# Signaux et sockets résesau

**SYE**

7 - Signaux & Sockets Reseau


## Résumé du document

Ce document couvre les bases de la communication réseau en architecture client-serveur, incluant les signaux et sockets en TCP/IP. Il décrit la création, l'écoute, la connexion, l'échange et la fermeture des connexions client-serveur. L'approche concurrente avec processus/threads est soulignée pour sa scalabilité, permettant une communication efficace avec plusieurs clients.


## Table des matières

# 1. Introduction aux signaux

Les signaux sont des informations de type événement qui peuvent être transmis à un processus d'une manière **asynchrone**.

- le **noyau** ou un **processus** peut envoyer des signaux
- chaque signal possède un comportement par défaut
- le signal provoque l'exécution d'un traitant (handler) dans l'espace utilisateur
  - le processus doit être dans un état **running**

**Un seul exemplaire d'un signal peut se retrouver dans la file d'attente.**

## 1.1. Gestion du signal

- Les signaux peuvent agir comme des alertes pour les processus.
- Lorsqu'une séquence de touches, comme `CTRL/C`, est activée, le noyau envoie un signal au processus concerné.
- Le processus peut choisir de réagir de manière spécifique à ce signal.
- Si aucune action spécifique n'est définie, le comportement par défaut du signal s'applique.
- Pour `CTRL/C`, le comportement par défaut entraîne la fin immédiate du processus.

### 1.1.1. Masquer / ignorer singal

- il est possible de **masquer** un signal cela le temporise et le délivrera lors ce qu'il sera démasqué
- il est possible d'**ignorer** un signal cela entrainera la perte du signals

## 1.2. Liste de signaux

Voici quelques signaux à noter

| **Signal** | **Evénement** |
|---|---|
| SIGINT | Frappe du caractère intr (CTRL/C) sur le clavier du terminal de contrôle |
| SIGKILL | Signal de terminaison |
| SIGTERM | Signal de terminaison |
| SIGUSR1 | Signal émis par un processus utilisateur |
| SIGUSR2 | Signal émis par un processus utilisateur |
| SIGCHLD | Terminaison d'un fils |

# 2. Sockets réseau

## 2.1. Introduction aux Sockets

- Un **socket** est une extrémité de communication permettant la transmission de données entre applications locales ou distantes.
- Types de sockets : **TCP**, **UDP**, et **raw** pour manipuler des paquets IP directement.
- Norme **POSIX** pour l'API des sockets, intégrant des appels système similaires à la manipulation de fichiers.

## 2.2. Sockets TCP

- Requièrent une connexion entre deux processus (client et serveur).
- Utilisés pour des communications locales ou entre machines.
- La **connexion loopback** (adresse IP `127.0.0.1`) est utilisée pour les communications locales.

## 2.3. Structures pour les Adresses

- **Structure sockaddr** : Structure générique pour décrire une adresse réseau.
- **Structure sockaddr_in** : Spécifique pour les adresses IP avec champs :
  - `sin_family`: Famille d'adresses (ex. `AF_INET` pour IPv4).
  - `sin_port`: Numéro de port en format réseau.
  - `sin_addr`: Adresse IP en format numérique.

### 2.3.1. Conversion d'Adresses

- **Fonctions de conversion IP** :
  - `inet_pton()`: Convertit une adresse IP en format texte vers le format numérique.
  - `inet_ntop()`: Convertit une adresse IP en format numérique vers le format texte.
- **Conversion de ports** :
  - `htons()` et `ntohs()`: Convertissent les numéros de port au format réseau.

# 3. Appels systèmes liés aux sockets

## 3.1. Syscalls dédiés à la gestion des sockets

Les appels système suivants permettent de gérer les sockets pour leur initialisation, la mise en écoute et l'établissement de connexions.

### 3.1.1. Gestion des sockets : Appels système principaux

- **Serveur :** Prépare la connexion
  - `bind()`: Associe le socket à une adresse IP et un port.
  - `listen()`: Met le socket en écoute pour les connexions entrantes.
  - `accept()`: Accepte une connexion entrante, créant une communication avec le client.
- **Client :** Établit la connexion
  - `connect()`: Connecte le socket client à un socket serveur à l'adresse IP et au port spécifiés.

Les informations détaillées sur ces fonctions sont disponibles dans les pages man (`man socket`).

