Dr. BAHI N.

Correction série TP1

Université Mohamed Khider Biskra
Faculté SESNV
Département d'informatique

Niveau: M1 **Module: SD** **Année: 2021-2022**

TP n° (1): Notions fondamentales sur l'architecture Client/Serveur et les sockets.

Partie 1: Architecture Client/Serveur.

1. Définir l'architecture client/serveur.

L'architecture **client/serveur** désigne un mode de communication entre plusieurs ordinateurs d'un réseau qui distingue un ou plusieurs postes clients du serveur : chaque logiciel client peut envoyer des requêtes à un serveur. Un serveur peut être spécialisé en serveur d'applications, de fichiers, de terminaux, ou encore de messagerie électronique.

2. Les deux schémas suivants représentent respectivement vu du client et vu du serveur. Expliquer-les.

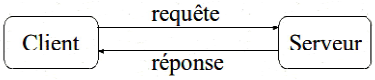


Schéma 1

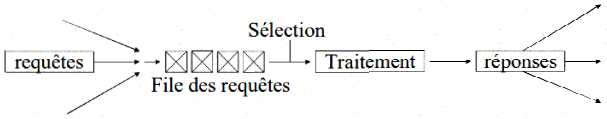


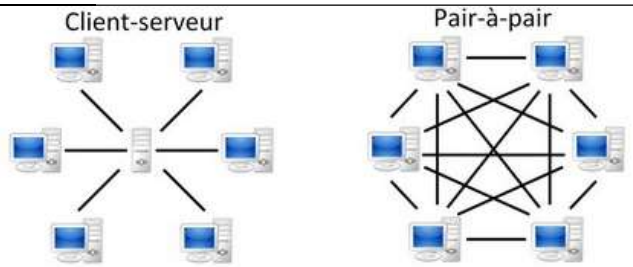
Schéma 2

Réponse:

Vu du serveur	**Vu du client**
Il est passif (ou maître) ; Il est à l'écoute, prêt à répondre aux requêtes envoyées par des clients ; Gestion des requêtes (priorité) ; Dès qu'une requête lui parvient, il la traite et envoie une réponse ; Exécution du service (séquentielle, concurrent) ; Mémorisation ou non l'état du client.	Il est actif (ou esclave) ; Il envoie des requêtes au serveur ; Il attend et reçoit les réponses du serveur.

3. Définir un réseau pair à pair ou (peer-to-peer - P2P)?

Réponse: Un réseau pair à pair est un modèle d'échange en réseau où chaque entité est à la fois client et serveur, contrairement au modèle client-serveur. La particularité des architectures pair-à-pair réside dans le fait que les échanges peuvent se faire directement entre deux ordinateurs connectés au système, sans transiter par un serveur central.



4. Signifient quoi les termes suivants:

a. Protocoles sans état

Réponse:

- Chaque requête est indépendante de la suivante: Un protocole sans état est un protocole de communication qui n'enregistre pas l'état d'une session de communication entre deux requêtes successives.
- Pas de causalité entre les requêtes: La communication est formée de paires requête-réponse indépendantes et chaque paire requête-réponse est traitée comme une transaction indépendante, sans lien avec les requêtes précédentes ou suivantes.
- Pas d'ordre d'arrivé.
- Un protocole sans état ne nécessite pas que le serveur conserve, au cours de la session de communication, l'état de chacun des partenaires. Donc, il n y a aucune nécessité pour gérer l'espace mémoire requis pour enregistrer l'état des échanges en cours. Autrement dit, si une session cliente meurt à mi-transaction, aucune partie du système n'est tenue de procéder au nettoyage de l'état en cours du serveur.
- L'inconvénient majeur de ce protocole réside dans la nécessité d'inclure des informations supplémentaires dans chaque requête, et ces informations supplémentaires doivent être interprétées par le serveur.

b. Protocole à état

Réponse:

- Certaines requêtes dépendent des autres pour un même client.
- Le serveur doit mémoriser la connexion du client.

