M1 Informatique - Systèmes Répartis

Séance 0 - systèmes et communication : rappels, fondamentaux, appels systèmes

Benoît Darties

Université de Montpellier

5 octobre 2025

Plan

Plan

Programme vs Processus

Programme

- Ensemble statique d'instructions (fichier exécutable).
- Ne consomme pas de ressources tant qu'il n'est pas lancé.

Processus

- Instance en cours d'exécution d'un programme.
- Possède un état dynamique (mémoire, CPU, liste de fichiers ouverts).

Rappel : Définition d'un processus

Définition

Un **processus** est un programme en cours d'exécution, disposant d'un espace mémoire propre, d'un `compteur ordinal` et d'un ensemble de ressources.

Caractéristiques

- Isolement : chaque processus a son **espace d'adressage** distinct.
- État : représenté par le contenu des registres, de la pile, et des variables.
- Communication : nécessite des mécanismes explicites (IPC, messages, **sockets**).

Idée clé

Un processus exécute en continu des **fonctions utilisateur**, mais lorsqu'il veut interagir avec le système (fichier, réseau, processus), il doit passer par des **appels système**.

Fonctions vs Appels Systèmes

Fonctions : page 3 du manuel

- Exécutées en **espace utilisateur**.
- Fournies par la bibliothèque standard (libc, glibc), ce sont des suites d'instructions ordinaires, manipulant uniquement l'espace mémoire du processus.
- Exemple : `printf()`, `strlen()`, `malloc()`.

Appels système : page 2 du manuel

- Exécutés en **espace noyau**, via une transition contrôlée (*trap*, interruption logicielle).
- Fournissent l'accès aux ressources matérielles et aux services du noyau (fichiers, réseau, processus).
- Impliquent un changement de contexte : du mode utilisateur vers le mode noyau.
- Exemple : `read()`, `write()`, `fork()`, `_exit()`.

Pourquoi distinguer fonctions et appels système ?

Idée clé

Les appels système garantissent la sécurité et l'intégrité du système, tandis que les **fonctions facilitent la programmation et la portabilité**. Complémentarité indispensable.

Sécurité et stabilité

- Le noyau protège l'accès aux ressources critiques (mémoire, disque, réseau).
- Seuls les **appels système** passent en **mode noyau** : le propriétaire du processus appelant doit avoir les droits d'accès aux ressources ciblées par l'appel système. Ainsi ceci empêche un processus utilisateur de corrompre le système ou d'accéder directement au matériel.

Efficacité et simplicité

- Les **fonctions de bibliothèque** encapsulent souvent des appels système, en ajoutant confort et portabilité (`fopen()` au lieu de `open()`).
- Elles permettent de réutiliser du code en espace utilisateur, sans passer inutilement en

Processus dans le système et hiérarchie

Identification

- Chaque processus possède un **PID** (Process ID).
- Consultation : commande `ps`, `top`, ou `pidof`.

Relation de parenté

- Chaque processus est rattaché à un processus parent **PPID** (Parent PID).
- Les processus forment un **arbre** enraciné (commande `pstree`).
- Le premier processus `init / systemd` (PID 1) est créé par le noyau.
- La terminaison d'un parent entraîne le rattachement des fils orphelins à `init / systemd`

Appels systèmes

- `getpid()` retourne le PID du processus courant.
- `getppid()` retourne le PID du parent.

Rappel : Espace d'adressage

Définition

L'**espace d'adressage** d'un processus est l'ensemble des adresses mémoire (logiques/virtuelles) auxquelles il peut accéder.

Organisation typique

- Chaque processus a son espace d'adressage isolé des autres (protection mémoire).
- **Code** : instructions du programme.
- **Données statiques** : variables globales, constantes.
- **Tas (heap)** : mémoire dynamique (malloc/new).
- **Pile (stack)** : variables locales, appels de fonctions.

Rappel : Compteur ordinal (Program Counter)

Définition

Le **compteur ordinal** (ou *Program Counter, PC*) est un registre matériel qui contient l'adresse mémoire de la **prochaine instruction** à exécuter par le processeur.

Fonctionnement

- Après chaque instruction, le PC est automatiquement incrémenté pour pointer vers l'instruction suivante.
- En cas de `jmp`, `call`, `return`, le PC est mis à jour avec une nouvelle adresse.
- Permet au processeur de suivre le flux séquentiel du programme.

