Sockets

LIFSE - Systèmes d'exploitation

L. Gonnord, **N. Louvet**, M. Moy, G. Pichon, F. Zara

30 mars 2026

Problématique

On veut pouvoir échanger des données : de tous les types, de toutes les tailles, de manière sécurisée et synchronisée.

Pour la communication entre processus, nous avons déjà vu :

- les signaux $\rightsquigarrow$ peu adaptés à l'échange de données.
- les fichiers réguliers $\rightsquigarrow$ certes oui, mais...
  - « plutôt lent » car cela sollicite le disque,
  - problèmes d'accès conccurents entre lecteurs et rédacteurs. UE LIFPCA l'an prochain !
- fichiers spéciaux :
  - entrées et sorties standards des processus,
  - tubes anonymes (processus sont créés dans un même programme),
  - tubes nommés (qui ont un nom dans le système de fichiers).

Ces modes fonctionnent entre processus d'un même système...

Il faut aussi pouvoir échanger entre processus via le réseau : c'est le rôle des sockets.

Les systèmes Unix utilisent le même système de descripteur de fichiers pour (presque) tous les types de fichiers : réguliers, spéciaux, blocs...

Cela permet d'utiliser les mêmes primitives d'accès, que ce soit sur des canaux de communication ou de vrais fichiers :

- on peut toujours utiliser `read()` et `write()` ;
- pour les sockets, il existe des adaptations comme `recv()` et `send()`.

On peut utiliser des bibliothèques de haut niveau pour les fichiers :

- la bibliothèque C avec `stdio.h` : `FILE *`, `fread()`, `fwrite()`...;
- la bibliothèque C++ avec `iostream` : `cin`, `>>`, `cout`, `<<`...;
- il existe des biblio. pour les échanges réseau (`libcurl`, `Boost.Asio`, ...).

On s'en tient aux primitives de bas niveau (`recv()`, `send()`)

Quelques notions sur les réseaux

Un dialogue via le réseau suppose l'existence de deux processus sur deux ordinateurs qui sont capables de se reconnaître et de se transmettre des informations.

Dans les modèles les plus souvent utilisés (TCP/IP et UDP/IP) les données sont adressées :

- à un ordinateur particulier via une adresse IP destination ;
- 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 une connexion.

Deux modèles de transport sont couramment utilisés :

- le modèle déconnecté (analogie : courrier papier), les données sont envoyées et reçues sous forme de paquets :
  - pas d'outil pour savoir si une donnée est perdue ;
  - pas d'outil pour assurer l'ordre dans lequel elles arrivent.

  ↪ protocole UDP (User Datagram Protocol)

- le modèle connecté (analogie : téléphone), une connexion est mise en place qui permet de gérer les paquets échangés :
  - si des paquets disparaissent ou arrivent dans le désordre, cela est automatiquement corrigé ;
  - la connexion est maintenue ; si elle se coupe, une erreur (`Broken Pipe`) est générée ;
  - pas d'outil pour délimiter les messages ou les données (page web, fichier...).

  ↪ protocole TCP (Transmission Control Protocol).

Nous n'utiliserons que le mode connecté, TCP/IP !

Avec TCP, il faut établir puis terminer la connexion ; deux acteurs :

- Le serveur qui attend une demande connexion et accepte le client.
- Le client qui est à l'origine de la demande de connexion.

↪ modèle client-serveur.

Chacun à des actions à faire :

- Le client doit contacter le processus serveur dont il connaît le nom de machine (ou son adresse IP) et le port.
- Le serveur doit faire deux choses :
  - se mettre en attente sur un certain port et certaines adresses IP ;
  - quand un client se connecte, dialoguer avec lui.

Un serveur peut accepter des connexions de plusieurs clients. Par exemple, un serveur web écoute sur le port 80 et répond à plein de clients.

Question : un serveur est en contact avec plusieurs clients. Lorsqu'il reçoit des données, comment reconnaît-il le client qui les envoie ?

Plan

1. Introduction
   - Problématique
   - Quelques notions sur les réseaux
2. Les sockets
   - Définition
   - Mise en place côté serveur
   - Mise en place côté client
   - Échange de données
   - En résumé
3. Transferts de données avec les sockets
   - Position du problème
   - Quelques pièges classiques
   - Comment s'y prendre ?
4. Conclusion

1 Introduction
- Problématique
- Quelques notions sur les réseaux

2 Les sockets
- Définition
- Mise en place côté serveur
- Mise en place côté client
- Échange de données
- En résumé

3 Transferts de données avec les sockets
- Position du problème
- Quelques pièges classiques
- Comment s'y prendre ?

4 Conclusion

Adresse IP et numéro de port

Adresse IP

- L'adresse IP sert à identifier une machine
- On utilise 127.0.0.1 (nom `localhost`) lorsqu'il s'agit de la machine locale

Numéro de port

- Codés sur 16 bits
- Les numéros de ports de 0 à 1 023 sont utilisés pour les services réseaux les plus courants. Ex : 25 pour SMTP, 80 pour HTTP

On identifie une communication réseau par un couple (adresse IP, numéro port).

Socket

L'outil central de la communication réseau est la socket.

Définition (Socket)

Tout comme un tube, une socket permet de définir un canal de communication entre deux processus, mais :

- elle est bidirectionnel ;
- elle permet l'utilisation du réseau ;
- elle permet de choisir différents protocoles.

