Cours "Système d'exploitation"

R3.05

2ème année BUT de Caen

Département d'informatique

*Développement système : **les sockets***

Table des matières

I. Introduction :

" La notion de socket a été introduite dans les distributions de Berkeley (un fameux système de type UNIX, dont beaucoup de distributions actuelles utilisent des morceaux de code), c'est la raison pour laquelle on parle parfois de sockets BSD (Berkeley Software Distribution)"

Un socket représente une interface de communication logicielle avec le système d'exploitation qui permet d'exploiter les services d'un protocole réseau et par laquelle une application peut envoyer et recevoir des données.

Un socket désigne aussi un ensemble normalisé de fonctions de communication (une API) qui est proposé dans quasiment tous les langages de programmation populaires (C, Java, C#, C++, ...) et répandue dans la plupart des systèmes d'exploitation (UNIX/Linux, Windows, ...).

Il s'agit d'un modèle permettant la communication inter processus (*IPC* - Inter Processus Communication) afin de permettre à divers processus de communiquer aussi bien sur une même machine qu'à travers un réseau TCP/IP.

La communication par socket est souvent comparée aux communications humaines. On distingue ainsi deux modes de communication :

- Le mode connecté (comparable à une communication téléphonique), utilisant le protocole TCP.

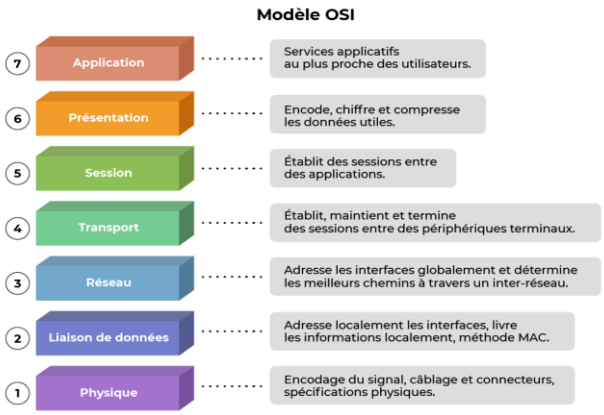
Dans ce mode de communication, une connexion durable est établie entre les deux processus, de telle façon que le socket de destination n'est pas nécessaire à chaque envoi de données.

- Le mode non connecté (analogue à une communication par courrier), utilisant le protocole UDP.

Ce mode nécessite l'adresse de destination à chaque envoi, et aucun accusé de réception n'est donné.

Les sockets sont généralement implémentés en langage C, et utilisent des fonctions et des structures disponibles dans la librairie ***<sys/socket.h>.***

Position des sockets dans le modèle OSI :



Les sockets se situent au-dessus de la couche transport du modèle OSI, qui utilisent les services de la couche réseau. Les couches Application, Présentation, et session utilisent les sockets.

II/ Utilisation des sockets :

Un socket étant un point de communication, il est relié à une adresse IP et un numéro de port.

En programmation, si on utilise comme point d'entrée initial le niveau TRANSPORT, il faudra alors choisir un des deux protocoles de cette couche :

- TCP (Transmission Control Protocol) est un protocole de transport fiable, en mode connecté.
- UDP (User Datagram Protocol) est un protocole souvent décrit comme étant non fiable, en mode non-connecté, mais plus rapide que TCP.

Le numéro de port sert à identifier un processus en cours de communication. C'est le système d'exploitation qui attribue les numéros de ports. Ce numéro est codé sur 16 bits ce qui implique qu'i existe un maximum de 65536 ports (2^{16}).

Deux situations sont possibles :

- **Processus client** : le numéro de port utilisé par le client est envoyé au serveur. Dans ce cas le client demande un numéro de port quelconque au système, à condition qu'il ne soit pas déjà utilisé.
- **Processus serveur** : le numéro de port utilisé par le serveur doit être connu du client. Dans ce cas le serveur doit demander un numéro de port précis au système d'exploitation qui vérifie seulement si le numéro demandé n'est pas déjà attribué.

1. Déroulement d'une connexion

La communication par socket utilise un descripteur pour désigner la connexion sur laquelle des données sont envoyées ou reçues. La première opération à effectuer est la création du socket qui permet de récupérer son identifiant. Cet identifiant est un entier unique qui identifie la connexion et permet d'envoyer ou recevoir des informations.

L'ouverture d'un socket se fait en deux étapes :

- La création d'un socket et de son descripteur par la fonction ***socket***
- La fonction ***bind*** permet de spécifier le type de communication associé au socket (protocole TCP ou UDP)

Un serveur doit être à l'écoute de messages éventuels. Toutefois, l'écoute se fait différemment selon que le socket est en mode connecté (TCP) ou non (UDP).

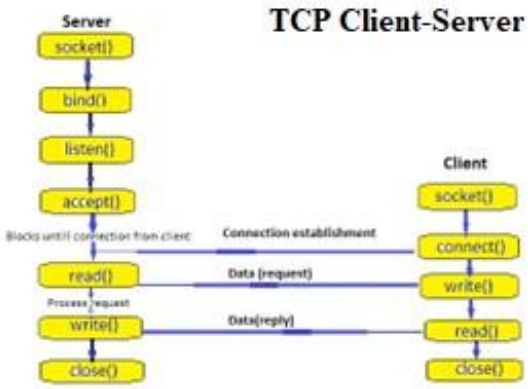
- **En mode connecté,** le message est reçu d'un seul bloc.

Ainsi en mode connecté, la fonction ***listen*** permet de placer le socket en mode passif (à l'écoute des messages). En cas de message entrant, la connexion peut être acceptée grâce à la fonction ***accept***. Lorsque la connexion a été acceptée, le serveur reçoit les données grâce à la fonction ***recv.***

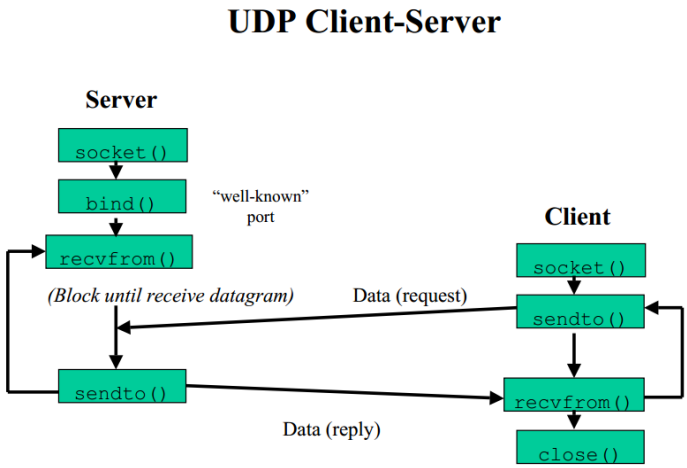
- **En mode non connecté**, comme dans le cas du courrier, le destinataire reçoit le message petit à petit (la taille du message est indéterminée) et de façon désordonnée. Le serveur reçoit les données grâce à la fonction ***recvfrom***.