c. TCP/IP

Réponse: Grâce au protocole TCP, les applications peuvent communiquer de façon sûre (grâce au système d'accusés de réception du protocole TCP), indépendamment des couches inférieures. Lors d'une communication à travers le

protocole TCP, les deux machines doivent établir une connexion. La machine émettrice (celle qui demande la connexion) est appelée client, tandis que la machine réceptrice est appelée serveur. On dit qu'on est alors dans un environnement Client-Serveur. Les machines dans un tel environnement communiquent en mode connecté, c'est-à-dire que la communication se fait dans les deux sens. TCP/IP ne dépend aucunement du système d'exploitation. Ainsi, différents systèmes d'exploitation peuvent communiquer ensemble.

d. UDP

Réponse: UDP est un protocole permettant l'envoi sans connexion de datagrammes dans des réseaux basés sur le protocole IP. Il permet une communication (transmission des données) rapide, sans délai car il n'établit pas de connexion. Ceci résulte également du fait que la perte de paquets individuels impacte uniquement la qualité de la transmission. En cas de connexion TCP, il est en revanche procédé automatiquement à une nouvelle demande des paquets perdus, ce qui bloque l'intégralité de la machine de transmission. Donc, UDP n'offre aucune garantie quant à la sécurité et à l'authenticité des données: le fait de renoncer à l'authentification mutuelle de l'expéditeur et du destinataire permet au protocole UDP d'assurer une vitesse de transmission exceptionnelle.

e. Port

Réponse:
Dans le cas général, un port est un point d'entrée à un service sur un équipement (pc, serveur,...) connecté à un réseau.
On a 2 définitions pour un port en tant que nom.
Physiquement, en tant que composant matériel externe ou interne dans lequel des connexions câblées sont branchées afin d'établir des lignes de communication et de transfert de données. Dans ce cas, le port fournit une interface physique spécifique entre les périphériques.
Un port virtuel, par contre, fait référence à la contrepartie en ligne ou au point de destination pour le transfert de données. Également appelé port réseau, ce port identifie un point où des données ou des informations sont envoyées.
Les ports virtuels et physiques sont nécessaires pour établir et maintenir un réseau.
Les ports de communication utilisent les protocoles TCP ou UDP pour communiquer.

5. Peuvent le serveur et les clients utilisent des protocoles différents ?

Réponse :
Le client et le serveur doivent utiliser le même protocole de communication.

6. Schématiser les différentes étapes des deux protocoles TCP/IP et UDP.

Réponse:

TCP/IP

A. Etablir une connexion entre un client et le serveur (Schéma 3)

a. Serveur ➔ Ouverture passive
i. Ouvrir un point d'accès à une connexion TCP,
ii. Se mettre en attente passive de demandes de connexion d'un autre système (Client).

b. Client ➔ Ouverture active
i. Le client envoie un segment SYN au serveur,
ii. Le serveur lui répond par un segment SYN/ACK,
iii. Le client confirme par un segment ACK.

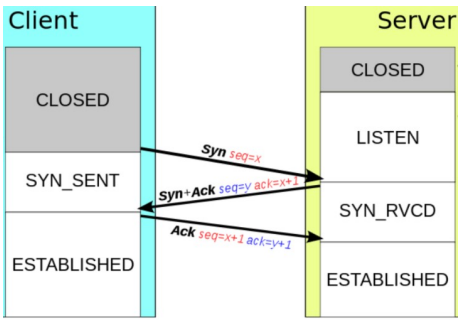


Schéma 3: Etablissement d'une connexion entre un client et le serveur selon le protocole TCP/IP.

Remarques:
- Durant cet échange initial, les numéros de séquence des deux parties sont synchronisés.
- Le client utilise son numéro de séquence initial dans le champ "Numéro de séquence" du segment SYN (x par exemple),
- Le serveur utilise son numéro de séquence initial dans le champ "Numéro de séquence" du segment SYN/ACK (y par exemple) et ajoute le numéro de séquence du client plus un (x+1) dans le champ "Numéro d'acquittement" du segment,
- Le client confirme en envoyant un ACK avec un numéro de séquence augmenté de un (x+1) et un numéro d'acquittement correspondant au numéro de séquence du serveur plus un (y+1).