En pratique, une socket est un descripteur de fichier de type `int`.

- Le serveur doit faire deux choses :
  - mettre en place une socket d'écoute, qui se met en attente sur un certain port et sur certaines adresses IP de l'ordinateur ;
  - créer une socket de dialogue avec chaque client qui se connecte.
- Un client contacte le processus serveur dont il connaît l'adresse IP et le port.

Mise en place côté serveur

Il faut une socket qui ne sert que pour être contactée par le(s) client(s).

- Les paramètres : le port et les adresses possibles (par défaut, toutes).
- Le résultat est une socket d'écoute, elle ne sert pas à communiquer.

En C POSIX, il plusieurs fonctions doivent être utilisées à la suite :

- `int getaddrinfo(const char *node, const char *service, ...)`
  ↪ elle transforme les adresses de machines et les services en une liste de structures utilisables par d'autres fonctions de l'API.
- `int socket(int domain, int type, int protocol)`
  ↪ crée le descripteur de fichier.
- `int bind(int s, const struct sockaddr *addr, socklen_t len)`
  ↪ réserve un port réseau de la machine.
- `int listen(int s, int bl)`
  ↪ met en place la socket d'écoute.

Le serveur va ensuite obtenir une nouvelle socket de dialogue pour chaque nouveau client qui viendra se connecter :

- les données envoyées par le client seront lues depuis cette socket ;
- les données écrites sur la socket seront envoyées à ce client particulier.

En C POSIX, le serveur accepte les demandes de connexion avec :

```
int accept(int s, struct sockaddr *addr, socklen_t *len);
```

- `s` est la socket d'écoute.
- retourne la socket de dialogue ou -1 en cas d'erreur.
- `addr` et `len` (résultats) : l'adresse du client.

L'appel `sd = accept(s, ...)` est bloquant : le processus ne reprend que lorsqu'un client est connecté, et alors `sd` permet le dialogue.

Mise en place côté client

Le client doit se connecter à un port du serveur et lui demander de créer une socket de dialogue.

- Si le port est fermé, ou qu'aucun processus n'est à l'écoute sur ce port, la requête sera refusée par le système de la machine contactée.
- Les paramètres nécessaires sont le nom et le port du serveur.
- Le résultat est une socket de dialogue.

En C POSIX, il faut encore plusieurs fonctions :

- `int getaddrinfo(...)`
- `int socket(int domain, int type, int protocol)`
  ↪ pour créer la socket.
- `int connect(int s, const struct sockaddr *adr, socklen_t l)`
  ↪ pour se connecter au serveur, et obtenir la socket de dialogue.

Échange de données

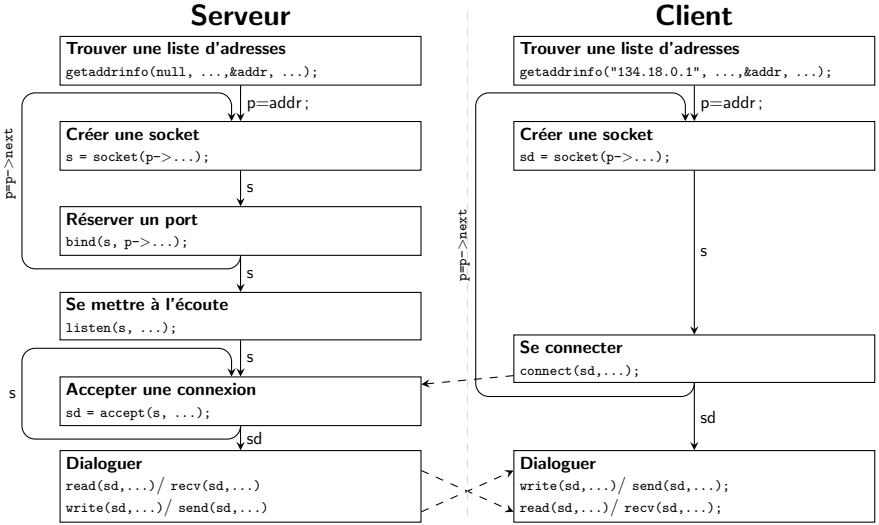
Une fois la connexion établie, les processus peuvent échanger des données grâce à `recv()` et `send()` :

- `ssize_t recv(int sd, void *buf, ssize_t len, int flags);`
- `ssize_t send(int sd, const void *buf, size_t len, int flags);`
  - `sd` : la socket de dialogue,
  - `buf` : les données, au plus de taille `len`,
  - `flags` : des options utiles pour le réseau.

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

Comme avec `read()` et `write()`, les fonctions `recv()` et `send()` retournent, en l'absence d'erreur, le nombre d'octets effectivement lus ou écrits.

