## Q.1

Socket serveur créée avec le descripteur : 3

Socket client créée avec le descripteur : 3

Dans un système Linux (y compris sous WSL), chaque processus gère une table de **descripteurs de fichiers**.  
 Les trois premiers sont réservés par défaut :

* **0** → stdin (entrée standard)
* **1** → stdout (sortie standard)
* **2** → stderr (sortie d’erreur standard)

Lorsqu'un processus ouvre un nouveau fichier ou crée une socket, le **premier descripteur disponible** est attribué.  
 Dans ton cas :

* **Le serveur et le client sont des processus distincts**, chacun ayant sa propre table des descripteurs.
* Comme aucun fichier supplémentaire n’a été ouvert avant la création de la socket, le premier descripteur libre est 3.

#### **Création d’une socket TCP**

Utiliser la fonction socket(), qui retourne un **descripteur de fichier**.  
  
CopierModifier  
int serveur\_socket = socket(AF\_INET, SOCK\_STREAM, 0);

if (serveur\_socket == -1) {

perror("Erreur lors de la création de la socket serveur");

exit(EXIT\_FAILURE);

}

* **Vérifier si le retour est négatif** (erreur).

#### **2️⃣ Option SO\_REUSEADDR**

Permet de **réutiliser immédiatement le port** après fermeture de la socket.  
  
CopierModifier  
int reuse = 1;

setsockopt(serveur\_socket, SOL\_SOCKET, SO\_REUSEADDR, &reuse, sizeof(reuse));

* **Éviter les erreurs** si le programme est redémarré rapidement.

#### **3️⃣ Configuration de l’adresse du serveur**

* On utilise struct sockaddr\_in pour stocker l’IP et le port.

Avant utilisation, **remettre la structure à zéro** avec memset().  
  
struct sockaddr\_in my\_addr;

memset(&my\_addr, 0, sizeof(my\_addr));

my\_addr.sin\_family = AF\_INET;

my\_addr.sin\_port = htons(8080); // Port choisi

my\_addr.sin\_addr.s\_addr = INADDR\_ANY; // Accepte toutes les connexions

* **INADDR\_ANY** → écoute sur toutes les interfaces réseau disponibles.

#### **4️⃣ Associer la socket au port et à l’IP (bind)**

bind() permet de **lier la socket à une adresse IP et un port**.  
  
if (bind(serveur\_socket, (struct sockaddr\*)&my\_addr, sizeof(my\_addr)) < 0) {

perror("Erreur de liaison (bind)");

close(serveur\_socket);

exit(EXIT\_FAILURE);

}

* **Toujours vérifier le retour** pour éviter des erreurs cachées.

#### **5️⃣ Fermeture propre de la socket**

Une fois qu’on a terminé, on **ferme la socket** avec close().  
close(serveur\_socket);

### **🔥 Code final minimal à retenir**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <unistd.h>

int main() {

// 1️⃣ Création de la socket

int serveur\_socket = socket(AF\_INET, SOCK\_STREAM, 0);

if (serveur\_socket == -1) {

perror("Erreur lors de la création de la socket serveur");

exit(EXIT\_FAILURE);

}

// 2️⃣ Autoriser la réutilisation du port

int reuse = 1;

setsockopt(serveur\_socket, SOL\_SOCKET, SO\_REUSEADDR, &reuse, sizeof(reuse));

// 3️⃣ Configuration de l'adresse du serveur

struct sockaddr\_in my\_addr;

memset(&my\_addr, 0, sizeof(my\_addr));

my\_addr.sin\_family = AF\_INET;

my\_addr.sin\_port = htons(8080);

my\_addr.sin\_addr.s\_addr = INADDR\_ANY;

// 4️⃣ Liaison de la socket (bind)

if (bind(serveur\_socket, (struct sockaddr\*)&my\_addr, sizeof(my\_addr)) < 0) {

perror("Erreur de liaison (bind)");

close(serveur\_socket);

exit(EXIT\_FAILURE);

}

printf("Socket serveur créée avec le descripteur : %d\n", serveur\_socket);

// 5️⃣ Fermeture propre de la socket

close(serveur\_socket);

return 0;