B. Transfert des données

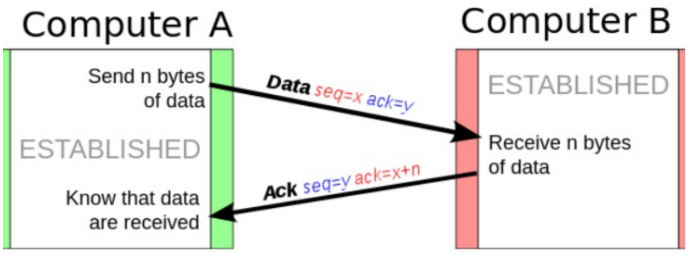


Schéma 4: Phase de transferts de données selon le protocole TCP/IP.

Remarques:

- Les numéros de séquence sont utilisés afin d'ordonner les segments TCP reçus et de détecter les données perdues,
- Les sommes de contrôle permettent la détection d'erreurs,
- Les acquittements ainsi que les temporisations permettent la détection des segments perdus ou retardés.

C. Fin d'une connexion

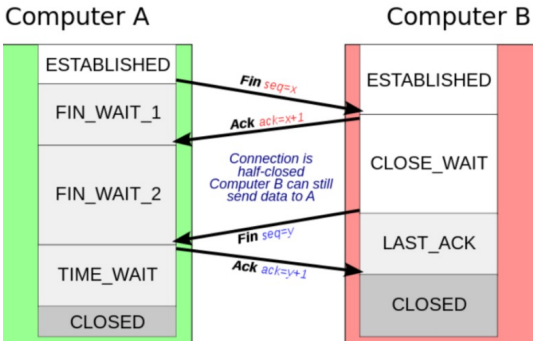


Schéma 5: Phase de terminaison d'une connexion selon le protocole TCP/IP.

Exemple:

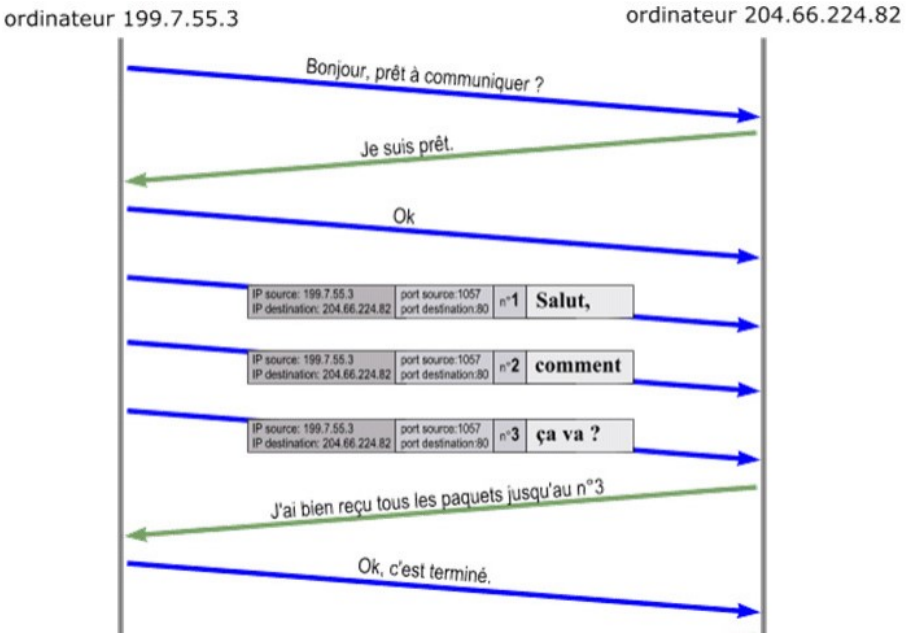


Figure 1: Phase de terminaison d'une connexion selon le protocole TCP/IP.