En résumé


Serveur
Trouver une liste d'adresses
getaddrinfo(null, ...,&addr, ...);
p=addr;
p=p->next
Créer une socket
s = socket(p->...);
s
Réserver un port
bind(s, p->...);
s
Se mettre à l'écoute
listen(s, ...);
s
s
Accepter une connexion
sd = accept(s, ...);
sd
Dialoguer
read(sd,...)/ recv(sd,...)
write(sd,...)/ send(sd,...)
Client
Trouver une liste d'adresses
getaddrinfo("134.18.0.1", ...,&addr, ...);
p=addr;
p=p->next
Créer une socket
sd = socket(p->...);
s
Se connecter
connect(sd,...);
sd
Dialoguer
write(sd,...)/ send(sd,...);
read(sd,...)/ recv(sd,...);

L'API (interface) POSIX des sockets a peu changé depuis sa création. Cela indique sa souplesse mais explique aussi sa difficulté d'utilisation :

- Un simple serveur en C demande 120 lignes de code...
- Il faut faire attention à la compatibilité avec IPv6.
- Certaines fonctions n'existent que pour des raisons historiques.

Pour simplifier, nous utiliserons en TP une bibliothèque *ad hoc* avec :

- `int create_server_socket(const char* port);`
  ↪ retourne une socket d'écoute sur le port passé en paramètre.
- `int accept_connection(int s);`
  ↪ bloque le serveur jusqu'à la connexion d'un client,
  et retourne alors une socket de dialogue.
- `int create_client_socket(const char *host, const char* port);`
  ↪ se connecte à un serveur et retourne une socket de dialogue.

Le code minimal d'un client-serveur avec la bibliothèque des TP :

- le serveur :

```
char msg[] = "coucou :)";
int s = create_server_socket("9999"); // création de la socket d'écoute
int sd = accept_connection(s); // attente de la connexion d'un client
close(s); // fermeture de la socket d'écoute
send(sd, msg, strlen(msg), 0); // envoi d'un message au client
close(sd); // fermeture de la socket de dialogue
```

- le client :

```
char msg[64];
int sd = create_client_socket("localhost", "9999"); // connexion
int rd = recv(sd, msg, 64, 0); // réception d'un message
msg[rd] = '\0';
std::cout << "Message : " << msg << std::endl; // affichage
close(sd); // fermeture de la socket de dialogue
```

1 Introduction
- Problématique
- Quelques notions sur les réseaux

2 Les sockets
- Définition
- Mise en place côté serveur
- Mise en place côté client
- Échange de données
- En résumé

3 Transferts de données avec les sockets
- Position du problème
- Quelques pièges classiques
- Comment s'y prendre ?

4 Conclusion

Position du problème

Une grosse part des difficultés dans l'utilisation des sockets vient du transfert de données. Il n'y a pas de méthode applicable dans tous les cas :

- transfert de texte, données numériques, données binaires, ...
- données plus au moins importantes (messages, page web, ...)

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

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

On va utiliser en TD/TP les primitives POSIX :

- `recv()` pour lire des octets.
- `send()` pour écrire des octets.

Routage dans les réseaux (LIFRES l'an prochain)

Lorsque l'on souhaite échanger des messages à travers un réseau, une étape fondamentale est le routage.

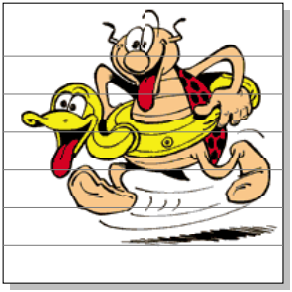
Routage

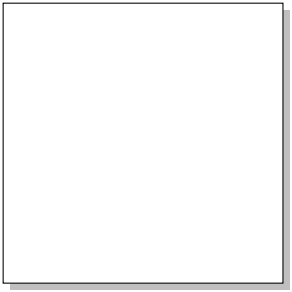
- Analogie avec une carte routière : plusieurs chemins pour communiquer avec une même destination.
- Un message est découpé en un ensembles de paquets IP, dont la taille dépend de caractéristiques du réseau.
- Chaque paquet peu emprunter une route différente.

Les paquets peuvent emprunter des routes diverses, avec des caractéristiques variées (vitesse, congestion du réseau). Ils peuvent donc arriver dans le désordre. L'utilisation de TCP garantie que les paquets seront fournis dans l'ordre d'envoi aux couches supérieures.

Exemple d'un transfert d'image : quand le réseau est peu chargé.

Exemple d'un transfert d'image : quand le réseau est peu chargé.

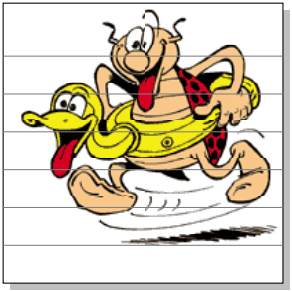


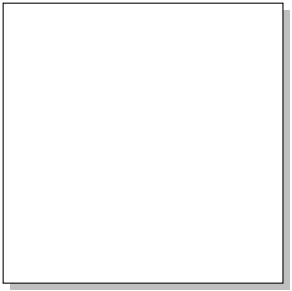


```
for(i=0; i<TAILLE_IM/TAILLE_MSG; i++) {
  send(s, &image[i*TAILLE_MSG], TAILLE_MSG, 0);
}
```

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

Exemple d'un transfert d'image : quand le réseau est peu chargé.