Caractéristiques

Chaque processus possède son propre PC, sauvegardé/restauré lors d'un **changement de contexte**, indispensable pour assurer l'illusion d'une exécution parallèle de plusieurs processus.

Création et recouvrement de processus

Création

- Fait par un processus parent via un appel système (`fork` sous Unix).
- Le processus fils hérite d'une copie de l'espace mémoire du parent.

Recouvrement (replacement)

- Le processus courant charge un nouveau programme en mémoire via `execve()`.
- Change le code exécuté mais conserve PID et ressources.

Appel système : `fork()`

Prototype

```
pid_t fork(void);
```

Rôle

Crée un nouveau processus fils, copie conforme du parent, avec un PID différent.

Comportement

- Retourne deux valeurs : `0` dans le fils, `PID fils` dans le parent.
- Permet l'exécution concurrente de code dans deux branches.

Appel système : execve()

Prototype execve()

```
int execve(const char *pathname, char *const argv[], char *const envp[]);
```

Rôle

Remplace le code et les données du processus courant par un nouveau programme.

Comportement

- Ne crée pas de nouveau processus, réutilise le PID courant.
- Plusieurs variantes (fonctions) : `execl`, `execv`, `execvp`, etc.

Appel système : `wait()`

Prototype

```
pid_t wait(int *wstatus);
```

Rôle

Permet à un processus parent d'attendre la terminaison d'un fils.

Comportement

- Retourne le PID du fils terminé.
- Évite les processus zombies en récupérant le code de sortie.

Appel système : `_exit()`

Prototype

```
void _exit(int status);
```

Rôle

Termine un processus proprement et renvoie un code d'état à son parent.

Comportement

- Libère les ressources associées (mémoire, fichiers ouverts).
- L'état de sortie est récupérable par `wait()` du parent.

Cycle de vie d'un processus : résumé

Création

`fork()` : le processus parent duplique son espace d'adressage et crée un processus fils avec un nouveau PID.

Recouvrement

`execve()` : le processus (parent ou fils) charge en mémoire un nouveau programme et remplace son code courant.

Attente

`wait()` : le parent se bloque jusqu'à la fin d'un processus fils, récupère son code de retour et évite la création de zombies.

Terminaison

`_exit()` : le processus libère ses ressources et renvoie un code d'état à son parent.