}

### **💡 Points clés à retenir**

✅ **Toujours vérifier les erreurs après chaque appel (socket(), bind(), etc.)** ✅ **Utiliser setsockopt() pour éviter les erreurs de port bloqué** ✅ **Toujours remettre sockaddr\_in à zéro (memset())** ✅ **Utiliser htons(port) pour convertir le port en format réseau** ✅ **Fermer la socket avec close()**

## Q.2

### **Résumé :**

* **INADDR\_ANY** permet à la socket d'accepter des connexions sur **toutes les interfaces réseau disponibles**, ce qui simplifie la gestion des connexions et permet d’éviter de spécifier explicitement une adresse IP.
* On ne récupère pas explicitement l'adresse IP d'une interface dans ce cas pour des raisons de **flexibilité** et de **simplicité**, afin de rendre le serveur accessible sans avoir à gérer les différentes interfaces réseau manuellemen

## Q.3

Adapter le serveur et le client pour qu'ils fonctionnent avec les ordres d’octets de format réseau (Big Endian). Utiliser les fonctions de conversion de l'API arpa/inet.h pour garantir l’interopérabilité entre des machines avec différentes architectures.

**serveur.c :**

* Utilisation de htonl() et htons() pour lier la socket serveur au format réseau.
* Initialisation de sockaddr\_in et utilisation de bind() pour lier la socket au port et à l'adresse.

**client.c :**

* Utilisation de htons() pour définir le port du serveur.
* Utilisation de inet\_aton() pour convertir l'adresse IP du serveur en format réseau.
* Utilisation de connect() pour établir une connexion avec le serveur.

Lorsque vous appelez bind() sur un port déjà utilisé, la fonction échoue et retourne une valeur négative (généralement -1). Concrètement :

* **Erreur retournée :** bind() renvoie -1, et la variable errno est définie sur EADDRINUSE (adresse déjà utilisée).
* **Message d'erreur :** Une utilisation de perror() affichera un message tel que « Erreur de liaison (bind): Address already in use ».
* **Conséquence :** Le processus ne peut pas utiliser ce port tant qu'il est occupé par un autre socket ou qu'il n'est pas libéré par le système (parfois à cause du mécanisme TIME\_WAIT).

Ce comportement permet d'éviter que deux applications ne se disputent le même port, ce qui pourrait provoquer des conflits de communication.

## Q.4

La fonction **accept()** retourne un **descripteur de fichier** qui représente la **connexion** établie avec le client. Plus précisément :

### **Valeur retournée par accept() :**

* Si la fonction **accept()** réussit, elle retourne un **nouveau descripteur de fichier** qui représente la **connexion avec le client**. Ce descripteur est utilisé pour communiquer avec le client spécifique.
* Si **accept()** échoue, elle retourne **-1** et la variable **errno** est définie pour indiquer l'erreur spécifique. Cela pourrait être dû à des problèmes de réseau, des erreurs système ou d'autres raisons.

## Q.5

### **4. Utilisation d'un buffer :**

* Le **buffer** est utilisé pour stocker les données envoyées et reçues. Vous devez vous assurer qu'il est suffisamment grand pour contenir toutes les données que vous souhaitez envoyer ou recevoir. Utilisez un buffer de taille fixe, par exemple 1024 octets, ou bien une gestion dynamique de mémoire si nécessaire.

### **5. Exécution côté serveur et client :**

* Le serveur doit écouter et accepter les connexions des clients. Une fois la connexion établie, il peut envoyer et recevoir des messages.
* Le client se connecte au serveur, envoie un message et attend une réponse.

### **Si le nombre d'octets à recevoir est plus grand que la taille du buffer :**

* **Les données excédentaires sont ignorées.**
* Le système va lire seulement jusqu'à la taille maximale du buffer que vous avez spécifiée. Cela signifie que si vous spécifiez un buffer de taille n mais qu'il y a m octets (où m > n) à recevoir, seuls n octets seront lus, et les m-n octets restants seront "oubliés" ou mis en attente pour une future lecture.
* **Aucun avertissement** ne sera donné par recv() à moins que vous ne l'ayez spécifié (comme une erreur de dépassement de tampon), et la fonction recv() retournera le nombre d'octets effectivement lus (qui est égal à la taille du buffer dans ce cas).