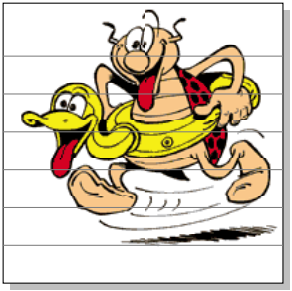


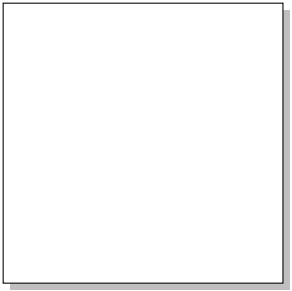


```
for(i=0; i<TAILLE_IM/TAILLE_MSG; i++) {
  send(s, &image[i*TAILLE_MSG], TAILLE_MSG, 0);
}
```

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

Exemple d'un transfert d'image : quand le réseau est peu chargé.

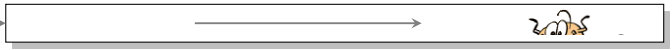




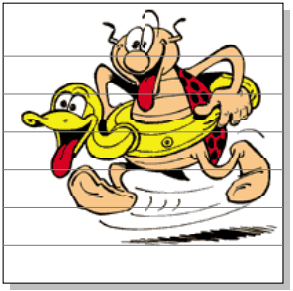
```
for(i=0; i<TAILLE_IM/TAILLE_MSG; i++) {
  send(s, &image[i*TAILLE_MSG], TAILLE_MSG, 0);
}
```

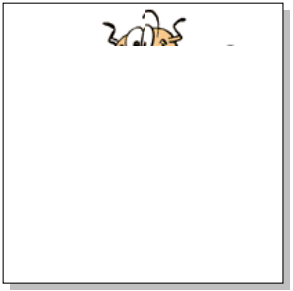
```
for(i=0; i<TAILLE_IM/TAILLE_MSG; i++) {
  recv(s, &image[i*TAILLE_MSG], TAILLE_MSG, 0);
}
```

Le réseau est peu chargé, les paquets arrivent facilement



Exemple d'un transfert d'image : quand le réseau est peu chargé.

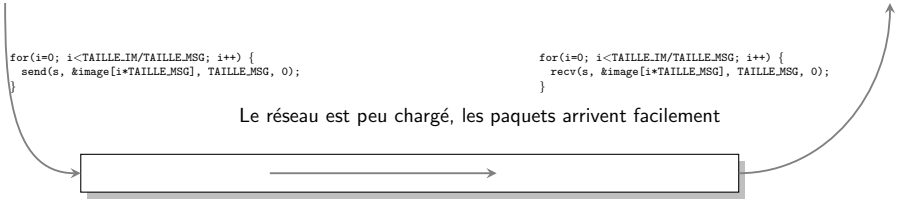




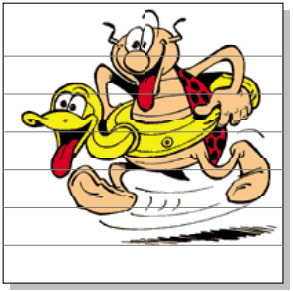
```
for(i=0; i<TAILLE_IM/TAILLE_MSG; i++) {
  send(s, &image[i*TAILLE_MSG], TAILLE_MSG, 0);
}
```

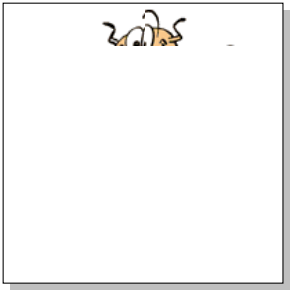
```
for(i=0; i<TAILLE_IM/TAILLE_MSG; i++) {
  recv(s, &image[i*TAILLE_MSG], TAILLE_MSG, 0);
}
```

Le réseau est peu chargé, les paquets arrivent facilement



Exemple d'un transfert d'image : quand le réseau est peu chargé.

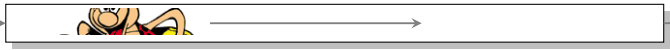




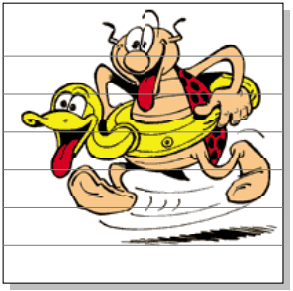
```
for(i=0; i<TAILLE_IM/TAILLE_MSG; i++) {
  send(s, &image[i*TAILLE_MSG], TAILLE_MSG, 0);
}
```

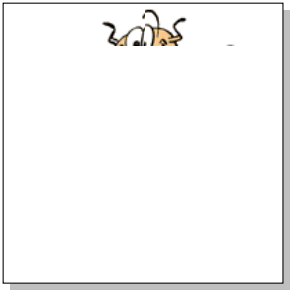
```
for(i=0; i<TAILLE_IM/TAILLE_MSG; i++) {
  recv(s, &image[i*TAILLE_MSG], TAILLE_MSG, 0);
}
```

Le réseau est peu chargé, les paquets arrivent facilement



Exemple d'un transfert d'image : quand le réseau est peu chargé.