Solution Partie 2 : Les sockets.

1. Définir les sockets.

Ce concept s'agit approximativement d'une extension de la portée des tubes, pour pouvoir faire dialoguer des processus s'exécutant sur différentes machines. On peut donc écrire des données dans un socket après l'avoir associée à un protocole de communication, et les couches réseau des deux stations s'arrangeront pour que les données ressortent à l'autre extrémité.

2. Pour écrire des codes en langage C réalisant des communications entre machines, on doit suivre des étapes bien définies selon le protocole utilisé (Partie1-Question6). D'abord, on s'intéresse au socket TCP:

b. Citer les bibliothèques nécessaires.

Réponse
```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

c. Donner la structure générale d'une application client-serveur TCP et client/serveur UDP (mode connecté et mode non connecté).

Réponse :
Protocole TCP/IP

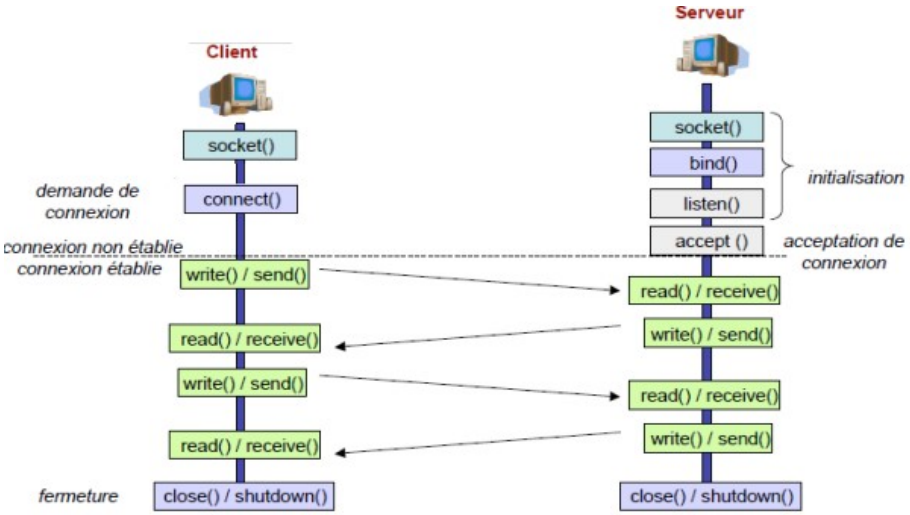


Figure 2: Etablissement d'une connexion entre deux machines (Serveur-Client). Séquence de primitives à invoquer.

Protocole UDP

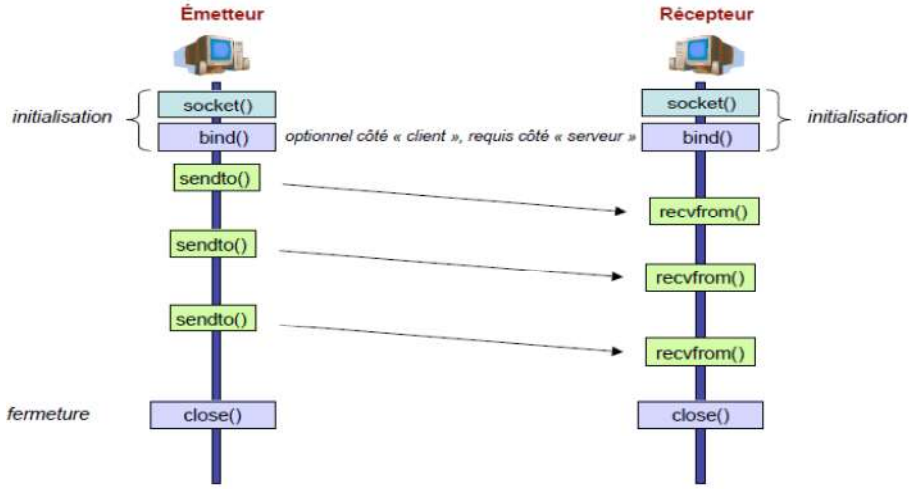


Figure 3: Etablissement d'un échange hors connexion entre deux machines. Séquence de primitives à invoquer.

d. Quelles sont les structures et les appels systèmes/fonctions nécessaires pour l'implémentation ?

Réponse :

1. La première étape consiste à créer une socket. Ceci s'effectue à l'aide de l'appel-système **socket( )**, défini dans **<sys/socket.h>** : **int socket (int domaine, int type, int protocole);**

Le descripteur est supérieur ou égal à zéro, ce qui signifie qu'une valeur de retour négative indique une erreur. **C**ette primitive crée un point de communication, et renvoie un descripteur. On dispose d'un entier permettant de distinguer la socket, mais aucun dialogue réseau n'a pris place. Il n'y a même pas eu d'échange d'informations avec les protocoles de communication: noyau. Celui-ci a simplement accepté de nous attribuer un emplacement dans sa table de sockets. A noter, qu'on appelle ce socket « interface de bienvenue », car il ne sert qu'à accepter des connexions.

Adressage: Avant de pouvoir l'utiliser, il faut identifier le socket, c'est-à-dire définir l'adresse complète de notre extrémité de communication. Le nom affecté au socket doit permettre de la trouver sans ambiguïté en employant le protocole réseau indiqué lors de sa création.

Pour les communications fondées sur le protocole IP, l'identité d'un socket contient l'adresse IP de la machine et le numéro de port employé. En fait, une machine ne devra obligatoirement identifier son socket que si elle doit être jointe par un autre programme. Si le client doit lui-même contacter un serveur, il lui faut connaître l'identité de l'autre

extrémité de la communication, mais l'extrémité locale sera automatiquement identifiée par le noyau.

Pour stocker l'adresse complète d'une socket, on emploie la structure d'adresse générique **sockaddr**, définie dans **<sys/socket.h>**.