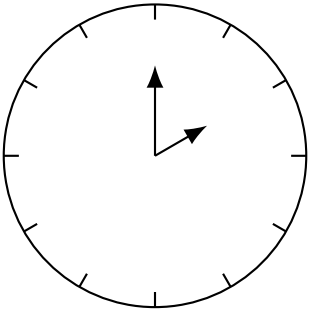
Exemple : création d'un fils et attente

Code C

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main(void) {
    pid_t pid = fork();
    if (pid == 0) {   // Code du fils
        printf("Je suis le fils, PID=%d\n", getpid());
        _exit(0); // terminer proprement
    } else if (pid > 0) {     // Code du parent
        int status;
        wait(&status); // attendre le fils
        printf("Fils terminé avec code %d\n", WEXITSTATUS(status));
    } else { // erreur : echec de la duplication
        perror("fork");
    }
    return 0;
}
```

Horloge commune et communication inter-processus



Message clé

Sur une **même machine**, tous les processus partagent **une horloge commune**.

Conséquences

Avec horloge commune

- ordonnancement global cohérent, plus simple de mesurer les temps (profilage, deadlines).
- l'ordre d'envoi / réception des messages par un processus est connu

Sans horloge commune

- pas de temps universel, difficultés pour synchroniser les événements
- nécessité d'horloges logiques (Lamport, vectorielles).
- Impact direct sur la **communication inter-processus** : il faut expliciter les causalités et utiliser des protocoles de synchronisation.
- cas typique des **systèmes répartis**

Plan

Contextualisation : panorama des communications

Deux grandes familles

- **IPC locales** : mécanismes inter-processus sur une même machine.
- **IPC distantes** : communication par réseau.

Exemples

- IPC locales : signaux, tubes, mémoire partagée, files de messages.
- IPC distantes : sockets (TCP/UDP)

Signaux

Principe

Les **signaux** sont des interruptions logicielles envoyées à un processus pour lui notifier un événement.

Exemples

- `SIGKILL`, `SIGTERM`, `SIGINT`, `SIGHUP`, `SIGCHLD`.
- Par défaut, chaque signal a une action prédéfinie (terminer, ignorer, stopper...).
- Un processus peut redéfinir un *gestionnaire* avec `signal()` ou `sigaction()`.

Tubes (pipes)

Principe

Mécanisme simple de communication **unidirectionnelle** entre deux processus apparentés.

Caractéristiques

- Appel système `pipe()` : crée un canal avec une extrémité lecture et une écriture.
- Héritage via `fork()` : communication parent–fils.
- Communication locale seulement.

Extension

FIFO (named pipe) : accessible à des processus non apparentés (créée avec `mkfifo()`).

Mémoire partagée et files de messages

Mémoire partagée

- Zone mémoire accessible par plusieurs processus.
- Très performante, mais nécessite synchronisation stricte (sémaphores).
- API SysV : `shmget()`, `shmat()`, `shmdt()`.
- API POSIX : `shm_open()`, `mmap()`.

Files de messages

- File FIFO dans le noyau où les processus déposent/lisent des messages.
- API SysV : `msgget()`, `msgsnd()`, `msgrcv()`.
- API POSIX : `mq_open()`, `mq_send()`, `mq_receive()`.

Passage de messages via sockets

Principe

Les processus **échangent des messages** via l'API `socket()`, en local (`AF_UNIX`) ou à distance (`AF_INET/INET6`).

Exemples

- **TCP** (connecté, fiable, flux) et **UDP** (non connecté, datagramme, best-effort).
- Modèle client–serveur : distinctions des rôles

Forces / Limites

- + **Universel** (même modèle local/distant), interopérable, passe les frontières machines.
- − Overhead réseau, gestion d'erreurs/temps d'attente, sérialisation des données.

Comparaison des mécanismes IPC

Tableau de synthèse

- **Signaux** : simples, notification d'événement, pas de données complexes.
- **Tubes** : communication unidirectionnelle, locale, parent–fils.
- **Mémoire partagée** : très rapide, mais difficile à synchroniser.
- **Files de messages** : communication structurée, mais overhead plus élevé.
- **Sockets** : plus universelles, locales ou distantes, base de la programmation réseau.

Synthèse des mécanismes de communication inter-processus

Tableau comparatif

Mécanisme	Local / Distant	Complexité	Données	Cas d'usage
Signaux	Local	Très faible	Notification	Stopper/contrôler un processus, synchro mini.
Tubes (pipes/FIFO)	Local	Faible	Flux	Dialogue parent–fils, chaînes de commandes
Mémoire partagée	Local	Élevée	Structures	Partage rapide de gros volumes de données
Files de messages	Local	Moyenne	Messages	File d'attente, producteur/consommateur