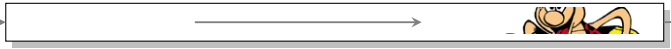




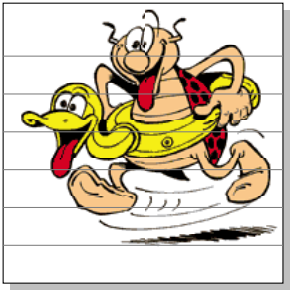
```
for(i=0; i<TAILLE_IM/TAILLE_MSG; i++) {
  send(s, &image[i*TAILLE_MSG], TAILLE_MSG, 0);
}
```

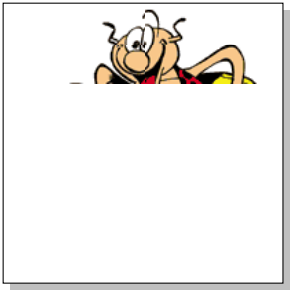
```
for(i=0; i<TAILLE_IM/TAILLE_MSG; i++) {
  recv(s, &image[i*TAILLE_MSG], TAILLE_MSG, 0);
}
```

Le réseau est peu chargé, les paquets arrivent facilement



Exemple d'un transfert d'image : quand le réseau est peu chargé.

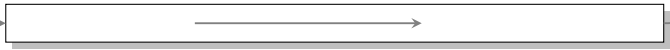




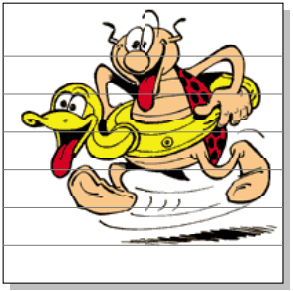
```
for(i=0; i<TAILLE_IM/TAILLE_MSG; i++) {
  send(s, &image[i*TAILLE_MSG], TAILLE_MSG, 0);
}
```

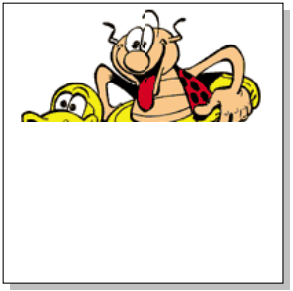
```
for(i=0; i<TAILLE_IM/TAILLE_MSG; i++) {
  recv(s, &image[i*TAILLE_MSG], TAILLE_MSG, 0);
}
```

Le réseau est peu chargé, les paquets arrivent facilement



Exemple d'un transfert d'image : quand le réseau est peu chargé.

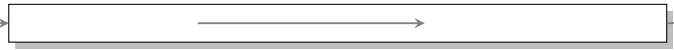




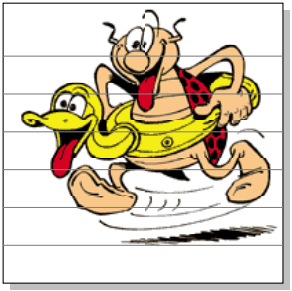
```
for(i=0; i<TAILLE_IM/TAILLE_MSG; i++) {
  send(s, &image[i*TAILLE_MSG], TAILLE_MSG, 0);
}
```

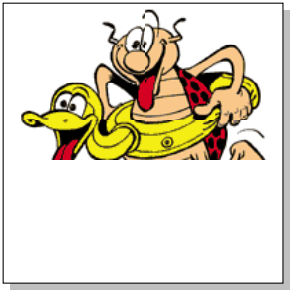
```
for(i=0; i<TAILLE_IM/TAILLE_MSG; i++) {
  recv(s, &image[i*TAILLE_MSG], TAILLE_MSG, 0);
}
```

Le réseau est peu chargé, les paquets arrivent facilement



Exemple d'un transfert d'image : quand le réseau est peu chargé.

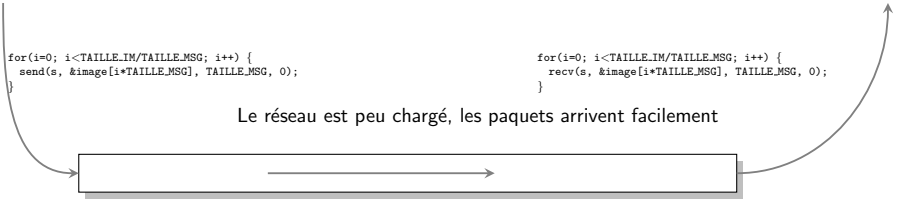




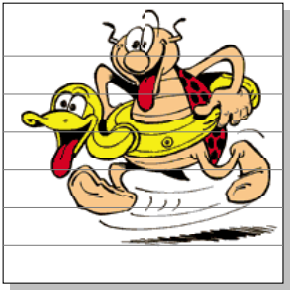
```
for(i=0; i<TAILLE_IM/TAILLE_MSG; i++) {
  send(s, &image[i*TAILLE_MSG], TAILLE_MSG, 0);
}
```

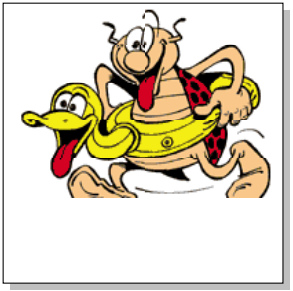
```
for(i=0; i<TAILLE_IM/TAILLE_MSG; i++) {
  recv(s, &image[i*TAILLE_MSG], TAILLE_MSG, 0);
}
```

Le réseau est peu chargé, les paquets arrivent facilement



Exemple d'un transfert d'image : quand le réseau est peu chargé.