- La fin de la connexion se fait grâce à la fonction ***close***.

2. Schéma d'une communication en mode connecté :



3. Schéma d'une communication en mode non connecté :



4. Présentation des fonctions :

Pour dialoguer, chaque processus devra préalablement créer un socket de communication en indiquant :

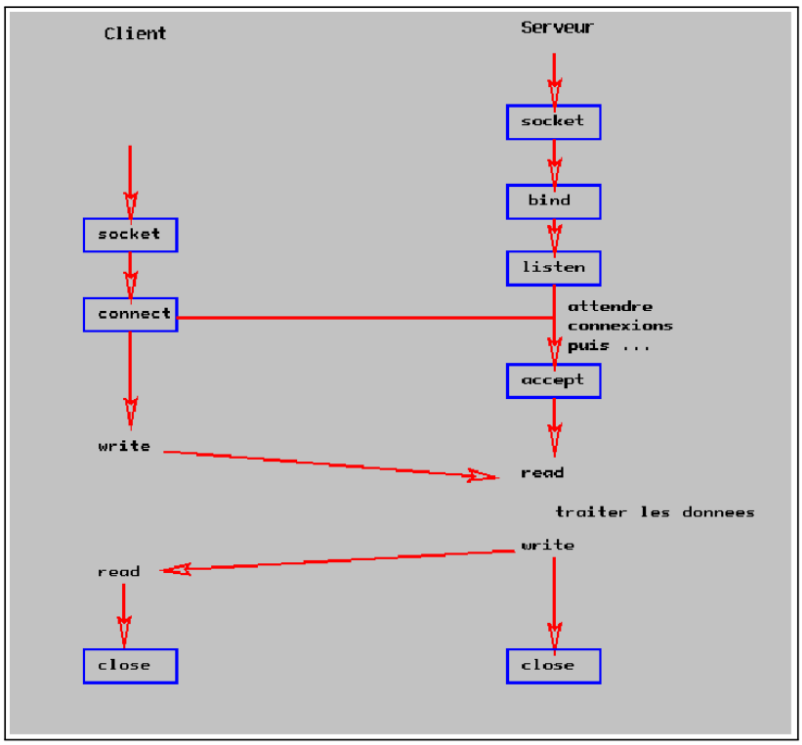
- Le domaine de communication : ceci sélectionne la famille de protocole à employer. Il faut savoir que chaque famille possède son adressage. Par exemple pour les protocoles Internet IPv4, on utilisera le domaine PF_INET ou AF_INET et AF_INET6 pour le protocole IPv6.
- Le type de socket à utiliser pour le dialogue. Pour PF_INET, on aura le choix entre :
  - SOCK_STREAM (qui correspond à un mode connecté donc TCP par défaut),
  - SOCK_DGRAM (qui correspond à un mode non connecté donc UDP),
  - SOCK_RAW (qui permet un accès direct aux protocoles de la couche Réseau comme IP, ICMP, ...).

- Le protocole à utiliser sur le socket. Le numéro de protocole dépend du domaine de communication et du type du socket. Normalement, il n'y a qu'un seul protocole par type de socket pour une famille donnée (*SOCK_STREAM* → *TCP* et *SOCK_DGRAM* → *UDP*). Néanmoins, rien ne s'oppose à ce que plusieurs protocoles existent, auquel cas il est nécessaire de le spécifier (c'est le cas pour SOCK_RAW où il faudra préciser le protocole à utiliser).

a) Programmation TCP

Dans cette partie, nous allons détailler les fonctions de mise en œuvre d'une communication client/serveur en utilisant un socket TCP.

Étapes de l'échange :



Réalisation d'un client TCP :

Étape 1 : La fonction socket

La fonction ***socket*** permet de créer un socket et renvoie un descripteur en cas de succès et -1 en cas d'erreur.

Syntaxe :

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domaine, int type, int protocole);
```

- **domaine** représente la famille de protocole utilisé (*AF_INET* ou PF_INET pour TCP/IP utilisant une adresse Internet sur 4 octets: l'adresse IP ainsi qu'un numéro de port afin de pouvoir avoir plusieurs sockets sur une même machine, *AF_UNIX* pour les communications UNIX en local sur une même machine)

- **type** indique le type de service (orienté connexion ou non). Dans le cas d'un service orienté connexion (c'est généralement le cas), l'argument *type* doit prendre la valeur SOCK_STREAM (communication par flot de données). Dans le cas contraire (protocole UDP) le paramètre *type* doit alors valoir SOCK_DGRAM (utilisation de datagramme, blocs de données). Il existe aussi les types SOCK_RAW, orienté vers l'échange de datagrammes au niveau bas du protocole, SOCK_RDM, orienté vers la transmission de datagrammes en mode non connecté avec maximum de fiabilité et SOCK_SEQPACKET, orienté vers la transmission de datagrammes en mode connecté.

- **protocole** permet de spécifier un protocole permettant de fournir le service désiré. Dans le cas de la suite TCP/IP il n'est pas utile, on le mettra ainsi toujours à 0.

Exemple:

```
int main()
{
	int descripteurSocket;

	// Crée un socket de communication
	descripteurSocket = socket(PF_INET, SOCK_STREAM, 0); /* 0 indique que l'on utilisera le
protocole par défaut associé à SOCK_STREAM soit TCP */

	// Teste la valeur renvoyée par l'appel système socket()
	if (descripteurSocket < 0) /* échec ? */
	{
		perror("socket"); // Affiche le message d'erreur
		exit(-1); // On sort en indiquant un code erreur
	}

	printf("Socket créée avec succès ! (%d)\n", descripteurSocket);
	// On ferme la ressource avant de quitter
	close(descripteurSocket);
	return 0;
}
```

Étape 2 : La fonction connect

La fonction *connect* permet de connecter le socket créé au serveur distant.

Syntaxe:

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

- **sockfd :** descripteur du socket créé par la fonction socket,
- **serv_addr :** structure d'adresse du serveur auquel on souhaite connecter le socket,
- **addrlen :** taille de cette structure

La fonction ***connect*** retourne 0 si tout se passe bien et -1 sinon.

L'interface de socket propose une structure d'adresse de socket générique et une structure d'adresse de socket IPV4.

Structure générique :

```
	struct sockaddr {
	u_char sa_len;            //longueur effective de l'adresse
	sa_family_t sa_family;   // famille de protocole
	char sa_data[14];         //adresse complète
	};
```

sa_len : est un octet (u_char) permettant de définir la longueur utile de l'adresse (la partie réellement utilisée par sa_data).

sa_family : représente la famille de protocole (AF_INET pour TCP/IP)

sa_data : chaine de 14 caractères maximum contenant l'adresse.

Structure d'adresse de socket IPV4 :