```
struct sockaddr
 {
        unsigned short int sa_family; //au choix
        unsigned char sa_data[14]; //en fonction de la famille
 };
```

2. Affectation d'adresse: Ceci s'effectue a l'aide de l'appel-système **bind**( ):

int bind (int sock, struct sockaddr * adresse, socklen_t longueur);

L'usage de la primitive **bind** est essentiel pour un serveur qui est connu à l'extérieur par son numéro de port.

3. Attente d'une connexion (TCP): **int listen(int sock, int LengthFA);** cette primitive place le socket du serveur en mode d'écoute et d'attente de demandes de connexions et précise la longueur de la file d'attente des demandes de connexion.

4. Primitive **accept** : **int accept(int sock, struct sockaddr *sockAddr, int *len);** c'est une attente bloquante d'une demande de connexion de la part d'un client i.e si aucune demande (requête) de connexion n'est arrivée, le serveur (processus appelant) est mis en attente de l'arrivée d'une requête. Lors de la connexion du client, un nouveau socket est créé et son numéro est retourné par la fonction. L'ancien socket de numéro **sock** peut satisfaire d'autres demandes de connexion (on appelle ce socket « interface de bienvenue », car il ne sert qu'à accepter des connexions). **sockAddr** donne en retour l'identité de la machine distante (adresse IP + port, stockés dans une structure dont la taille est également retournée).

5. **int connect(int sock, struct sockaddr *sockAddr, int len);** cette primitive est nécessaire pour établir une connexion, elle est invoquée par le client. Dans un service en mode connecté, cette primitive rend un résultat après l'établissement effectif de la connexion ou l'échec.

6. Envoi (écriture) de données en mode connecté :

int send(int numSock, char * buffer, int nbOctets, int flag);

ou

write()……

7. Réception (lecture) de données en mode connecté:

int receive(int numSock, char * buffer, int tailleMax, int flag);

ou

read()……

8. Envoi de données en mode non connecté:

int sendto(int sock, char * buffer, int nbOctets, int flag, struct sockaddr *sockAddr, int len);

9. Réception de données en mode non connecté:

int recvfrom(int sock, char * buffer, int tailleMax, int flag, struct sockaddr *sockAddr, int *len);

10. Fin de connexion: **int close (int sock);**

e. Expliquer les paramètres des différents appels systèmes en donnant des exemples sur leurs valeurs possibles.

Réponse

int socket (int domaine, int type, int protocole);

Le premier argument de cette routine est le domaine de communication. Il s'agit d'une constante symbolique pouvant prendre plusieurs valeurs. En voici quelques exemples :

AF_INET : protocole fondé sur IP.

AF_Local, AF_UNIX : communication limitée aux processus résidant sur la même machine.

Le second argument est le type de socket. Nous ne considérerons que deux cas :

SOCK_STREAM : le dialogue s'effectue en mode connecté, avec un contrôle de flux d'une extrémité à l'autre de la communication.

SOCK_DGRAM : la communication a lieu sans connexion, par transmission de paquets de données.

Le troisième argument indique le protocole désiré. 0 = protocole par défaut.

La structure **sockaddr: cette** structure contient les membres suivants :

Nom	Type	Signification
sa_family	unsigned short int	Famille de communication
sa_data	char []	Données propres au protocole

En réalité, cette structure est une coquille vide, permettant d'employer un type homogène pour toutes les communications réseau. Pour indiquer véritablement l'identité d'un socket, on utilise une structure dépendant de la famille de communication, puis on emploie une conversion de type **(struct sockaddr *)** lors des appels-système.

Pour les sockets de la famille **AF_INET** (**PF_INET**) reposant sur le protocole IP, la structure utilisée est **sockaddr_in**, définie dans **<netinet/in.h>** :

struct in_addr { **unsigned** int s_addr; }; // une adresse Ipv4 (32 bits)

