Processus, suite : communication entre processus

ASR5 - Système d'Exploitation

Fabien Rico

Univ. Claude Bernard Lyon 1

2 mars 2018

Fabien Rico	fabien.rico@univ-lyon1.fr	CM+TD+TP
Jacques Bonneville	jacques.bonneville@univ-lyon1.fr	TP
Adil Khalfa	adil.khalfa@cc.in2p3.fr	TD + TP
Yves Caniou	yves.caniou@univ-lyon1.fr	TP
Dorra Boughzala	dorra.boughzala@ens-lyon.fr	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 peut fonctionner"**.

- 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 fichier.

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 fichier pour les canaux de communication 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.

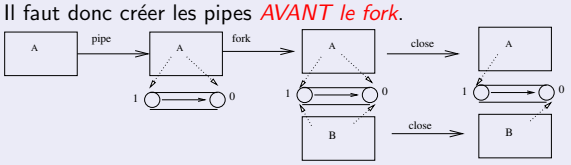
- **A ferme l'écoute du pipe**
- A démarre les calculs.
- **A envoie les résultats à B.**
- A attend la fin de B
- A traite la fin de B
- A se termine

- **B ferme l'écriture du pipe**
- **B écoute sur le pipe**
- **B reçoit les résultats de A.**
- B finit les calculs.
- B affiche les résultats.
- B 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 fichier).
- Ça ne marchera pas.
- Il faut donc 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 familial.
- 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**
- A démarre les calculs.
- **A envoie les résultats à B.**
- 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 reçoit les résultats de A.**
- B finit les calculs.
- B affiche les résultats.
- B se termine.

Exemple :

```c
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 ?

```c
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 droits 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 transport de données.

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

Identité sur le réseau

Un dialogue via le réseau suppose l'existance de 2 processus sur 2 ordinateurs qui sont capablent de se reconnaitre et se transmettre des informations.
Dans les modèles le plus souvent utilisé (TCP/IP et UDP/IP) les données sont :

- adressées à un ordinateur particulier via une *adresse IP destination* ;
- adressées à un processus particulier via un *numéro de port destination*.

De la même manière, l'origine des données est connue via *l'adresse IP source* et le *numéro de port source*.
Ces 4 valeurs permettent d'identifier un échange d'informations voire une connexion.

Modèle de transport

Il y a 2 modèles couramment utilisés :

- le modèle déconnecté, *comme les lettres*, les données sont envoyées et reçues sous forme de *datagramme* :
  - il n'y a pas d'outil pour savoir qu'une donnée est perdue ;
  - il n'y a pas d'outil pour assurer l'ordre dans lesquelles elles arrivent.
  - habituellement cela est implémenté par le **protocole UDP**
- le modèle connecté, *comme le téléphone*, une connexion est mise en place de qui permet de faire un échange :
  - si des paquets disparaissent ou arrivent dans le désordre, cela est automatiquement corrigé ;
  - la connexion est maintenue et s'il elle se coupe de manière irréversible, une erreur ou une exception sera générée (`Brocken Pipe`) ;
  - il n'y a pas d'outil pour délimiter les messages ou les « ensembles de données » (page web, fichier, entête, envoie,...)

Nous étudierons le mode connecté

Client serveur

En mode connecté, il faut établir la connexion puis la terminer. Cela implique 2 acteurs :

- Le *serveur* qui attend l'établissement et accepte le client.
- Le *client* qui initie la connexion.

Chacun à des actions à faire :

- Le *client* doit juste contacter le serveur dont il connait le *nom de la machine* (ou son @IP) et le *port*.
- Le *serveur* doit faire 2 choses :
  - mettre en place une *socket d'écoute*, notamment, il *réserve un port sur certaines @IP* auprès du système ;
  - créer la *socket* permettant de discuter avec le client qui se connecte.

Une même socket d'écoute permet de créer des connexions avec plusieurs clients. Par exemple, une serveur web qui écoute sur le port 80 répond à plusieurs clients.

Question

Un serveur est simultanément en contact avec plusieurs clients. Lorsqu'il reçoit des données, comment reconnait-il le client qui les envoie ?

Socket

L'outil central de la communication réseau est la *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.
- Elle est bidirectionnel.

Rappel, nous n'étudierons que le mode connecté ce qui actuellement utilise le protocole TCP/IP

Interface de programmation

L'interface des sockets a peu changé depuis sa création. Cela indique sa qualité et sa souplesse mais explique aussi sa difficulté d'utilisation.
Comme pour les *tubes* ou les *fichier*, en C/C++, la socket est représentée par un entier.
Rappelons qu'il y à 3 actions à effectuer :

- Le serveur doit mettre en place une *socket d'écoute*.
- Le client doit se connecter au serveur pour créer la *socket de discussion* avec lui.
- Cela n'est possible si le serveur accepte le client et crée une *socket de discussion* avec lui.