```
struct sockaddr_in{
        uint8_t sin_len;          //taille de la structure
        sa_family_t sin_family;   //famille de la socket
        in_port_t sin_port;       //numéro de port
        struct in_addr sin_addr; //adresse IPV4
        char sin_zero[8];  //ajustement pour être compatible avec socaddr
};
```

Avec

```
struct in_addr{
        in_addr_t s_addr; //adresse IPv4 sur 32 bits
};
```

- **sin_len** : utilisé uniquement avec les sockets de routage
- **sin_family** : AF_INET pour un socket IPV4
- **sin_port** : numéro de port sur 16 bits
- **sin_addr** : adresse IPV4 sur 32 bits

Ces structures sont déclarées dans ***<netinet/in.h>***

Il faut donc initialiser une structure ***sockaddr_in*** avec les informations du serveur (adresse IPV4 et numéro de port). Pour cela il faudra utiliser les fonctions suivantes :

- ✓ ***inet_aton*** : pour convertir une adresse IP depuis la notation IPV4 décimale pointée vers une forme binaire (dans l'ordre d'octet du réseau).
- ✓ ***htons :*** pour convertir le numéro de port (sur 16 bits) depuis l'ordre des octets de l'hôte vers celui du réseau.
- ✓ ***inet_pton :*** pour transformer l'adresse décimale pointée en adresse sur 32 bits
- ✓ ***inet _ntop :*** pour convertir une adresse en 32 bits en adresse sous forme décimale pointée

⚠ *Vous trouverez les syntaxe de ces différentes fonctions dans les pages de manul.*

Il existe deux façons de stocker un entier codé sur 16 bits en mémoire :

- ✓ ***little-endian :***

Octet de poids fort	Octet de poids faible

- ✓ ***big-endian :***

Octet de poids faible	Octet de poids fort

Par exemple le nombre **325** en little-endian : **00000001 | 01000101** sera lu sur un system en big-endian **01000101 | 00000001** càd **17665**

⚠ L'ordre des octets du réseau est en fait ***big-endian***. Il est donc plus prudent d'appeler des fonctions qui respectent cet ordre pour coder des informations dans les en-têtes des protocoles réseaux. Les champs port et adresse des structures socket doivent être en ordre réseau.

Exemple ;

```c
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h> /* pour memset */
#include <netinet/in.h> /* pour struct sockaddr_in */
#include <arpa/inet.h> /* pour htons et inet_aton */
int main()
{
	int descripteurSocket;
	struct sockaddr_in pointDeRencontreDistant;
	socklen_t longueurAdresse;
	// Création d'un socket de communication
	descripteurSocket = socket(PF_INET, SOCK_STREAM, 0);
	// Teste la valeur renvoyée par l'appel système socket()
	if (descripteurSocket < 0) {
		perror("socket"); // Affiche le message d'erreur
		exit(-1); // On sort en indiquant un code erreur
	}
	printf("Socket créé avec succès ! (%d)\n", descripteurSocket);
	//--- Début de l'étape n°2 :
	// Obtient la longueur en octets de la structure sockaddr_in
	longueurAdresse = sizeof(pointDeRencontreDistant);
	// Initialise à 0 la structure sockaddr_in
	memset(&pointDeRencontreDistant, 0x00, longueurAdresse);
	// Renseigne la structure sockaddr_in avec les informations du serveur
distant
	pointDeRencontreDistant.sin_family = PF_INET;
	// On choisit le numéro de port d'écoute du serveur
	pointDeRencontreDistant.sin_port = htons(IPPORT_USERRESERVED); // = 5000
	// On choisit l'adresse IPv4 du serveur
	inet_aton("192.168.52.2", &pointDeRencontreDistant.sin_addr); // à modifier
selon ses besoins
	// Débute la connexion vers le processus serveur distant
	if ((connect(descripteurSocket, (struct sockaddr*)&pointDeRencontreDistant,
			longueurAdresse)) == -1)
		{
			perror("connect"); // Affiche le message d'erreur
			close(descripteurSocket); // On ferme la ressource avant de
quitter
			exit(-2); // On sort en indiquant un code erreur
		}
	//--- Fin de l'étape n°2 !
	printf("Connexion au serveur réussie avec succès !\n");
	// On ferme la ressource avant de quitter
	close(descripteurSocket);
	return 0;
}
```

Pour tester ce client il faut un serveur !!

⚠ Dans l'architecture client/serveur, on rappelle que c'est le client qui initie l'échange. Il faut donc que le serveur soit en écoute avant que le client fasse sa demande.

Étape 3 : échange des données : fonctions read, write, recv et send

Une communication TCP est Bidirectionnelle (*full duplex*) et orientée flux d'octets.il faut donc des fonctions pour écrire et pour lire des octets dans le socket.

⚠ *Normalement les octets envoyés ou reçus respectent un protocole de la couche **Application**. Pour les exemples du cours, les données envoyées ou reçues seront de simples caractères ASCII. Les règles d'échanges seront donc : Le client envoie en premier une chaine de caractères et le serveur répondra "OK".*

Les fonctions d'échanges de données sur un socket TCP sont :

- read() et write() qui permettent la réception et l'envoi d'octets sur un descripteur de socket
- recv() et send() qui permettent la réception et l'envoi d'octets sur un descripteur de socket avec un paramètre flags

⚠ Les appels recv() et send() sont spécifiques aux sockets en mode connecté. La seule différence avec read() et write() est la présence de flags (cf. man send).

Exemple d'un client avec échange :

```
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h> /* pour memset */
#include <netinet/in.h> /* pour struct sockaddr_in */
#include <arpa/inet.h> /* pour htons et inet_aton */
#define LG_MESSAGE 256
int main()
{
	int descripteurSocket;
	struct sockaddr_in pointDeRencontreDistant;
	socklen_t longueurAdresse;
	char messageEnvoi[LG_MESSAGE]; /* le message de la couche Application ! */
	char messageRecu[LG_MESSAGE]; /* le message de la couche Application ! */
	int ecrits, lus; /* nb d'octets ecrits et lus */
	int retour;
	// Crée un socket de communication
	descripteurSocket = socket(PF_INET, SOCK_STREAM, 0); /* 0 indique que l'on
utilisera le protocole par défaut associé à SOCK_STREAM soit TCP */
	// Teste la valeur renvoyée par l'appel système socket()
	if (descripteurSocket < 0) /* échec ? */
	{
		perror("socket"); // Affiche le message d'erreur
		exit(-1); // On sort en indiquant un code erreur
	}
	printf("Socket créée avec succès ! (%d)\n", descripteurSocket);
	// Obtient la longueur en octets de la structure sockaddr_in
	longueurAdresse = sizeof(pointDeRencontreDistant);
	// Initialise à 0 la structure sockaddr_in
```