### 3.1.2. Exemples d'utilisation

#### 3.1.2.1. Côté serveur

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
bind(sockfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
listen(sockfd, 5);
int clientfd = accept(sockfd, (struct sockaddr*)NULL, NULL);
```

#### 3.1.2.2. Côté client

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
connect(sockfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
```

## 3.2. Gestion du flux de données

- La gestion des données dans un socket fonctionne de manière similaire aux fichiers.
- Attention à l'argument `<len>` ou `<count>` dans les fonctions `read()` et `write()` :
  - **`write()` / `send()`** : `<len>` spécifie le nombre d'octets à envoyer depuis le buffer.
    - Retourne le nombre d'octets effectivement envoyés.
    - En mode bloquant (par défaut), la fonction attend jusqu'à ce que tous les octets soient envoyés.
  - **`read()` / `recv()`** : `<len>` spécifie le nombre maximum d'octets à lire dans le buffer.
    - Retourne le nombre d'octets réellement reçus, souvent inférieur à `<len>`.

### 3.2.1. Exemple de transmission de données

**Envoi (côté client) :**

```
char buffer[] = "Hello Server";
int bytes_sent = send(sockfd, buffer, strlen(buffer), 0);
```

**Réception (côté serveur) :**

```
char buffer[1024];
int bytes_received = recv(clientfd, buffer, sizeof(buffer), 0);
```

### 3.2.2. Contrôle du mode bloquant et non-bloquant

- Par défaut, les sockets sont en mode bloquant. Cela signifie que les fonctions `read()` et `write()` attendent jusqu'à ce que l'opération soit terminée.
- Pour changer le mode de blocage, utilisez les appels `ioctl()` ou `fcntl()` pour configurer des options telles que le mode non-bloquant, ce qui permet d'éviter l'attente en cas d'absence de données.

### 3.2.3. Exemple de configuration en mode non-bloquant

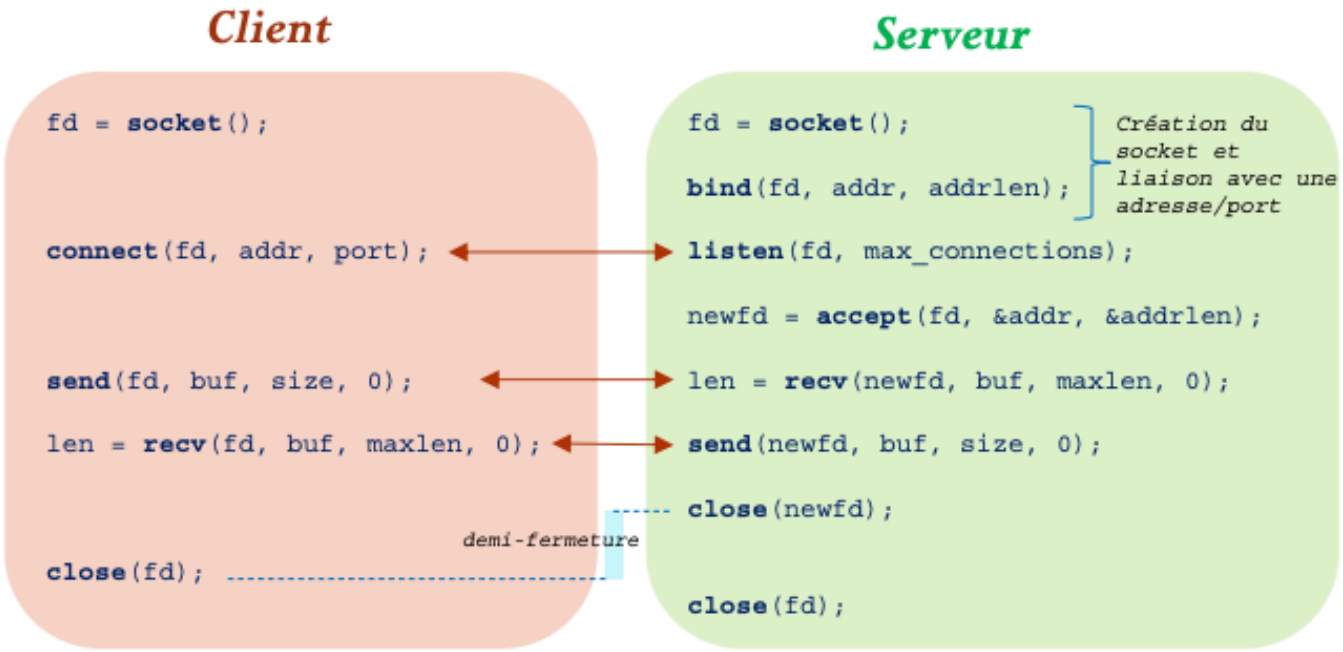
```
#include <fcntl.h>

int flags = fcntl(sockfd, F_GETFL, 0);
fcntl(sockfd, F_SETFL, flags | O_NONBLOCK);
```

### 3.3. Résumé

- **Sockets** : Permettent la communication entre applications locales et distantes.
- **Appels système principaux** :
  - Serveur : `bind()`, `listen()`, `accept()`.
  - Client : `connect()`.
- **Gestion des données** :
  - `send()` / `write()` : Envoie de données (le nombre d'octets peut varier selon le mode).
  - `recv()` / `read()` : Réception de données (la taille reçue peut être inférieure au maximum spécifié).
- **Modes de fonctionnement** :
  - Bloquant par défaut, peut être configuré en mode non-bloquant avec `fcntl()` ou `ioctl()`.