Fonctions utilitaire

La première fonction de l'API est utile pour le client et le serveur, elle permet de résoudre les noms de machine en adresses et les noms de services en numéros de port.

```
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 caractères* ;
- hints : un formulaire de requête (pour filtrer certains résultats) ;
- res : l'adresse d'un pointeur où sera stocké le résultat.

Création de la socket d'écoute

Il faut créer une socket qui ne sert que pour être contacté par le client.

- Les paramètres sont :
  - le port utilisé ;
  - la liste des adresses possibles (par défaut toutes les adresses).
- Le résultat est une *socket d'attente*, *d'écoute*, *de serveur* (Java) ou un *endpoint* (`C#`, `boost::asio`...)
- Elle ne permet pas de transmettre de donnée.

Dans la plupart des langages, cette action est effectuée par une fonction, en C/C++ il en faut plusieurs :

- **getaddrinfo** pour la résolution de noms et la création des objets sockaddr (les adresses).
- int **socket**(int domain, int type, int protocol) pour créer l'entier « socket ».
- int **bind**(int sfd, const struct sockaddr *ad, socklen_t alen) pour réserver un numéro de port d'une certaine adresse IP du système.
- int **listen**(int sfd, int bl) qui transforme effectivement la socket en socket d'écoute.

Connexion du client

Le client doit se connecter au *endpoint* du serveur et lui demander de créer une socket de discussion.

- Si le serveur n'a pas de *endpoint*, la requête sera refusée par le système de la machine contactée.
- Si le paquet est filtré par un pare-feu ou que le serveur ne peut pas répondre, cela peut prendre du temps.
- Les paramètres nécessaires sont le nom et le port du *endpoint*.
- Le résultat est une socket de discussion avec le serveur.

En C/C++, il faut là aussi plusieurs fonctions :

- **getaddrinfo** pour la résolution de noms et la création des objets sockaddr (les adresses).
- int **socket**(int domain, int type, int protocol) pour créer l'entier « socket ».
- int **connect**(int sfd, const struct sockaddr *s_add, socklen_t alen); pour se connecter au serveur.

Acceptation du client

Une socket en mode connecté est un canal de communication entre 2 processus. Le *serveur* doit donc créer une *socket de discussion* pour chaque nouveau client.

- Toutes les données envoyées par le client pourrons être lues depuis cette socket.
- Toutes les données écrites sur cette socket seront envoyées à ce client particulier.

En C/C++, il y a une fonction :
int **accept**(int sockfd, struct sockaddr *adr, socklen_t *alen);

- 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 ou -1 en cas d'erreur.

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 serveur (suite)

```c
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);
```

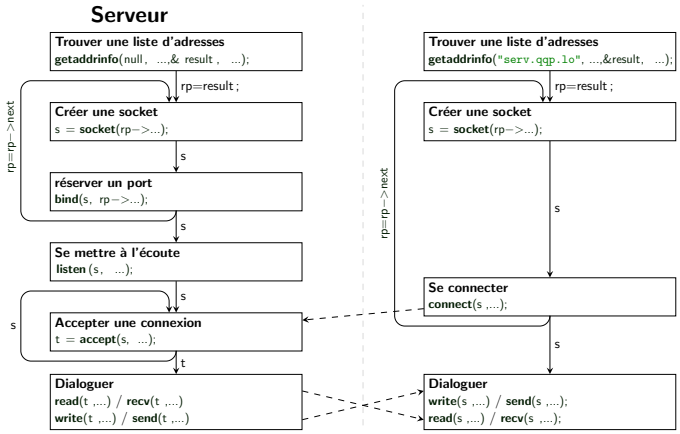
Mise en place coté client