```
	memset(&pointDeRencontreDistant, 0x00, longueurAdresse);
	// Renseigne la structure sockaddr_in avec les informations du serveur distant
	pointDeRencontreDistant.sin_family = PF_INET;
	// On choisit le numéro de port d'écoute du serveur
	pointDeRencontreDistant.sin_port = htons(IPPORT_USERRESERVED); // = 5000
	// On choisit l'adresse IPv4 du serveur
	inet_aton("192.168.52.2", &pointDeRencontreDistant.sin_addr); // à modifier
selon ses besoins

	// Débute la connexion vers le processus serveur distant
	if ((connect(descripteurSocket, (struct
sockaddr*)&pointDeRencontreDistant,longueurAdresse)) == -1)
	{
		perror("connect"); // Affiche le message d'erreur
		close(descripteurSocket); // On ferme la ressource avant de quitter
		exit(-2); // On sort en indiquant un code erreur
	}
	printf("Connexion au serveur réussie avec succès !\n");
	//--- Début de l'étape n°3 : echange client/serveur
	// Initialise à 0 les messages
	memset(messageEnvoi, 0x00, LG_MESSAGE * sizeof(char));
	memset(messageRecu, 0x00, LG_MESSAGE * sizeof(char));
	// Envoie un message au serveur
	sprintf(messageEnvoi, "Hello world !\n");
	ecrits = write(descripteurSocket, messageEnvoi, strlen(messageEnvoi)); //
message à TAILLE variable
	switch (ecrits)
	{
	case -1: /* une erreur ! */
			perror("write");
			close(descripteurSocket);
			exit(-3);
	case 0: /* le socket est fermé */
			fprintf(stderr, "Le socket a été fermé par le serveur !\n\n");
			close(descripteurSocket);
			return 0;
	default: /* envoi de n octets */
			printf("Message %s envoyé avec succès (%d octets)\n\n",
messageEnvoi, ecrits);
	}
	/* Reception de la reponse du serveur */
	lus = read(descripteurSocket, messageRecu, LG_MESSAGE * sizeof(char)); /*
attend un message   de TAILLE fixe */
	switch (lus)
	{
	case -1: /* une erreur ! */
		perror("read");
		close(descripteurSocket);
		exit(-4);
	case 0: /* le socket est fermée */
		fprintf(stderr, "Le socket a été fermé par le serveur !\n\n");
		close(descripteurSocket);
		return 0;
	default: /* réception de n octets */
		printf("Message reçu du serveur : %s (%d octets)\n\n", messageRecu,
lus);
	}
	//--- Fin de l'étape n°3 !
	// On ferme la ressource avant de quitter
	close(descripteurSocket);
	return 0;
}
```

Pour tester ce client, on peut démarrer un serveur ***netcat*** sur le port 5000 : ***nc -l -p 5000*** puis, on exécute le client : $ ***./ExClientTCP***

Dans l'exemple ci-dessus, la fonction ***close()*** a été utilisée pour fermer le socket et donc la communication. En TCP, la communication étant bidirectionnelle, il est possible de fermer plus finement l'échange en utilisant l'appel ***shutdown().***

La fonction *shutdown*

La fonction ***shutdown*** permet de terminer tout ou une partie d'une communication bidirectionnelle sur un socket.

Syntaxe :

```
#include <sys/socket.h>
int shutdown(int s, int how);
```

- **s :** descripteur du socket
- **how :**

✓ Si la valeur est **SHUT_RD** la réception est désactivée**.**
✓ Si la valeur est **SHUT_WR** l'émission est désactivée**.**
✓ Si la valeur est **SHUT_RDWR** l'émission et la réception sont désactivées**.**

Réalisation d'un serveur TCP :

Un serveur TCP a lui aussi besoin de créer un socket SOCK_STREAM dans le domaine PF_INET. Mis à part cela, le code source d'un serveur TCP basique est très différent d'un client TCP dans le principe. On rappelle qu'un serveur TCP attend des demandes de connexion en provenance de processus client. Le processus client doit connaître au moment de la connexion le numéro de port d'écoute du serveur.

Détails des fonctions :

Étape 1 : La fonction socket

Idem que pour un client TCP

Étape 2 : La fonction bind

Une fois le socket créé, il faut le lier à un point de communication local défini par une adresse IP et un port, c'est le rôle de la fonction ***bind.***

Syntaxe :

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr* addr,socklen_t addrlen);
```

- **sockfd** représente le descripteur du socket nouvellement créé
- **addr** est une structure qui spécifie l'adresse locale à travers laquelle le programme doit communiquer. Le format de l'adresse est fortement dépendant du protocole utilisé.
- **addrlen** indique la taille du champ *addr*. On utilise généralement *sizeof(addr)*.

Quand un socket est créé avec l'appel système socket, il existe dans l'espace des noms mais n'a pas de nom assigné. La fonction ***bind*** affecte l'adresse spécifiée dans addr au socket référencé par le descripteur de fichier sockfd. addrlen indique la taille, en octets, de la structure d'adresse pointée par addr.
Traditionnellement cette opération est appelée « affectation d'un nom à une socket ». Il est normalement nécessaire d'affecter une adresse locale avec bind avant qu'un socket SOCK_STREAM puisse recevoir des connexions. Les règles d'affectation de nom varient suivant le domaine de communication.
Cette fonction renvoie 0 en cas de succès et -1 sinon.
De même que pour un client TCP, il faut initialiser une structure ***sockaddr_in*** avec les informations locales du serveur (adresse IPV4 et numéro de port). Pour cela les fonctions suivantes sont nécessaires :

- ✓ ***htonl*** : pour convertir une adresse IP (sur 32 bits) depuis l'ordre des octets de l'hôte vers celui du réseau
- ✓ ***htons*** : pour convertir le numéro de port (sur 16 bits) depuis l'ordre des octets de l'hôte vers celui du réseau.

⚠ Normalement il faudrait indiquer l'adresse IPv4 de l'interface locale du serveur qui acceptera les demandes de connexions. Il est possible de préciser avec ***INADDR_ANY*** que toutes les interfaces locales du serveur accepteront les demandes de connexion des clients.