Sockets	Local & Distant	Moyenne	Flux	Réseau, services client–serveur, Internet

Message clé

Les sockets offrent une API **universelle**, utilisable à la fois en local et en distant, ce qui en fait le mécanisme de base pour la **programmation réseau**.

Plan

Communication distante

Notion

Communication entre **deux** processus situés sur deux machines **différentes** (supposées différentes).

Identification

D'un point de vue extérieur à la machine, un processus est *identifié* par :

- l'adresse de la machine sur lequel il est exécuté (ie. adresse IP)
- un numéro de port de communication (0–65535) auquel il est rattaché : il y écoute les messages qui lui sont destinés

Définition de rôles

Serveur

- Fournit un service motivant la connexion du client.
- Rôle : répondre aux demandes.

Client

- Envoie des requêtes au serveur.
- Initialise le dialogue.

Éléments nécessaires

Protocole de communication

- Mode connecté (ie. TCP)
- Mode non connecté (ie. UDP)

Identification des processus impliqués

Du point de vue de l'émetteur d'un message :

- Adresse IP machine locale + port local
- Adresse IP machine distante + port distant

Modes de communication

Connecté (TCP)

- Envoi/réception synchronisés.
- Données acquittées.
- Phase préalable d'établissement de connexion.

Non connecté (UDP)

- Envoi/réception asynchrones.
- Pas d'acquittement, communication non garantie.
- Plus simple mais moins fiable.

Plan

la socket : support de communication

Définition

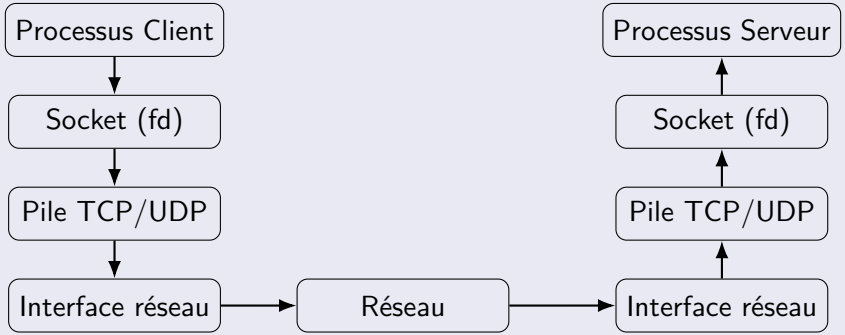
- Interface logicielle uniforme simplifiée exploitant les services d'un protocole réseau.
- descripteur sur lequel on va pouvoir :
  - lire des messages entrants,
  - envoyer des messages sortants

Avantages

- Abstraction des spécificités réseau.
- Facilité de programmation.
- Protocoles intégrés.
- Point d'entrée/sortie pour dialoguer avec d'autres processus.

Rôle d'une socket dans la communication inter-processus

La socket en tant qu'interface



Idée clé

Une socket est un **point d'entrée/sortie** permettant à un processus de dialoguer, localement ou à distance, via la pile protocolaire du noyau.

Gestion des sockets

- Une socket est identifiée par un descripteur (comme un fichier).
- Stocké dans la table des descripteurs du processus.
- Héritage en cas de `fork()`.
- Création par l'appel système : `socket()`.

Prototype de `socket()`

Prototype

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

Arguments

- **domain** : famille de protocoles
  - `PF_INET` : IPv4
  - `PF_INET6` : IPv6
  - `PF_UNIX` : communication locale (IPC)
- **type** : mode de communication
  - `SOCK_STREAM` : orienté connexion (TCP)
  - `SOCK_DGRAM` : datagramme (UDP)
- **protocol** : protocole précis (souvent 0 = par défaut).

valeur retour : Un **descripteur de fichier** (int > 0), ou -1 en cas d'erreur.

AF_INET vs PF_INET

Historique

- **PF_INET** : *Protocol Family*, utilisé pour indiquer une famille de protocoles.
- **AF_INET** : *Address Family*, utilisé pour indiquer une famille d'adresses.

Pratique

- Les deux constantes ont la **même valeur numérique**.
- `socket()` accepte souvent indifféremment `PF_INET` ou `AF_INET`.
- Dans les structures (ex. `sockaddr_in.sin_family`), il faut utiliser **AF_INET**.

Attribution d'une adresse

- Serveur : doit spécifier un numéro de port (et éventuellement une IP).
- Client : port choisi automatiquement, mais doit connaître l'adresse du serveur.

Rôle des structures `sockaddr`, `sockaddr_in` et `sockaddr_in6`

`struct sockaddr`

- Structure **générique**, utilisée par les différents appels systèmes impliqués dans la communication réseau) pour lire / stocker les paires adresses IP / port de communication
- Sert d'**enveloppe** : les fonctions réseau attendent un pointeur vers `sockaddr`, quel que soit le protocole.

`struct sockaddr_in` : version spécialisée pour IPv4

- Contient un champ `sin_port` (port TCP/UDP) et `sin_addr` (adresse IPv4 sur 32 bits).
- On la **cast** en `(struct sockaddr*)` lors des appels systèmes.

`struct sockaddr_in6` : version spécialisée pour IPv6.

- Contient `sin6_port` (port) et `sin6_addr` (adresse IPv6 sur 128 bits).
- Permet d'écrire du code compatible IPv6 de la même manière qu'avec IPv4.

Structure `struct sockaddr_in` (IPv4)

Définition