Ces appels systèmes permettent d'établir des connexions fiables en mode client-serveur, offrant flexibilité et contrôle sur le transfert de données dans les applications réseau.

# 4. Approche client-serveur




L'architecture client-serveur permet la communication entre deux entités où le serveur écoute et gère les connexions des clients. Voici les étapes principales et les concepts associés.

## 4.1. Strcuture connexion client-serveur

### 4.1.1. Établissement de Connexion Client-Serveur

1. **Initialisation du Serveur :**
   - Le serveur crée un socket (représenté par un descripteur de fichier `fd`) pour écouter les connexions entrantes.
   - Il associe ce socket à une adresse IP et un port avec l'appel système `bind()`, ce qui est essentiel pour définir l'adresse à laquelle les clients se connecteront.
   - Ensuite, il utilise `listen()` pour demander au noyau de créer une file d'attente pour les connexions entrantes, définissant le nombre maximal de connexions en attente.
2. **Connexion d'un Client :**
   - Le client initialise également un socket et utilise `connect()` pour tenter de se connecter au serveur en spécifiant l'adresse IP et le port du serveur.
3. **Acceptation de la Connexion par le Serveur :**
   - Lorsque le serveur reçoit une demande de connexion, il utilise `accept()` pour accepter cette connexion.
   - Cette opération associe un nouveau socket dédié à cette connexion avec le client, évitant toute interférence avec le socket d'écoute principal.
   - Une fois connecté, le client et le serveur peuvent commencer à échanger des données.
4. **Fermeture de la Connexion :**
   - La fermeture des sockets (avec `close()`) est indépendante pour chaque côté (client ou serveur), permettant une "demi-fermeture" où une des deux parties peut fermer sa connexion sans que l'autre partie ne le fasse immédiatement.

### 4.1.2. Serveur Itératif vs Serveur Concurrent

Pour traiter plusieurs clients sans délai, il existe deux principaux types de serveurs :

- **Serveur Itératif :**
  - Gère un client à la fois dans un seul processus. Une nouvelle connexion ne peut être acceptée qu'après la fin de la précédente.
  - Simple à implémenter mais peu efficace pour des charges de travail avec plusieurs clients.

- **Serveur Concurrent :**
  - Permet des communications en parallèle avec plusieurs clients.
  - Dès qu'une connexion client est acceptée, le serveur crée un nouveau processus ou un thread dédié pour gérer cette connexion, ce qui permet au serveur principal de retourner immédiatement en écoute.
  - Chaque client se voit attribuer un port particulier (côté serveur) pour une communication isolée et full-duplex (bidirectionnelle).

#### 4.1.3. Serveur TCP/IP

L'utilisation de TCP pour l'architecture client-serveur présente plusieurs avantages :

- **Fiabilité de la Connexion** : Le protocole TCP garantit une transmission robuste et point-à-point.
- **Contrôle de Flux** : TCP gère le débit pour éviter la surcharge du réseau.
- **Transmission Full-Duplex** : Permet l'envoi et la réception simultanés de données entre le client et le serveur.

#### 4.1.4. Schéma de Fonctionnement d'un Serveur Concurrent

1. **Écoute sur un Port Passif** : Le serveur principal écoute les demandes de connexion.
2. **Connexion du Client** : Le client utilise `connect()` pour se connecter au serveur.
3. **Création d'un Processus ou Thread** : Le serveur crée un nouveau processus (par exemple via `fork()`) ou un thread pour gérer le client.
4. **Échange de Données** : Le client et le processus du serveur échangent des données avec `read()` et `write()`.
5. **Fermeture des Connexions** : Une fois la communication terminée, le processus dédié au client est terminé, et les ports sont libérés.

#### 4.1.5. Avantages de l'Approche Concurrente

- **Réactivité** : Le serveur est toujours prêt à accepter de nouvelles connexions.
- **Isolation** : Chaque client est géré dans un processus ou un thread distinct, minimisant les interférences entre les connexions.
- **Scalabilité** : Permet de gérer plusieurs clients en même temps, optimisant ainsi les ressources du serveur.