Ex1ServeurTCP :

```
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h> /* pour memset */
#include <netinet/in.h> /* pour struct sockaddr_in */
#include <arpa/inet.h> /* pour htons et inet_aton */
#include <unistd.h> /* pour sleep */
#define PORT IPPORT_USERRESERVED // = 5000
int main()
{
	int socketEcoute;
	struct sockaddr_in pointDeRencontreLocal;
	socklen_t longueurAdresse;
	// Crée un socket de communication
	socketEcoute = socket(PF_INET, SOCK_STREAM, 0); /* 0 indique que l'on
utilisera le
                                                    protocole par défaut associé à
SOCK_STREAM soit TCP */
	// Teste la valeur renvoyée par l'appel système socket()
	if (socketEcoute < 0) /* échec ? */
	{
		perror("socket"); // Affiche le message d'erreur
		exit(-1); // On sort en indiquant un code erreur
	}
	printf("Socket créée avec succès ! (%d)\n", socketEcoute);
	// On prépare l'adresse d'attachement locale
	longueurAdresse = sizeof(struct sockaddr_in);
	memset(&pointDeRencontreLocal, 0x00, longueurAdresse);
	pointDeRencontreLocal.sin_family = PF_INET;
```

```
	pointDeRencontreLocal.sin_addr.s_addr = htonl(INADDR_ANY); // toutes les
interfaces locales disponibles
	pointDeRencontreLocal.sin_port = htons(PORT); // = 5000
	// On demande l'attachement local du socket
	if ((bind(socketEcoute, (struct sockaddr*)&pointDeRencontreLocal,
longueurAdresse)) < 0)
	{
		perror("bind");
		exit(-2);
	}
	printf("Socket attachée avec succès !\n");
	// On s'endort ... (cf. test)
	sleep(2);
	// On ferme la ressource avant de quitter
	close(socketEcoute);
	return 0;
}
```

Remarque :

Un numéro de port identifie un processus communiquant ! Si on exécute deux fois le même serveur l'attachement local au numéro de port 5000 du deuxième processus échoue car ce numéro de port est déjà attribué par le système d'exploitation au premier processus serveur.

Étape 3 : mise en attente des connexions *: La fonction listen :*

Le serveur après avoir attaché (bind) un socket d'écoute, doit la placer en attente passive pour être capable d'accepter les demandes de connexion des processus clients. Cette fonction ne s'utilise qu'en mode connecté (donc avec le protocole TCP)

Syntaxe :

```
#include <sys/types.h>
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

- **sockfd** : représente le descripteur du socket nouvellement créé
- **backlog :** définit une longueur maximale pour la file d'attente des connexions

La fonction *listen* retourne 0 en cas de succès et -1 sinon.

Si la file de connexions est pleine le serveur se retrouve dans la situation DOS (Deny Of Service). Il ne peut plus traiter les nouvelles connexions.

Étape 4 : accepter les demandes de connexions *: La fonction accept :*

La fonction accepte permet au serveur d'accepter les demandes de connexion des processus clients.

Syntaxe :

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr* adresse, socklen_t* longueur);
```

- **sockfd** : descripteur du socket précédemment ouvert.
- **adresse** : représente un tampon destiné à stocker l'adresse de l'appelant
- **longueur** : taille de l'adresse de l'appelant.

La fonction *accept* retourne un identifiant du socket de réponse en cas de succès et la valeur *INVALID_SOCKET (-1)* en cas d'erreur.

⚠ *Si un client se connecte au socket d'écoute, la fonction accept retourne l'identifiant d'un socket de dialogue. Le socket d'écoute peut alors accepter des nouvelles connexions.*

Étape 5 : dialoguer avec le client

Pour dialoguer avec un client, les fonctions *read/write* ou *recv/send* vues précédemment seront utilisées.

Exemple d'un serveur TCP mono-client (un client à la fois) :

```
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h> /* pour memset */
#include <netinet/in.h> /* pour struct sockaddr_in */
#include <arpa/inet.h> /* pour htons et inet_aton */
#include <unistd.h> /* pour sleep */
#define PORT IPPORT_USERRESERVED // = 5000
#define LG_MESSAGE 256
int main()
{
	int socketEcoute;
	struct sockaddr_in pointDeRencontreLocal;
	socklen_t longueurAdresse;
	int socketDialogue;
	struct sockaddr_in pointDeRencontreDistant;
	char messageEnvoi[LG_MESSAGE]; /* le message de la couche Application ! */
	char messageRecu[LG_MESSAGE];  /* le message de la couche Application ! */
	int ecrits, lus;  /* nb d'octets ecrits et lus */
	int retour;
	// Crée un socket de communication
	socketEcoute = socket(PF_INET, SOCK_STREAM, 0); /* 0 indique que l'on
utilisera TCP */
	// Teste la valeur renvoyée par l'appel système socket()
	if (socketEcoute < 0) /* échec ? */
	{
		perror("socket"); // Affiche le message d'erreur
		exit(-1);         // On sort en indiquant un code erreur
	}
	printf("Socket créé avec succès ! (%d)\n", socketEcoute);
	// On prépare l'adresse d'attachement locale
	longueurAdresse = sizeof(struct sockaddr_in);
	memset(&pointDeRencontreLocal, 0x00, longueurAdresse);
	pointDeRencontreLocal.sin_family = PF_INET;
	pointDeRencontreLocal.sin_addr.s_addr = htonl(INADDR_ANY); // toutes les
interfaces locales disponibles
	pointDeRencontreLocal.sin_port = htons(PORT);
```

```
	// On demande l'attachement local au socket
	if ((bind(socketEcoute, (struct sockaddr*)&pointDeRencontreLocal,
longueurAdresse)) < 0)
	{
		perror("bind");
		exit(-2);
	}
	printf("Socket attaché avec succès !\n");
	// On fixe la taille de la file d'attente à 5 (pour les demandes de
connexion non encore traitées)
	if (listen(socketEcoute, 5) < 0)
	{
		perror("listen");
		exit(-3);
	}
	printf("Socket placé en écoute passive ...\n");
	// boucle d'attente de connexion : en théorie, un serveur attend
indéfiniment !
	while (1)
	{
		memset(messageEnvoi, 0x00, LG_MESSAGE * sizeof(char));
		memset(messageRecu, 0x00, LG_MESSAGE * sizeof(char));
		printf("Attente d'une demande de connexion (quitter avec Ctrl-
C)\n\n");
		// c'est un appel bloquant
		socketDialogue = accept(socketEcoute, (struct
sockaddr*)&pointDeRencontreDistant, &longueurAdresse);
		if (socketDialogue < 0)
		{
			perror("accept");
			close(socketDialogue);
			close(socketEcoute);
			exit(-4);
		}
		// On réceptionne les données du client
		lus = read(socketDialogue, messageRecu, LG_MESSAGE * sizeof(char));
// ici appel bloquant
		switch (lus)
		{
			case -1: /* une erreur ! */
				perror("read");
				close(socketDialogue);
				exit(-5);
			case 0: /* socket fermé */
				fprintf(stderr, "Le socket a été fermé par le client
!\n\n");
				close(socketDialogue);
				return 0;
			default: /* réception de n octets */
				printf("Message reçu : %s (%d octets)\n\n",
messageRecu, lus);
		}
		// On envoie des données vers le client
		sprintf(messageEnvoi, "ok\n");
		ecrits = write(socketDialogue, messageEnvoi, strlen(messageEnvoi));
		switch (ecrits)
		{
		case -1: /* une erreur ! */
			perror("write");
			close(socketDialogue);
			exit(-6);
```

```
                case 0: /*socket fermé */
                        fprintf(stderr, "Le socket a été fermé par le client !\n\n");
                        close(socketDialogue);
                        return 0;
                default: /* envoi de n octets */
                        printf("Message %s envoyé (%d octets)\n\n", messageEnvoi,
ecrits);
                }
                // On ferme le socket de dialogue et on se replace en attente ...
                close(socketDialogue);
        }

        // On ferme la ressource avant de quitter
        close(socketEcoute);
        return 0;
}
```

