ipv6).

⇒*Comment contacter un serveur si on ne connaît pas la famille d'adresses utilisée ?*

La fonction **getaddrinfo**() est apparue pour cela :

- elle permet une résolution de nom ;
- elle retourne plusieurs adresses dans une liste chaînée ;
- elle utilise la même forme pour toutes les familles ;
- pour chaque adresse, elle propose les valeurs des arguments nécessaires aux fonctions **socket**, **bind** et **connect**.

En pratique, on fait appel à **getaddrinfo**, et on parcours le résultat en s'arrêtant dès que l'une des propositions fonctionne.

Mise en place coté serveur

```
bon = 0;
for (rp=result; rp!=NULL; rp = rp->ai_next) {
  // on parcourt la liste pour trouver une adresse qui convient

  s = socket(rp->ai_family, rp->ai_socktype,rp->ai_protocol);
  if (s == -1) {
    perror("Création de la socket");
    continue;
    // si le résultat est -1 cela n'a pas fonctionné on recommence avec la prochaine
  }

  // si la socket a été obtenue, on essaye de réserver le port
  res = bind(s, rp->ai_addr, rp->ai_addrlen);
  if (res == 0 ) {
    bon = 1;
    break;
    // cela a fonctionné on sort de la boucle
  }

  // sinon le bind a été impossible, il faut fermer la socket
  perror("Imposible de réserver l'adresse");
  close (s);
}

if (bon == 0) { // Cela n'a jamais fonctionné
  fprintf(stderr, "Impossible d'obtenir une adresse\n");
  exit(1);
}
```

Mise en place coté client

```
bon = 0;
for (rp=result; rp!=NULL; rp = rp->ai_next) {
  // on parcourt la liste

  s = socket(rp->ai_family, rp->ai_socktype,rp->ai_protocol);
  if (s == -1) {
    perror("Création de la socket");
    continue;
    // si le résultat est -1 cela n'a pas fonctionné on recommence
  }

  // si la socket a été obtenue, on essaye de se connecter
  res = connect(s, rp->ai_addr, rp->ai_addrlen);
  if (res == 0 ) {// cela a fonctionné on est connecté
    bon = 1;
    break;
  }

  perror("Imposible de se connecter");
  close (s);
}

if (bon == 0) { // Cela n'a jamais fonctionné
  fprintf(stderr, "Impossible de se conecter\n");
  exit(1);
}
```

Attente de connexions - I

La socket du serveur doit se mettre en attente passive par la commande
`int listen(int sockfd, int backlog);`

- backlog : initialement le nombre de demandes de connexions, qui pouvaient être en attente avant traitement, sans que le système réponde par un refus.

Attente de connexions - II

Une socket en mode connecté n'accepte que la présence de 2 processus. Le *serveur* doit donc se mettre en attente et à chaque demande d'un nouveau client il doit créer une nouvelle socket pour permettre la discussion et l'attente de nouvelles connexions
`int accept(int sockfd, struct sockaddr *adresse, socklen_t *longueur);`

- adresse (*résultat*) : permet d'obtenir l'adresse du client.
- longueur (*résultat*) : la longueur de adresse.
- *retourne* : le *descripteur de fichier* de la socket de discussion

Attention La variable adresse doit être suffisamment grande pour contenir une adresse ipv4 ou ipv6. Contrairement à ce qui est écrit, il ne faut pas utiliser une variable de type `struct sockaddr` mais de type `struct sockaddr_storage`.

Attente de connexions - III

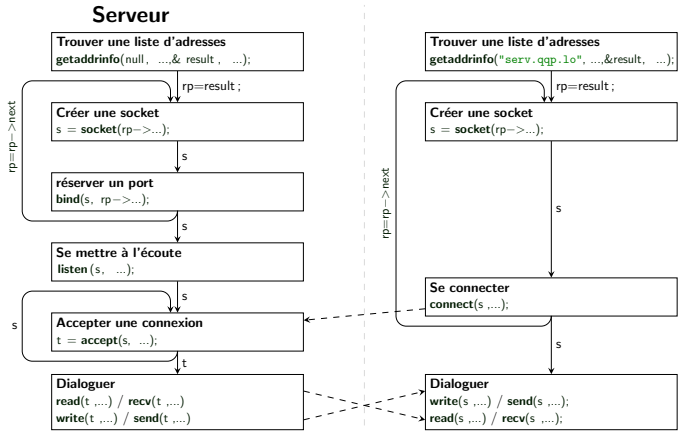
```
int t;
struct sockaddr_storage tadr;
socklen_t taddrlen = sizeof(tadr);
char hname[NI_MAXHOST], sname[NI_MAXSERV];

sock_err = listen (s, 5);
if (sock_err = -1) {
  perror("listen");
  close(s);
  exit(1);
}
t = accept (s, (struct sockaddr *) &tadr, &taddrlen);
//s : la socket d'attente
//t : la socket de discussion
res = getnameinfo((struct sockaddr*)&taddr, taddrlen,
                  hname, NI_MAXHOST,
                  sname, NI_MAXSERV,
                  NI_NUMERICSERV);
if (res != 0) {
  fprintf(stderr, "getnameinfo: %s\n", gai_strerror(res));
  exit (1);
}
printf ("La socket %d a eu un client depuis %s sur le port %s\n",
        s, hname, sname);
```