Exemple :

c

CopierModifier

char buffer[10]; // Buffer de taille 10 octets

int n = recv(socket, buffer, sizeof(buffer), 0);

// Si le serveur envoie 20 octets, seuls les 10 premiers seront reçus.

### **2. Si le nombre d'octets à recevoir est plus petit que la taille du buffer :**

* **Le buffer non utilisé est laissé vide.**
* recv() lira les données disponibles et les placera dans le buffer, mais il n’écrira pas dans la partie excédentaire du buffer.
* Si le nombre d'octets reçus est inférieur à la taille du buffer, les octets restants dans le buffer resteront non modifiés.
* Si la connexion est fermée de l'autre côté, recv() retournera **0** pour indiquer qu'il n'y a plus de données à lire.

## Q.6

**Résumé** :

* **Adresse IP** : Le serveur voit toujours 127.0.0.1 (localhost) si le client et le serveur sont sur la même machine.
* **Port** : Le port côté client est différent à chaque nouvelle connexion, car il est attribué dynamiquement par le système d'exploitation.

## Q.7

Dans le processus père, la fonction **fork()** retourne le PID (Process ID) du processus fils qui vient d'être créé, c'est-à-dire une valeur positive.

**Ce que cela signifie :**

* Une valeur positive indique que le processus père peut identifier et gérer le processus fils (par exemple, pour attendre sa fin avec waitpid()).
* Cela permet de différencier le code exécuté dans le processus père (où la valeur est > 0) de celui exécuté dans le processus fils (où fork() retourne 0).

En résumé, la valeur retournée dans le processus père est le PID du processus fils, ce qui signifie qu'un nouveau processus a été créé avec succès.

# Lexique

int serveur\_socket = socket(AF\_INET, SOCK\_STREAM, 0);

🔹 Création de la socket serveur :

socket() retourne un descripteur de fichier (identifiant numérique).

Paramètres :

* AF\_INET → Utilisation du protocole IPv4.
* SOCK\_STREAM → Utilisation de TCP (mode connexion, fiable).
* 0 → Choix automatique du protocole (TCP est l’unique choix ici).

Si socket() échoue, il retourne -1.

int reuse = 1;

setsockopt(serveur\_socket, SOL\_SOCKET, SO\_REUSEADDR, &reuse, sizeof(reuse));

🔹 Réutilisation immédiate du port :

* setsockopt() configure des options sur la socket.
* SOL\_SOCKET → Niveau socket.
* SO\_REUSEADDR → Permet de réutiliser immédiatement le port après fermeture.
* &reuse, sizeof(reuse) → Active l’option (1 pour vrai).

struct sockaddr\_in my\_addr;

memset(&my\_addr, 0, sizeof(my\_addr));

🔹 Déclaration et initialisation de l’adresse du serveur :

* struct sockaddr\_in → Contient les informations IP et port.
* memset() met tous les octets à zéro pour éviter des valeurs parasites.

my\_addr.sin\_family = AF\_INET;

my\_addr.sin\_port = htons(8080);

my\_addr.sin\_addr.s\_addr = INADDR\_ANY;

🔹 Configuration des paramètres de l’adresse :

* sin\_family = AF\_INET → Spécifie IPv4.
* sin\_port = htons(8080) → Définit le port 8080 (converti en format réseau avec htons()).
* sin\_addr.s\_addr = INADDR\_ANY → Permet d’écouter sur toutes les interfaces réseau.

if (bind(serveur\_socket, (struct sockaddr\*)&my\_addr, sizeof(my\_addr)) < 0) {

perror("Erreur de liaison (bind)");

close(serveur\_socket);

exit(EXIT\_FAILURE);

}

🔹 Association de la socket à une adresse et un port (bind) :

* bind() attache la socket au port 8080 sur toutes les interfaces disponibles.
* (struct sockaddr\*)&my\_addr → Conversion nécessaire car bind() attend struct sockaddr\*.
* sizeof(my\_addr) → Taille de la structure passée en argument.
* Si bind() échoue (ex : port déjà utilisé), perror() affiche l’erreur, on ferme la socket et on quitte.