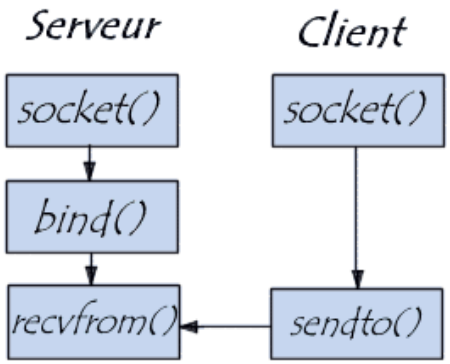
Remarque :

Un problème apparaît pour le serveur : comment dialoguer avec le client connecté et continuer à attendre des nouvelles connexions ? Il y a plusieurs solutions à ce problème notamment la programmation multi-tâche car ici le serveur a besoin de paralléliser plusieurs traitements.

b) Programmation UDP :

La mise en œuvre d'une communication client/serveur UDP se fait à l'aide d'un socket UDP. L'échange est semblable à celui d'une communication client/serveur TCP.

Rappel du schéma d'un échange UDP :



Réalisation d'un client UDP :

Étape 1 : création du socket : La fonction socket :

Idem que pour un client TCP.

Étape 2 : envoi d'une requête au serveur : La fonction sendto :

La fonction ***sendto*** permet de transmettre un message à un autre socket.

Syntaxe :

```
	#include <sys/types.h>
	#include <sys/socket.h>
	ssize_t sendto(int s, const void* buf, size_t len, int flags,const struct sockaddr* to, socklen_t tolen);
```

- **s:** descripteur du socket émetteur
- buf: message à émettre
- len: longueur du message
- flags: représente le type d'envoi
  - ✓ *MSG_DONTROUTE* est utilisé pour empêcher la transmission d'un paquet vers une passerelle, n'envoyer de données que vers les hôtes directement connectés au réseau. Ceci n'est normalement employé que par les programmes de diagnostique ou de routage. Cette option n'est définie que pour les familles de protocoles employant le routage, pas les sockets par paquets.
  - ✓ *MSG_OOB* indiquera que les données urgentes (*Out Of Board*) doivent être envoyées
  - ✓ *MSG_DONTWAIT* active le mode non-bloquant. Une opération qui devrait bloquer renverra EAGAIN à la place.
  - ✓ *MSG_NOSIGNAL* demande de ne pas envoyer de signal SIGPIPE d'erreur sur les sockets connectés lorsque le correspondant coupe la connexion. L'erreur EPIPE est toutefois renvoyée.
  - ✓ *0* indique un envoi normal.
- to: adresse de la cible
- tolen: taille de l'adresse de la cible

La fonction *sendto* retourne le nombre d'octets effectivement envoyés.

Exemple d'un client UDP:

```
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h> /* pour memset */
#include <netinet/in.h> /* pour struct sockaddr_in */
#include <arpa/inet.h> /* pour htons, htonl et inet_aton */
#define LG_MESSAGE 256
int main()
{
	int descripteurSocket;
	struct sockaddr_in pointDeRencontreDistant;
	socklen_t longueurAdresse;
	char messageEnvoi[LG_MESSAGE]; /* le message de la couche Application ! */
	int ecrits; /* nb d'octets ecrits */
	int retour;
	// Crée un socket de communication
	descripteurSocket = socket(PF_INET, SOCK_DGRAM, 0); /* 0 indique que l'on utilisera UDP */
	// Teste la valeur renvoyée par l'appel système socket()
	if (descripteurSocket < 0) /* échec ? */
	{
		perror("socket"); // Affiche le message d'erreur
```

```
		exit(-1); // On sort en indiquant un code erreur
	}
	printf("Socket créé avec succès ! (%d)\n", descripteurSocket);
	// On prépare l'adresse d'attachement locale
	longueurAdresse = sizeof(struct sockaddr_in);

	// Obtient la longueur en octets de la structure sockaddr_in
	longueurAdresse = sizeof(pointDeRencontreDistant);
	// Initialise à 0 la structure sockaddr_in
	memset(&pointDeRencontreDistant, 0x00, longueurAdresse);
	// Renseigne la structure sockaddr_in avec les informations du serveur
distant
	pointDeRencontreDistant.sin_family = PF_INET;
	// On choisit le numéro de port d'écoute du serveur
	pointDeRencontreDistant.sin_port = htons(IPPORT_USERRESERVED); // = 5000
	// On choisit l'adresse IPv4 du serveur
	inet_aton("192.168.52.2", &pointDeRencontreDistant.sin_addr); // à modifier
selon ses besoins
	// Initialise à 0 le message
	memset(messageEnvoi, 0x00, LG_MESSAGE * sizeof(char));
	// Envoie un message au serveur
	sprintf(messageEnvoi, "Hello world !\n");
	ecrits = sendto(descripteurSocket, messageEnvoi, strlen(messageEnvoi), 0,
(struct
		sockaddr*)&pointDeRencontreDistant, longueurAdresse);
	switch (ecrits)
	{
	case -1: /* une erreur ! */
		perror("sendto");
		close(descripteurSocket);
		exit(-3);
	case 0:
		fprintf(stderr, "Aucune donnée n'a été envoyée !\n\n");
		close(descripteurSocket);
		return 0;
	default: /* envoi de n octets */
		if (ecrits != strlen(messageEnvoi))
			fprintf(stderr, "Erreur dans l'envoie des données !\n\n");
		else
			printf("Message %s envoyé avec succès (%d octets)\n\n",
messageEnvoi, ecrits);
	}
	// On ferme la ressource avant de quitter
	close(descripteurSocket);
	return 0;
}
```

Réalisation d'un serveur UDP :

Le code source d'un serveur UDP basique est très similaire à celui d'un client UDP. Évidemment, un serveur UDP a lui aussi besoin de créer un socket SOCK_DGRAM dans le domaine PF_INET. Puis, il doit utiliser l'appel système ***bind*** pour lier son socket d'écoute à une interface et à un numéro de port local à sa machine car le processus client doit connaître et fournir au moment de l'échange ces informations.