Primitives de communication

Une fois la connexion effectuée, les processus peuvent envoyer des données grâce à **read** et **write** mais aussi grâce à des primitives dédiées :

- `ssize_t recv(int s, void *buf, ssize_t len, int flags);`
- `ssize_t send(int s, const void *buf, size_t len, int flags);`
  - s : la socket
  - buf : les données, au plus de taille len
  - flags : des options



Lire des données est une opération blocante et en envoyer peut aussi l'être.
Attention aux interblocages !

Ce qu'il faut retenir

Toutes ses fonctions ont l'air compliquées, le plus simple serveur en C demande 120 lignes de codes.

- La pluspart du temps elles n'ont *aucun intréret*.
- En général, il suffit de repérer et modifier les paramêtres important :
  - le `port` découte pour mettre en place le serveur ;
  - le `nom` et le `port` du serveur a contacter pour le client.
- Il faut bien comprendre que :
  - le serveur met en place une première socket (acceptor en C++) pour écouter les messages du client ;
  - une fois que le client s'est connecté, le serveur crée une *autre socket* pour discuter avec lui ;
  - on utilise des **read** et **write** pour envoyer ou recevoir des données.

Où est la difficulté

Pourquoi consacre-t-on autant de temps à vous parler de cela ?

En général, les langages fournissent des fonctions simples pour créer des sockets :

- ServerSocket ServerSocket(int port) pour créer une socket d'écoute
- Socket ServerSocket::**accept**() pour attendre et accepter un client
- Socket ClientSocket(String host, int port) du coté client pour se connecter.

En C/C++ ce n'est pas vrai, mais nous vous fournissont une bibliothèque de fonctions qui le permettent.
La difficulté est ailleur !

Les problèmes fréquemment rencontrés

L'utilisation de ces primitives de communications entraine plusieurs difficultés :

1. Problème de gestion des pointeurs.
2. Utilisation bas niveau des données.
3. Frontières des messages

Grâce à votre niveau en programmation, le premier point ne doit pas vous causer de problèmes.

Données bas niveau

read, **write** et leur équivalent permettent de transférer des *zones de mémoire*.
On peut transférer des données complexes simplement :

```
typedef struct etudiant {
  int numero;
  char nom[255];
  char prenom[255];
} Tetudiant;
...
Tetudiant e;
...
res = write(fd, (const void *) &e, sizeof(e));
```

Mais
- Il faut que les données soient contiguës en mémoire (pas de pointeur)
- Il faut que les données soient définies de façon identique à chaque bout
- Attention aux options de compilation (align)

Données bas niveau et sérialisation

Les canaux de communication traitent les données comme un ensemble d'octets. On peut *calquer* une structure existante sur les données brutes.

- Aucun traitement n'est fait.
- Les pointeurs ne peuvent pas être transférés.
- Les données doivent avoir une taille connue de part et d'autre du canal.

Cela complique beaucoup le transfert de structure de données évoluées comme les objets ou des tableaux associatifs.
Certains langages de haut niveau permettent de transformer n'importe quelle donnée en flux. C'est la *sérialisation*.

- Objet `Serializable` en JAVA
- fonction `serialize` en PHP
- ...

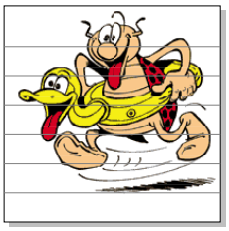
Vous devez utiliser ces outils pour transférer des données complexes via le réseau.

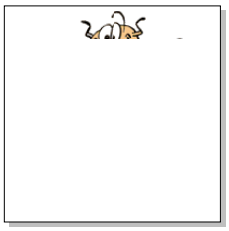
Frontière de messages

Tous les messages sont vus comme un seul flux de données.

- Tout se passe bien si chaque message arrive et est lu immédiatement
- Mais que se passe-t'il si
  - un message arrive en 2 morceaux .
  - deux messages arrivent en même temps ?

Si on se lève tôt :

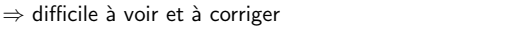




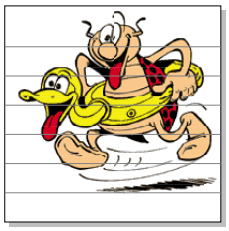
```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  write(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  read(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

Le réseau est peu chargé, les paquets arrivent facilement



Plus tard :



```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  write(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  read(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

Le réseau est chargé, les paquets sont coupés en deux, mais le récepteur ne le sait pas.