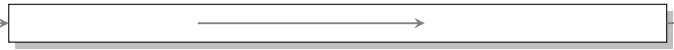




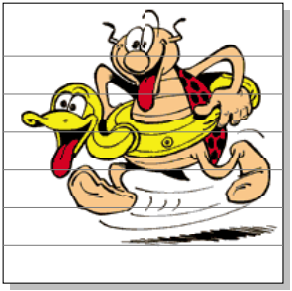
```
for(i=0; i<TAILLE_IM/TAILLE_MSG; i++) {
  send(s, &image[i*TAILLE_MSG], TAILLE_MSG, 0);
}
```

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

Le réseau est peu chargé, les paquets arrivent facilement



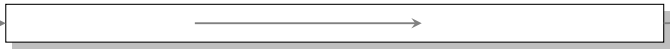
Exemple d'un transfert d'image : quand le réseau est peu chargé.



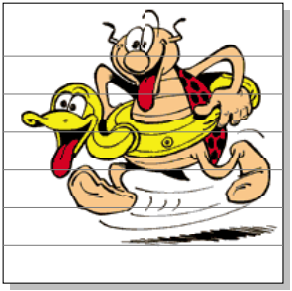
```
for(i=0; i<TAILLE_IM/TAILLE_MSG; i++) {
  send(s, &image[i*TAILLE_MSG], TAILLE_MSG, 0);
}
```

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

Le réseau est peu chargé, les paquets arrivent facilement



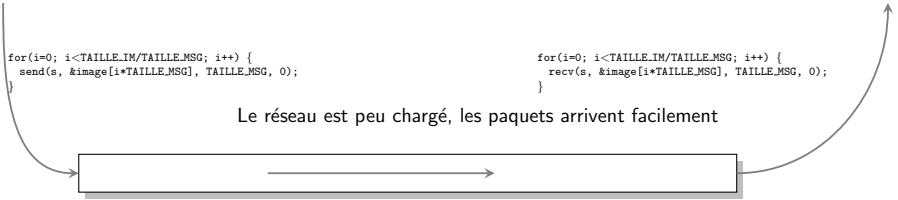
Exemple d'un transfert d'image : quand le réseau est peu chargé.



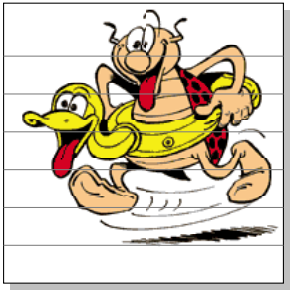
```
for(i=0; i<TAILLE_IM/TAILLE_MSG; i++) {
  send(s, &image[i*TAILLE_MSG], TAILLE_MSG, 0);
}
```

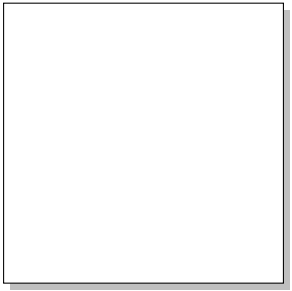
```
for(i=0; i<TAILLE_IM/TAILLE_MSG; i++) {
  recv(s, &image[i*TAILLE_MSG], TAILLE_MSG, 0);
}
```

Le réseau est peu chargé, les paquets arrivent facilement



Exemple d'un transfert d'image : quand le réseau est très chargé.

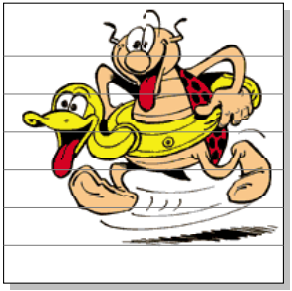


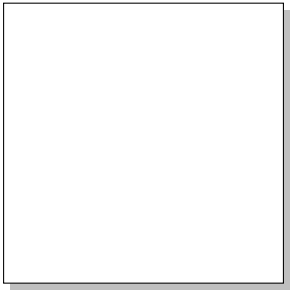


```
for(i=0; i<TAILLE_IM/TAILLE_MSG; i++) {
  send(s, &image[i*TAILLE_MSG], TAILLE_MSG, 0);
}
```

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

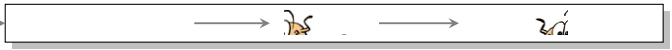
Exemple d'un transfert d'image : quand le réseau est très chargé.



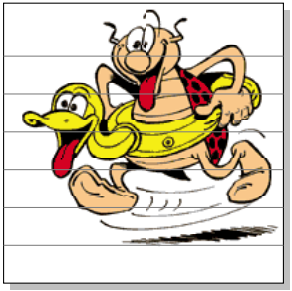


```
for(i=0; i<TAILLE_IM/TAILLE_MSG; i++) {
  send(s, &image[i*TAILLE_MSG], TAILLE_MSG, 0);
}
```

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

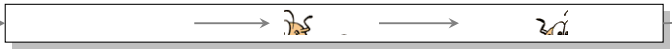


Exemple d'un transfert d'image : quand le réseau est très chargé.

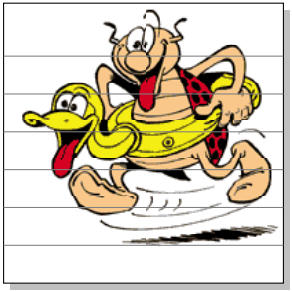


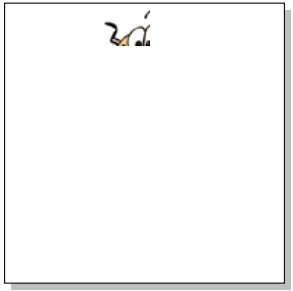
```
for(i=0; i<TAILLE_IM/TAILLE_MSG; i++) {
  send(s, &image[i*TAILLE_MSG], TAILLE_MSG, 0);
}
```

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



Exemple d'un transfert d'image : quand le réseau est très chargé.