```
struct sockaddr_in {
    short           sin_family;   // famille d'adresses : AF_INET
    unsigned short  sin_port;     // numéro de port (Network Byte Order)
    struct in_addr  sin_addr;     // adresse IPv4 (32 bits)
    char            sin_zero[8];  // padding (non utilisé)
};

struct in_addr {
    unsigned long   s_addr;       // adresse IP (Network Byte Order)
};
```

Conversion d'adresses et de ports

Problème

Les machines n'utilisent pas toutes le même ordre des octets (endianness) :

- **little endian** (Intel) : octet de poids faible en premier.
- **big endian** (réseau) : octet de poids fort en premier.

Pour communiquer correctement, le réseau impose le **Network Byte Order** (big endian).

Fonctions de conversion

- `htons(uint16_t hostshort)` : convertit un entier court (16 bits, port) en format réseau.
- `htonl(uint32_t hostlong)` : convertit un entier long (32 bits, adresse IP) en format réseau.
- Fonctions inverses : `ntohs()`, `ntohl()`.

Exemple

Conversion d'adresses et de ports

Exemple

```
struct sockaddr_in serv;
serv.sin_port = htons(4444);           // port 4444 en ordre réseau
serv.sin_addr.s_addr = htonl(INADDR_ANY); // 0.0.0.0 en ordre réseau
```

Introduction à `bind()`

Pourquoi utiliser `bind()` ?

- Dans la communication réseau, chaque socket doit être identifiée par une **adresse locale** : **(adresse IP, numéro de port)**.
- L'appel système `bind()` permet de réaliser ce lien entre la socket créée par `socket()` et cette adresse locale.
- C'est l'équivalent de « dire au système : je veux écouter sur tel port ».
- appel présent en UDP mais aussi en TCP

Conséquences

- Côté **serveur** : indispensable, sinon impossible de recevoir des connexions.
- Côté **client** : facultatif, le système attribue automatiquement une adresse/port éphémère si non spécifié.

Associer une adresse locale : `bind()`

Prototype

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

Rôle

- Associe une socket à une adresse locale (IP + port).
- Obligatoire côté **serveur** pour annoncer le port d'écoute.
- Facultatif côté **client** (sinon le noyau choisit automatiquement un port éphémère).

Retour

- 0 si succès.
- −1 en cas d'erreur (par ex. port déjà utilisé, droits insuffisants).

Plan

Mode non connecté (UDP)

- Communication par datagrammes.
- Asynchrone, sans acquittement.
- Risque de perte de paquets.

Étapes d'une communication en UDP

Initialisation

- on crée une **socket** : interface de communication.
- On prépare une structure contenant notre adresse locale (IP + port) : **sockaddr**
- On associer la socket à cette adresse : `bind()`

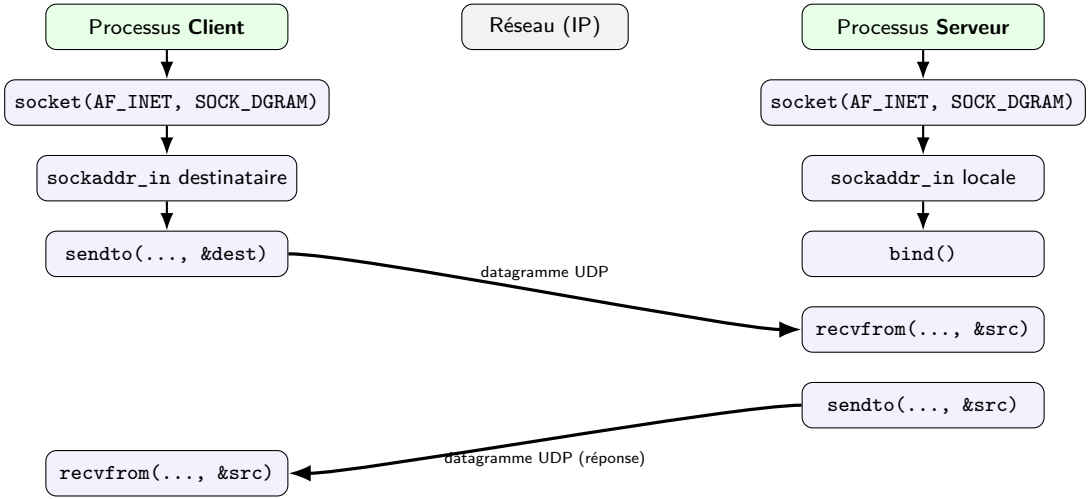
Pour l'envoi d'un message

- On prépare une structure `sockaddr` contenant l'adresse du destinataire.
- On envoye les données sur la socket, en fournissant la structure destinataire : `sendto()`

Pour la réception d'un message

- On prépare une `sockaddr` pour stocker l'adresse de l'émetteur et un buffer tampon.
- On se met en attente d'un message avec `recvfrom()` : à l'arrivée d'un message, les données reçues remplissent le tampon et la structure `sockaddr` de l'émetteur.

Schéma des étapes d'une communication **UDP**



Mode non connecté : `sendto()` et `recvfrom()`

Principe de fonctionnement

- En UDP (mode non connecté), il n'y a pas d'association préalable entre une socket et un correspondant unique.
- Chaque envoi ou réception doit préciser explicitement l'**adresse IP et le port** du destinataire ou de l'émetteur.
- C'est ce qui distingue UDP de TCP : pas de session, pas de notion de « flux » persistant.

Conséquences pratiques

- `sendto()` : l'adresse du destinataire doit être passée à chaque envoi.
- `recvfrom()` : l'adresse de l'expéditeur est récupérée à chaque réception.
- Flexible (communication avec plusieurs pairs) mais plus de travail côté application.

Envoi en mode non connecté : `sendto()`

Prototype