Étape 1 : création du socket : La fonction socket :

Idem que pour un client TCP.

Étape 2 : attachement au socket : la fonction bind :

La fonction ***bind*** est la même que pour un serveur TCP

Étape 3 : réception de la requête du client : La fonction recvfrom :

La fonction *recvfrom* permet de recevoir un message sur un socket.

Syntaxe :

```
#include <sys/types.h>
#include <sys/socket.h>
   ssize_t recvfrom(int s, void* buf, size_t len, int flags,struct sockaddr*
      from, socklen_t* fromlen);
```

- **s :** représente le descripteur du socket précédemment ouvert.
- **buf :** représente un tampon qui recevra les octets envoyés par le client.
- **len :** indique le nombre d'octets à lire.
- **flags :** correspond au type de lecture à adopter :
  - ✓ **MSG_OOB :** permet la lecture des données hors bande qui ne seraient autrement pas placées dans le flux de données normales. Certains protocoles placent ces données hors bande en tête de la file normale, et cette option n'a pas lieu d'être dans ce cas.
  - ✓ **MSG_PEEK :** permet de lire les données en attente dans la file sans les enlever de cette file. Ainsi une lecture ultérieure renverra à nouveau les mêmes données.
  - ✓ **MSG_WAITALL :** demande que l'opération de lecture soit bloquée jusqu'à ce que la requête complète soit satisfaite. Toutefois la lecture peut renvoyer quand même moins de données que prévu si un signal est reçu, ou si une erreur ou une déconnexion se produisent.
  - ✓ **MSG_NOSIGNAL :** désactive l'émission de SIGPIPE sur les sockets connectés dont le correspondant disparaît.
  - ✓ **MSG_ERRQUEUE :** demande de lire les erreurs provenant de la file d'erreur du socket. Les erreurs sont transmises dans un message dont le type dépend du protocole (IP_RECVERR pour IPv4). Il faut alors fournir un buffer de taille suffisante.
  - ✓ **La valeur 0 :** indique une lecture normale.
- **from :** correspond à l'adresse d'une structure qui contiendra l'adresse de l'émetteur.
- **fromlen :** adresse d'un entier qui indique la taille de la structure de l'adresse de l'émetteur.

La fonction *recvfrom* renvoie le nombre d'octets lus en cas de succès ou -1 en cas d'erreur. De plus cette fonction bloque le processus jusqu'à la réception des données. La valeur de retour sera 0 si la connexion a été fermée normalement.

Exemple d'un serveur UDP :

```
//Serveur udp
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h> /* pour memset */
#include <netinet/in.h> /* pour struct sockaddr_in */
#include <arpa/inet.h> /* pour htons, htonl et inet_aton */
#define PORT IPPORT_USERRESERVED // = 5000
#define LG_MESSAGE 256
```

```c
int main()
{
	int descripteurSocket;
	struct sockaddr_in pointDeRencontreLocal;
	struct sockaddr_in pointDeRencontreDistant;
	socklen_t longueurAdresse;
	char messageRecu[LG_MESSAGE]; /* le message de la couche Application ! */
	int lus; /* nb d'octets lus */
	int retour;
	// Crée un socket de communication
	descripteurSocket = socket(PF_INET, SOCK_DGRAM, 0);
	// Teste la valeur renvoyée par l'appel système socket()
	if (descripteurSocket < 0) /* échec ? */
	{
		perror("socket"); // Affiche le message d'erreur
		exit(-1); // On sort en indiquant un code erreur
	}
	printf("Socket créé avec succès ! (%d)\n", descripteurSocket);
	// On prépare l'adresse d'attachement locale
	longueurAdresse = sizeof(struct sockaddr_in);
	memset(&pointDeRencontreLocal, 0x00, longueurAdresse);
	pointDeRencontreLocal.sin_family = PF_INET;
	pointDeRencontreLocal.sin_addr.s_addr = htonl(INADDR_ANY); // n'importe
quelle interface locale disponible
	pointDeRencontreLocal.sin_port = htons(PORT); // <- 5000
	// On demande l'attachement local de la socket
	if ((bind(descripteurSocket, (struct sockaddr*)&pointDeRencontreLocal,
longueurAdresse))< 0)
	{
		perror("bind");
		exit(-2);
	}
	printf("Socket attaché avec succès !\n");
	// Obtient la longueur en octets de la structure sockaddr_in
	longueurAdresse = sizeof(pointDeRencontreDistant);
	// Initialise à 0 la structure sockaddr_in (c'est l'appel recvfrom qui
remplira cette structure)
	memset(&pointDeRencontreDistant, 0x00, longueurAdresse);
	// Initialise à 0 le message
	memset(messageRecu, 0x00, LG_MESSAGE * sizeof(char));
	// Réceptionne un message du client
	lus = recvfrom(descripteurSocket, messageRecu, sizeof(messageRecu), 0,
(struct sockaddr   *)&pointDeRencontreDistant, &longueurAdresse);
	switch (lus)
	{
	case -1: /* une erreur ! */
		perror("recvfrom");
		close(descripteurSocket);
		exit(-3);
	case 0:
		fprintf(stderr, "Aucune donnée n'a été reçue !\n\n");
		close(descripteurSocket);
		return 0;
	default: /* réception de n octets */
		printf("Message %s reçu avec succès (%d octets)\n\n", messageRecu,
lus);
	}
	// On ferme la ressource avant de quitter
	close(descripteurSocket);
	return 0;
}
```

III/ Programmation IPv6

La programmation des sockets reste identique à la programmation IPv4 mais avec une structure d'adresse adaptée pour IPv6. Pour l'appel socket() il faudra utiliser PF_INET6 ou AF_INET6.

```
struct in6_addr {
	union {
		__u8 u6_addr8[16];
		__be16 u6_addr16[8];
		__be32 u6_addr32[4];
	} in6_u;
#define s6_addr in6_u.u6_addr8
#define s6_addr16 in6_u.u6_addr16
#define s6_addr32 in6_u.u6_addr32
};
struct sockaddr_in6 {
	unsigned short int sin6_family; /* AF_INET6 */
	__be16 sin6_port; /* Transport layer port # */
	__be32 sin6_flowinfo; /* IPv6 flow information */
	struct in6_addr sin6_addr; /* IPv6 address */
	__u32 sin6_scope_id; /* scope id (new in RFC2553) */
};
```

Comme pour les IPV4 on a les 3 champs de sockaddr_in6 qui permettent de bien construire l'adresse d'un socket :

1. **sin6_family**, valant AF_INET6
2. **sin6_addr**, l'adresse IP sur 6 octets
3. **sin6_port**, pour le numéro de port.