```c
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);
}
```

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 bloquante et en envoyer peut aussi l'être. Attention aux interblocages !

Ce qu'il faut retenir

Toutes ces fonctions sont complexes à manipuler,

- Le plus simple serveur en C demande 120 lignes de code alors qu'il n'y a que quelques actions à effectuer (voir les autres langages).
- La plupart des exemples proposés sur internet utilisent du code qui est incompatible avec IPv6.
- Certaines fonctions n'ont *pas d'intérêt*, elle existent pour des raisons historique.
- En général, il suffit de partir d'un code fonctionnel, de repérer et modifier les paramètres importants pour créer une connexion. Ces paramètres sont :
  - le `port` d'écoute pour mettre en place le serveur (éventuellement les adresse d'écoutes si on ne souhaite pas toutes les autoriser) ;
  - le `nom` et le `port` du serveur à contacter pour le client.

Bibliothèque d'application

Pour évitez les complications inutiles, en TP nous vous demanderons d'utiliser une librairie d'application : `Socklib`.

- int socklib :: CreeSocketServeur(const string &port) : crée un *endpoint*.
- int socklib :: AcceptConnexion(int s) : accepte un nouveau client qui fait une demande sur le *endpoint* s et renvoie une socket de discussion.
- int socklib :: CreeSocketClient(const string &host, const string &port) : crée une socket de discussion de client en se connectant sur le *endpoint* `host:port`. Le résultat est la socket de discussion avec ce client.

Difficulté

Si on regarde la plupart des documentation, la mise en place de la connexion occupe une grande partie des explications.

- Elle est cependant relativement simple.
- Il est facile de limiter une API à 3 fonctions tout en restant général.
- Mais ou est la difficulté ?

La difficulté est dans le transfert de données.

- Il n'y a pas de méthode efficace dans tous les cas :
  - transfert de texte, de données binaires ...
  - transfert d'ensemble plus ou moins importants (message, page web, fichier ...)
  - possibilité ou non de bloquer les programmes

Problème du transfert de données

La plupart des erreurs viennent de problèmes dans la programmation des transfert de données.
Le système fournis comme primitives :

- **recv**, **read** pour lire des octets.
- **send**, **write** pour écrire des octets.

Il y a 2 niveaux de programmation pour le transfert de données :

- Utiliser les primitives de base pour créer des primitives plus complexes : entiers, textes, données binaire de taille connues ...
- Utiliser les primitives complexes pour implémenter un véritable protocole de communication.

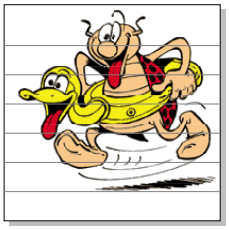
Pourquoi se compliquer la vie ?

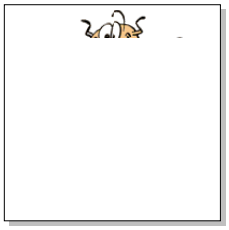
Tous les messages sont vus comme un seul flux de données.

- 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 ?

Exemple d'un transfert d'image

Tôt le matin

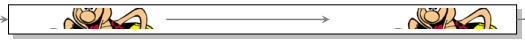




```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  write(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

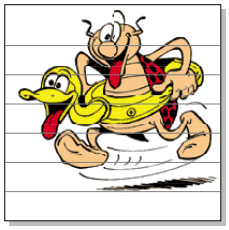
```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  read(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

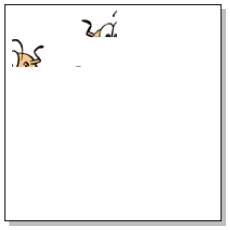
Le réseau est peu chargé, les paquets arrivent facilement



Exemple d'un transfert d'image

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.

Question

Comment feriez-vous pour transférer une image ?

Comment trouver une solution ?

Pour réussir un échange de données, il faut :

- Empêcher que le hasard ou l'environnement perturbe la communication.
  - Il faut implémenter des primitives complexes.
  - Par exemple vector<char> read_all(int T) doit lire un tableau d'octet de taille T exactement, ni plus ni moins.
  - Si moins d'octet arrive, il doit attendre, s'il y en a plus, il doit laisser les octets en trop ...
- S'assurer à tout moment que chaque processus sache ce qu'il doit faire :
  - Envoyer ou lire des données.
  - La taille des données ou le moyen de stopper la lecture.
  - Si on ne le fait pas, on risque *l'interblocage*.
  - C'est le rôle du *protocole de communication*.

Exemple de protocole : HTTP

Le protocole HTTP (1.1) permet au client (un navigateur) de télécharger le contenu d'une page web.

- Le client parle en premier.
- Il envoie une requête avec :
  - un entête sous forme de lignes de texte terminées par \r\n ;
  - l'entête mentionne la page à télécharger et le mode de transfert sur la première ligne ;
  - des valeurs de champs vous la forme NOM: VALEUR\r\n ;
  - l'entête se termine avec une ligne vide ;
  - dans certains modes (POST), des données sont transférées ensuite.
- Le serveur répond avec un entête sur le même modèle qui mentionne la méthode pour lire la page.
- Ensuite le serveur envoie la page (il y a plusieurs méthodes).
- Dès que la page a été envoyée, le client peut en demander une autre.

À chaque instant, le client et le serveur savent ce qu'ils doivent lire et comment le faire. Le moindre décalage provoque une erreur de lecture ou un blocage.

Votre travail

A la fin de l'UE, vous devez :

- Savoir utiliser les primitives de base (`read/write`) pour implémenter les primitives complexes :
  - lire une ligne de texte terminée par \r\n ;
  - lire un tableau de taille fixe ;
  - lire un entier ;
  - ...
- Savoir utiliser ces primitives complexes pour proposer un protocole fonctionnel.

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 fournit (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 écrit en java qui écoute sur le réseau

```
> ./java indian_server
```

- Un client en C qui envoie des entiers

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

Conclusion

Communication

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

Difficulté

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