```
ssize_t sendto(int sockfd,
               const void *buf, size_t len,
               int flags,
               const struct sockaddr *dest_addr,  socklen_t addrlen);
```

Paramètres

- `sockfd` : descripteur de la socket (UDP).
- `buf`, `len` : tampon contenant les données et sa taille.
- `flags` : options d'envoi (souvent 0).
- `dest_addr`, `addrlen` : adresse et taille de la destination.

Retour : Nombre d'octets envoyés, ou `-1` en cas d'erreur.

Réception en mode non connecté : `recvfrom()`

Prototype

```
ssize_t recvfrom(int sockfd,
                 void *buf, size_t len,
                 int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
```

Paramètres

- `sockfd` : descripteur de la socket (UDP).
- `buf`, `len` : tampon de réception et taille max.
- `flags` : options (souvent 0).
- `src_addr`, `addrlen` : adresse source du client (remplie par `recvfrom()`)

Retour : Nombre d'octets reçus, ou `-1` en cas d'erreur.

Exemple serveur UDP perroquet

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main(void) {
  // 1) Création de la socket UDP
  int sock_fd = socket(PF_INET, SOCK_DGRAM, 0);

  // 2) Préparation des structures d'adresses
  struct sockaddr_in server_addr, client_addr;
  socklen_t addr_len = sizeof(struct sockaddr_in);
  char buffer[1024];

  // 3) Configuration de l'adresse locale
  server_addr.sin_family      = AF_INET;
  server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
  server_addr.sin_port        = htons(4444);
```

Exemple serveur UDP perroquet

```
22    // 4) Association socket / port
23    bind(sock_fd, (struct sockaddr*)&server_addr, addr_len);
24
25    // 5) Boucle de réception et d'écho
26    for (;;) {
27      int bytes_recv = recvfrom(sock_fd, buffer, sizeof(buffer), 0,
28              (struct sockaddr*)&client_addr, &addr_len);
29
30      // Réenvoi du message au client
31      sendto(sock_fd, buffer,  bytes_recv,  0,
32              (struct sockaddr*)&client_addr,  addr_len);
33    }
34
35    // (Jamais atteint ici : fermeture de la socket)
36    close(sock_fd);
37    return 0;
38  }
```

Exemple client UDP : envoi et réception

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main(void) {
  // 1) Création de la socket UDP
  int sock_fd = socket(PF_INET, SOCK_DGRAM, 0);

  // 2) Préparation de l'adresse du serveur
  struct sockaddr_in server_addr;
  socklen_t addr_len = sizeof(struct sockaddr_in);
  char buffer[1024];

  server_addr.sin_family      = AF_INET;
  server_addr.sin_port        = htons(4444);
  server_addr.sin_addr.s_addr = inet_addr("1.2.3.4");
```

Exemple client UDP : envoi et réception