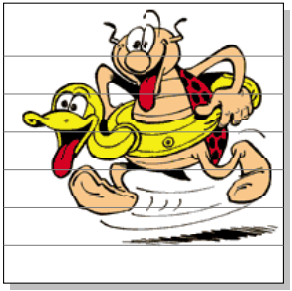


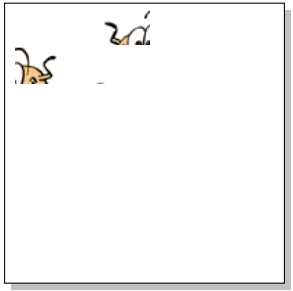
```
for(i=0; i<TAILLE_IM/TAILLE_MSG; i++) {
  send(s, &image[i*TAILLE_MSG], TAILLE_MSG, 0);
}
```

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

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

Exemple d'un transfert d'image : quand le réseau est très chargé.



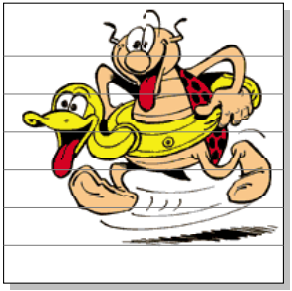


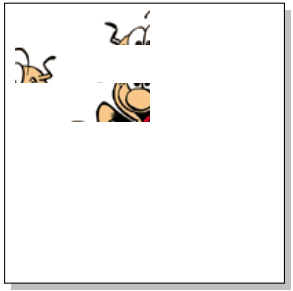
```
for(i=0; i<TAILLE_IM/TAILLE_MSG; i++) {
  send(s, &image[i*TAILLE_MSG], TAILLE_MSG, 0);
}
```

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

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

Exemple d'un transfert d'image : quand le réseau est très chargé.



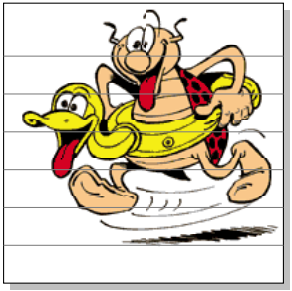


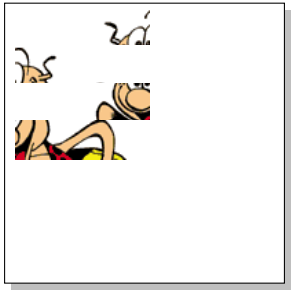
```
for(i=0; i<TAILLE_IM/TAILLE_MSG; i++) {
  send(s, &image[i*TAILLE_MSG], TAILLE_MSG, 0);
}
```

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

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

Exemple d'un transfert d'image : quand le réseau est très chargé.



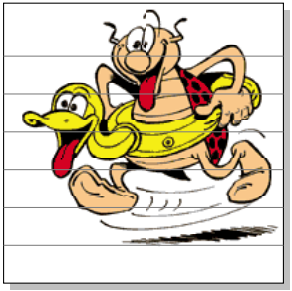


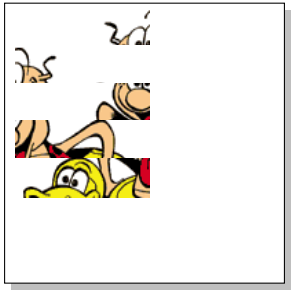
```
for(i=0; i<TAILLE_IM/TAILLE_MSG; i++) {
  send(s, &image[i*TAILLE_MSG], TAILLE_MSG, 0);
}
```

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

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

Exemple d'un transfert d'image : quand le réseau est très chargé.



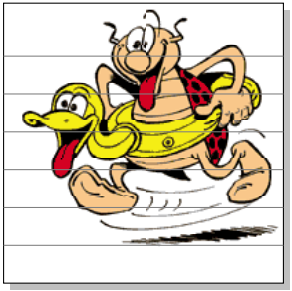


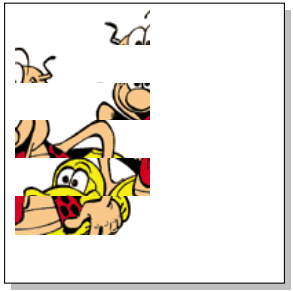
```
for(i=0; i<TAILLE_IM/TAILLE_MSG; i++) {
  send(s, &image[i*TAILLE_MSG], TAILLE_MSG, 0);
}
```

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

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

Exemple d'un transfert d'image : quand le réseau est très chargé.



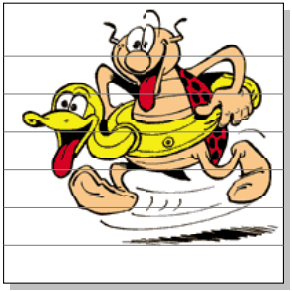


```
for(i=0; i<TAILLE_IM/TAILLE_MSG; i++) {
  send(s, &image[i*TAILLE_MSG], TAILLE_MSG, 0);
}
```

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

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

Exemple d'un transfert d'image : quand le réseau est très chargé.