Que nous apprend l'exemple ?

Le programme donné a un bug

- visible sur les gros transferts, mais jamais lors de tests simples.
- qui n'apparaît que dans des conditions de stress du réseau
- ⇒ difficile à voir et à corriger

Messages

Le transfert de flux d'octets n'est pas naturel, et cause de nombreuses erreurs.

- Il faut vérifier que tout le message est arrivé
- Il faut vérifier qu'on ne déborde pas sur le message suivant

Pour éviter les erreurs, il faut souvent définir un protocole de transfert qui reconstruit les frontières des messages.

Cela peut-il arriver ?

Les données n'arrivent pas ?

Exemple

- Un serveur en écoute qui affiche des données

```
> ./testlecture.ex -p 8083 -a 20 PRINT
```

- Un client qui les envoie

```
> nc localhost 8083
```

Cela peut-il arriver ?

Exemple (La machine ajoute des choses à la fin)

- Un serveur en écoute qui affiche des données sans attendre

```
> ./testlecture.ex -p 8083 -t 100  PRINT
```

- Un client qui envoie plusieurs messages

```
> nc localhost 8083
Voila un message long à lire
Et un court
```

Cela peut-il arriver ?

Exemple (Certaines données se perdent ?)

- Un serveur en écoute qui affiche des données sans attendre et fait attention de bien annuler le buffer

```
> ./testlecture.ex -p 8083 -w 1 -z -t 13  PRINT
> ./testlecture.ex -p 8083 -w 1 -z -t 14  PRINT
> ./testlecture.ex -p 8083 -w 1 -z -t 100  PRINT
```

- Un client qui envoie le contenu d'un fichier

```
> cat ./test.txt | nc localhost 8083
```

Cela peut-il arriver ?

Exemple (Les données sont modifiées en passant dans le réseaux ?)

- Un serveur en écoute avec la fonction de lecture qu'on vous fourni (par exemple lecture d'une ligne)

```
> ./testlecture.ex -p 8083 -l toto.pdf
```

- Un client qui envoie le contenu d'un fichier pdf

```
> cat ./cm-IPC.pdf | nc 127.0.0.1 8080
```

Cela peut-il arriver ?

Exemple (Les valeurs sont modifiées par le réseau)

- Un serveur ecrit en java qui écoute sur le réseau

```
> ./java indian_server
```

- Un client en C qui envoie des entiers

```
> ./client_indian.ex
34
```

Que peut-il arriver ?

Tout ce que vous savez c'est que *grâce à TCP, tous les octets arrivent dans le bon ordre* sinon, une erreur serait détectée. Mais vous ne savez pas :

- si un envoi arrive d'un seul coup ;
- si plusieurs envois arrivent en même temps ;
- quelle est la taille de l'information à lire ;
- quel processus doit lire et à quel moment doit-il le faire ;
- quand doit-il s'arrêter ;
- si les deux programmes utilisent la même facon de coder les choses (seul les caractères sur 1 octet sont standards).

Il est donc nécessaire de définir un protocole de communication qui permet de savoir quand lire et écrire, comment détecter la fin d'un message, quelles informations sont échangées...

Ce que nous apprennnent les erreurs

Lorsque qu'on a récupéré un exemple sur le net et échangé un texte de 30 caractères entre 2 programmes, on a fait le plus simple. Il reste :

- à savoir qui envoie et qui lit pour éviter les attentes infinies ;
- à reconstituer les messages ;
- à définir un encodage robuste ;
- à definir une réaction en cas d'erreur.

Quelques solutions

Il faut toujours définir qui doit parler et ce qui est envoyé. C'est le rôle des protocoles de communication. A chaque échange de données, il faut faire en sorte que les limites du message échangé soit connues ou facile à reconnaitre.

- Utiliser un fanion exemple, fin de ligne, 0,...Par exemple, ce qui est envoyé par un client FTP (une fin de ligne), HTML (une ligne vide), mail (une ligne contenant un .). Par contre, cela pose des problèmes si le fanion existe dans le message.
- N'utiliser la socket que pour un message, on ferme pour signifier la fin. Par exemple pour la connexion de données de FTP, pour la fin d'une page HTML dans HTML/1.0... Mais cela impose de refaire des connexions pour chaque message.
- Utiliser un entête qui décrit le message (au moins type et taille). Par exemple les websockets (type, taille, message), entête IP...

Conclusion

Communication

- Fichiers
- Signaux
- Canal de communication (tube, socket, RPC)
- Mémoire partagée

Difficulté

- Notion de protocoles.
- Quelle est la véritable difficulté ?