```
21    // 3) Envoi d'un message au serveur
22    sendto(sock_fd, "Hello World", 11, 0, (struct sockaddr*)&server_addr, addr_len);
23
24    // 4) Réception de la réponse du serveur
25    int bytes_recv = recvfrom(sock_fd, buffer, sizeof(buffer), 0,
26                              (struct sockaddr*)&server_addr, &addr_len);
27
28    buffer[bytes_recv] = '\0';
29    printf("Recu : %s\n", buffer);
30
31    // 5) Fermeture de la socket
32    close(sock_fd);
33    return 0;
34 }
```

Plan

Mode connecté (TCP)

- Transfert des données en flot.
- Connexion préalable obligatoire.
- Données acquittées.
- Connexion fiable (pas de perte).

Étapes d'une communication en TCP (1/3)

Initialisation

- on crée une **socket** : interface de communication.
- On prépare une structure contenant notre adresse locale (IP + port) : **sockaddr**
- On associer la socket à cette adresse : `bind()`

Côté serveur

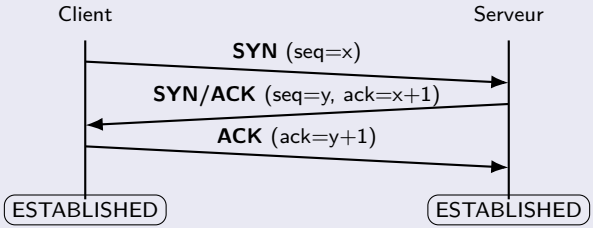
- On met la socket en attente de connexions avec `listen()`.
- Lorsqu'on accepte une connexion entrante avec `accept()`, on récupère une nouvelle socket dédiée à la communication avec ce client.

Côté client

- On prépare une structure `sockaddr` contenant l'adresse du serveur à contacter
- On établit la connexion avec `connect()`.

Étapes d'une communication en TCP (2/3) : *three-way handshake*

Séquence SYN → SYN/ACK → ACK



Conséquences

- Établit une **connexion fiable** (numéros de séquence, acquittements).
- **négociation** d'options : Taille de segments max MMS , Fenêtre de contention CW, etc.
- Après l'ACK final, les deux côtés passent à `ESTABLISHED` et échangent un **flux** bidirectionnel.

Étapes d'une communication en TCP (3/3)

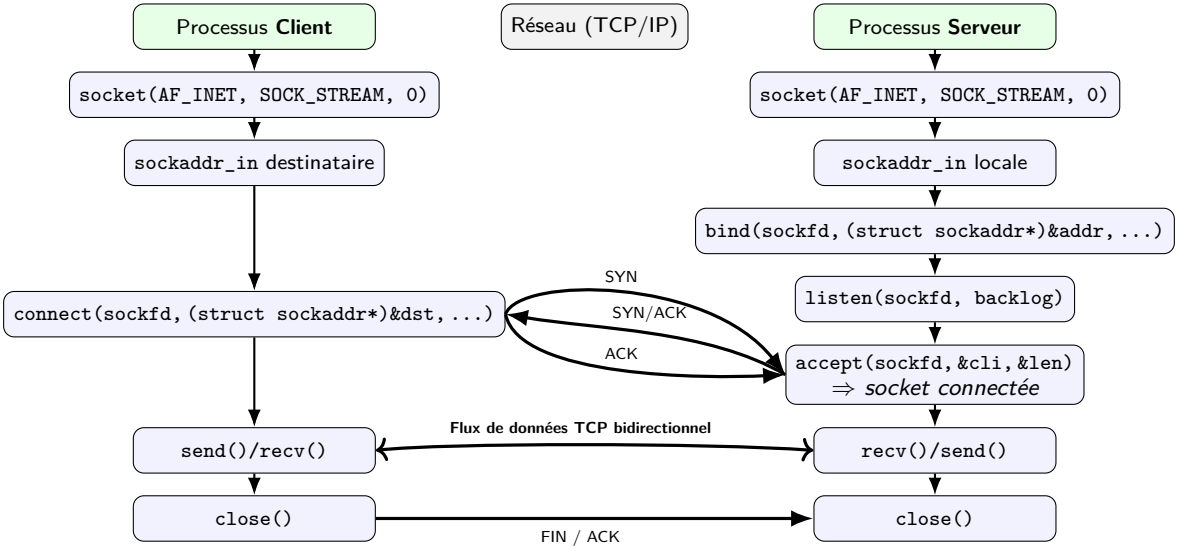
Échange de données

- Une fois connectés, client et serveur disposent chacun d'une **socket connectée** qui représente le canal de communication : socket de travail
  - `send()` ou `write()` : pour envoyer des données.
  - `recv()` ou `read()` : pour recevoir des données.

Fermeture de la connexion

- On ferme la socket avec `close()` lorsque la communication est terminée.

Schéma d'une communication **TCP** (simplifié sur le flux)



,

Mise en attente de connexions : `listen()`

Prototype

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

Rôle

- Transforme une socket **passive** en socket d'écoute.
- Prépare le noyau à mettre en file d'attente les connexions entrantes.
- Étape obligatoire avant `accept()` côté serveur TCP.

Paramètres et retour

- `sockfd` : descripteur de la socket serveur (créée + bindée).
- `backlog` : taille max de la file d'attente des connexions en attente.
- Retourne `0` si succès, `-1` en cas d'erreur.

Établir une connexion client -> serveur : `connect()`

Prototype

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

Rôle

- Associe une socket locale à une adresse distante (IP + port).
- Utilisé côté **client TCP** pour établir une connexion avec un serveur.
- En UDP, peut fixer une adresse par défaut (facilite l'usage de `send`/`recv` au lieu de `sendto`/`recvfrom`).

Retour

- 0 si succès.
- `-1` en cas d'erreur (par exemple si le serveur n'écoute pas).

Accepter une connexion : `accept()`

Prototype

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

Rôle

- Utilisé côté **serveur TCP** après `listen()`.
- Bloque en attente d'une demande de connexion entrante.
- Crée la nouvelle socket de communication dédiée au client : **socket de travail**

Paramètres et retour

- `sockfd` : socket d'écoute créée avec `socket()` + `bind()` + `listen()`.
- `addr`, `addrlen` : contiennent l'adresse du client accepté.
- Retour : descripteur de la nouvelle socket (ou `-1` en cas d'erreur).

Terminaison d'une connexion (fermeture de socket)

Appel système