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

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

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

Le programme précédent est buggé, mais le bug

- n'est visible que sur de gros transferts, jamais lors de tests simples ;
- n'apparaît que dans des conditions de « stress » du réseau.

Messages

Le transfert d'un 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.

Il faut définir un protocole pour retrouver les frontières des messages.

Pour réussir un échange de données, il faut :

- implémenter des primitives robustes, par exemple
  `vector<char> read_all(int fd, int n)` pour lire exactement `n` octets.
- établir un protocole de communication dans lequel chaque processus
  - sait quand il doit lire ou écrire un message.
  - sait retrouver les frontières entre les messages.

Exemple : 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 accompagnée de lignes d'entête. Par exemple :

```
GET / HTTP/1.1
Host: www.example.org
Connection: Close
```

  Chaque ligne fini par `CRLF`, et la requête se termine par deux `CRLF`.
- Le serveur répond avec une entête formatée de la même manière et qui indique la taille de la page à lire en octets, puis il envoie la page.

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

Quelques pièges classiques

1er exemple : un serveur attend une connexion, lit par blocs d'au plus 13 caractères sur la socket de dialogue, et affiche ceux qui sont lus :

```
while(1) {
    char buf[13];
    res = recv(sd, buf, 13, 0);
    if(res == 0) break; // reception de EOF
    if(res < 0) exit_error("recv");
    printf("Lecture de %d octets : *%s*\n", res, buf);
}
```

Un client tente d'envoyer une chaîne :

```
nlouvet:~/$ echo -n "toto fait du velo" | nc localhost 8083 -N
```

Le serveur affiche :

```
Lecture de 13 octets : *toto fait du *
Lecture de 4 octets : *velo fait du *
```

2ème exemple : Dans le 1er exemple, on ne plaçait pas de '\0' en fin de la chaîne à afficher, d'où l'erreur. Corrigeons :

```
#define LEN 13
while(1) {
    char buf[LEN];
    res = recv(sd, buf, LEN-1, 0);
    if(res == 0) break; // reception de EOF
    if(res < 0) exit_error("read");
    buf[res] = '\0';
    printf("Lecture de %d octets : *%s*\n", res, buf);
}
```

Un client envoye une chaîne :

```
nlouvet:~/$ echo -n "toto fait du velo" | nc localhost 8083
```

Le serveur affiche :

```
Lecture de 12 octets : *toto fait du*
Lecture de 5 octets : * velo*
```

C'est déjà plus satisfaisant, mais pas forcément parfait !

On veut envoyer le fichier suivant au serveur :

```
nlouvet:~/$ cat test.txt
certains mots disparaissent dans les réseaux
```

On fait donc :

```
nlouvet:~/$ cat test.txt | nc localhost 8083 -N
```

Le serveur affiche :

```
Lecture de 12 octets : *certains mot*
Lecture de 12 octets : *s*              <- !!!
Lecture de 12 octets : *sent dans le*
Lecture de 11 octets : *s réseaux
*
```

Indication :

```
nlouvet:~/$ xxd test.txt
00000000: 6365 7274 6169 6e73 206d 6f74 7300 2064 certains mots. d
00000010: 6973 7061 7261 6973 7365 6e74 2064 616e isparaissent dan
00000020: 7320 6c65 7320 72c3 a973 6561 7578 0a s les r..seaux.
```

Un dernier pour la route : On veut échanger des messages de 4 caractères, donc on utilise l'option `MSG_WAITALL` de `recv()` pour s'assurer qu'on a bien tout reçu :

```
#define LEN 4
while(1) {
  char buf[LEN+1];
  res = recv(sd, buf, LEN, MSG_WAITALL);
  if(res == 0) break;
  if(res < 0) exit_error("read");
  buf[res] = '\0';
  printf("Lecture de %d octets : *%s*\n", res, buf);
}
```

Le client envoye :

```
nlouvet:~/$ echo -n "héhé" | nc localhost 8083
```

Le serveur affiche :

```
Lecture de 4 octets : *héh*
Lecture de 2 octets : *é*
```

C'est encore une histoire d'encodage des caractères !

Comment s'y prendre ?

Grâce à TCP, vous savez que tous les octets arrivent dans le bon ordre, sinon une erreur est détectée. Mais vous ne savez pas :

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

Il est nécessaire de définir un protocole de communication :

- savoir quand lire et quand écrire,
- savoir comment détecter la fin d'un message,
- savoir quelles informations sont échangées

1. Introduction
   - Problématique
   - Quelques notions sur les réseaux
2. Les sockets
   - Définition
   - Mise en place côté serveur
   - Mise en place côté client
   - Échange de données
   - En résumé
3. Transferts de données avec les sockets
   - Position du problème
   - Quelques pièges classiques
   - Comment s'y prendre ?
4. Conclusion

Conclusion

Moyens de communication entres processus

Nous avons vu :

- les fichiers réguliers,
- les signaux (messages simples),
- les tubes anonymes ou nommés (entre processus d'un même système),
- les sockets (échanges via le réseau).

Nous n'avons pas parlé des systèmes de fichier réseau (comme NFS), des segments mémoire partagés, ou des Remote Procedure Calls (RPC)...

Difficultés

- Choisir la structure d'un fichier (fichiers réguliers).
- Définir un protocole de communication (tubes ou sockets).