Les octets de l'adresse IP et le numéro de port sont stockés dans l'ordre réseau (big-endian); cela n'est pas forcément le cas de toutes les machines hôtes sur laquelle s'exécuteront tous les programmes s'appuyant sur les sockets internet d'où la nécessité de faire certaines conversions et adaptations pour construire une adresse de socket.

Remarque :

La nouvelle structure d'adresse sockaddr_in6 a une longueur de 28 octets. Elle est donc plus longue que le type générique sockaddr. Pour corriger cela, une nouvelle structure sockaddr_storage a été définie, de taille suffisante afin de pouvoir prendre en compte tous les protocoles supportés. Il y a aussi des nouvelles fonctions de conversion :

- ***getaddrinfo()*** (équivalent à gethostbyname(), gethostbyaddr(), getservbyname(), getservbyport())

- ***getnameinfo()*** (équivalent à gethostbyaddr() et getservbyport())

- ***inet_pton()*** et ***inet_ntop()*** (équivalent à inet_addr() et inet_ntoa())

Remarques :

- ✓ On appellera getaddrinfo() avant la création de la socket avec socket() ;
- ✓ L'adresse wildcard : IN6ADDR_ANY_INIT (équivalent à INADDR_ANY) ;
- ✓ L'adresse de loopback : IN6ADDR_LOOPBACK_INIT (équivalent à INADDR_LOOPBACK).

Exemple :

```
#include <sys/types.h>
#include <sys/socket.h>
// Un socket en mode connecte
int socket_tcp = socket(PF_INET6, SOCK_STREAM, 0); //par defaut TCP
// Un socket en mode non connecte
int socket_udp = socket(PF_INET6, SOCK_DGRAM, 0); //par defaut UDP
// Un socket en mode raw
int socket_icmp = socket(PF_INET, SOCK_RAW, IPPROTO_ICMPV6); //on choisit ICMPv6
```

1. Comparaison IPV4 / IPV6

Constante	Signification	Usage principal	Valeur numérique (souvent)
AF_INET	*Address Family Internet* (IPv4)	Utilisée dans les structures d'adresses (`sockaddr_in`) pour indiquer une adresse IPv4	2
PF_INET	*Protocol Family Internet* (IPv4)	Utilisée dans la création de sockets (`socket(PF_INET, ...)`) pour indiquer la famille de protocoles Internet	2
AF_INET6	*Address Family Internet v6* (IPv6)	Utilisée dans les structures d'adresses (`sockaddr_in6`) pour indiquer une adresse IPv6	10
PF_INET6	*Protocol Family Internet v6* (IPv6)	Utilisée dans la création de sockets (`socket(PF_INET6, ...)`) pour indiquer la famille de protocoles Internet v6	10

2. Concept de Dual-Stack

Dual-Stack permet à un hôte ou serveur de supporter simultanément IPv4 et IPv6.

- **Avantages** :
  - Compatibilité avec les anciens réseaux IPv4.
  - Transition progressive vers IPv6 sans rupture de service.
- **Fonctionnement** :
  - Permet à deux piles réseau de coexister.
  - Permet au système de choisir automatiquement la meilleure pile (souvent IPv6 si disponible).

3. Exemple complet utilisant le concept Dual-Stack :

Seveur TCP :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

#define PORT "8080"
#define BUF_SIZE 1024

int main() {
   struct addrinfo hints, *res, *p;
   int sockfd, newfd;
   char buffer[BUF_SIZE];
   memset(&hints, 0, sizeof(hints));
   hints.ai_family = AF_UNSPEC;     // IPv4 ou IPv6
   hints.ai_socktype = SOCK_STREAM; // TCP
   hints.ai_flags = AI_PASSIVE;     // Pour bind()
   if (getaddrinfo(NULL, PORT, &hints, &res) != 0) {
      perror("getaddrinfo");
      exit(EXIT_FAILURE);
   }
   // Essai de création et bind sur chaque résultat
   for (p = res; p != NULL; p = p->ai_next) {
      sockfd = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
      if (sockfd == -1) continue;
```

```
    if (bind(sockfd, p->ai_addr, p->ai_addrlen) == 0) break;
    close(sockfd);
  }
  if (p == NULL) {
    fprintf(stderr, "Impossible de binder le socket\n");
    exit(EXIT_FAILURE);
  }
  listen(sockfd, 5);
  printf("Serveur en écoute sur le port %s (IPv4/IPv6).\n", PORT);
  // Accepter une connexion
  newfd = accept(sockfd, NULL, NULL);
  if (newfd == -1) { perror("accept"); exit(EXIT_FAILURE); }
  // Réception
  int n = recv(newfd, buffer, BUF_SIZE-1, 0);
  if (n > 0) {
    buffer[n] = '\0';
    printf("Message reçu: %s\n", buffer);
    // Réponse
    const char *reply = "Bonjour, message bien reçu !";
    send(newfd, reply, strlen(reply), 0);
  }
  close(newfd);
  close(sockfd);
  freeaddrinfo(res);
  return 0;
}
```

Client TCP :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

#define BUF_SIZE 1024

int main(int argc, char *argv[]) {
   if (argc != 3) {
      fprintf(stderr, "Usage: %s <hostname> <port>\n", argv[0]);
      exit(EXIT_FAILURE);
   }
   const char *hostname = argv[1];
   const char *port = argv[2];
   struct addrinfo hints, *res, *p;
   int sockfd;
   char buffer[BUF_SIZE];
   memset(&hints, 0, sizeof(hints));
   hints.ai_family = AF_UNSPEC;     // IPv4 ou IPv6
   hints.ai_socktype = SOCK_STREAM; // TCP
   if (getaddrinfo(hostname, port, &hints, &res) != 0) {
      perror("getaddrinfo");
      exit(EXIT_FAILURE);    }
```

```
  // Essai de connexion
  for (p = res; p != NULL; p = p->ai_next) {
     sockfd = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
     if (sockfd == -1) continue;
     if (connect(sockfd, p->ai_addr, p->ai_addrlen) == 0) break;
     close(sockfd);    }
  if (p == NULL) {
     fprintf(stderr, "Impossible de se connecter\n");
     exit(EXIT_FAILURE);    }
  // Envoi
  const char *msg = "Hello serveur, voici un message du client !";
  send(sockfd, msg, strlen(msg), 0);
  // Réception
  int n = recv(sockfd, buffer, BUF_SIZE-1, 0);
  if (n > 0) {
     buffer[n] = '\0';
     printf("Réponse du serveur: %s\n", buffer);    }
  close(sockfd);
  freeaddrinfo(res);
  return 0;
}
```

https://web.maths.unsw.edu.au/~lafaye/CCM/sockets/sockfonc.htm

https://lps.cofares.net/Sockets/