```
int close(int sockfd);
```

Principe

- Libère le descripteur de la socket dans le processus local.
- Informe l'hôte distant de la fin de la communication.
- Peut être appelé côté client comme côté serveur.

Remarques

- Pour TCP, envoie un segment `FIN` (fin normale de connexion).
- Si plusieurs processus partagent la socket, elle ne se ferme vraiment qu'au dernier `close()`.
- Alternative : `shutdown()` pour fermer uniquement en lecture ou en écriture.

Exemple serveur TCP - envoi d'un message puis fermeture immédiate

```c
// Exemple minimal de serveur TCP (IPv4)
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main(void) {
  // 1) Création de la socket d'écoute
  int listen_fd = socket(AF_INET, SOCK_STREAM, 0);

  // 2) Adresse locale (0.0.0.0:4444)
  struct sockaddr_in server_addr;
  memset(&server_addr, 0, sizeof(server_addr));
  server_addr.sin_family = AF_INET;
  server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
  server_addr.sin_port = htons(4444);
```

Exemple serveur TCP - envoi d'un message puis fermeture immédiate

```
20    // 3) Association et mise en écoute
21    bind(listen_fd,  (struct sockaddr*)&server_addr,  sizeof(server_addr));
22    listen(listen_fd, 10);
23
24    // 4) Boucle d'acceptation
25    for (;;) {
26      struct sockaddr_in client_addr;
27      socklen_t client_len = sizeof(client_addr);
28
29      int conn_fd = accept(listen_fd,  (struct sockaddr*)&client_addr, &client_len);
30
31      const char *msg = "hello world";
32      send(conn_fd, msg, (int)strlen(msg), 0);
33      close(conn_fd);
34    }
35
36    // (jamais atteint ici)
37    close(listen_fd);
38    return 0;
39  }
```

Exemple client TCP — Réception d'un message et fermeture immédiate

```c
// Exemple minimal de client TCP (IPv4)

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main(void) {
  // 1) Création de la socket de communication
  int sock_fd = socket(AF_INET, SOCK_STREAM, 0);

  // 2) Configuration de l'adresse du serveur
  struct sockaddr_in server_addr;
  memset(&server_addr, 0, sizeof(server_addr));
  server_addr.sin_family      = AF_INET;
  server_addr.sin_port        = htons(4444);
  server_addr.sin_addr.s_addr = inet_addr("1.2.3.4");
```

Exemple client TCP — Réception d'un message et fermeture immédiate

```
    // 3) Connexion au serveur
    connect(sock_fd, (struct sockaddr*)&server_addr, sizeof(server_addr));

    // 4) Réception d'un message
    char buffer[1024];
    recv(sock_fd, buffer, sizeof(buffer) - 1, 0);

    // 5) Fermeture de la socket
    close(sock_fd);
    return 0;
}
```