```
struct sockaddr_in
{
        unsigned short int sin_family; // PF_INET
        unsigned short int sin_port; // numéro de port
        struct in_addr sin_addr; // <- adresse IPv4
        unsigned char sin_zero[8]; // ajustement pour être compatible avec sockaddr
};
```

Nom	Type	Signification
sin_family	short int	Famille de communication AF_INET.
sin_port	unsigned short	Numéro de port, dans l'ordre des octets du réseau.
sin_addr	struct in_addr	Adresse IP de l'interface (dont le membre s_addr est dans l'ordre des octets du réseau).

Pour remplir les champs de la structure **sockaddr_in**, nous emploierons donc la méthode suivante :
e1. Mettre à zéro tout le contenu de l'adresse, à l'aide de la fonction **memset**( ).
e2. Remplir le champ **sin_family** avec **AF_INET**.
e3. Remplir le champ **sin_port** avec le membre **s_port** d'une structure servent renvoyée par **getservbyname( )** ou par **getservbyport( )**.
e4. Remplir le champ si **n_addr** avec le contenu du membre **h_addr** de la structure **hostent** renvoyée par **gethostbyname( )** ou avec le retour de la fonction **inet_aton()**. On convertit explicitement le type **char *** du membre **h_addr** en pointeur sur une structure **in_addr** afin de pouvoir copier son champ **s_addr**, qui est entier.

int bind (int sock, struct sockaddr * adresse, socklen_t longueur);
sock: désigne le descripteur de socket, ***adresse**: un pointeur sur la structure « sockaddr » contenant : l'adresse du site distant et le numéro de port du processus distant. Le dernier argument représente la longueur de l'adresse. Le type socklen_t n'est pas disponible sur tous les Unix. Dans ce cas le troisième argument est un entier (int).

Donc, le socket représenté par le descripteur passé en premier argument est associée à l'adresse passée en seconde position. L'appel-système bind( ) peut échouer en renvoyant-1.

int listen(int sock, int LengthFA); La primitive **listen()** permet à un serveur travaillant en mode sur connexion (TCP/IP) d'indiquer au noyau le nombre de connexions qu'il peut accepter simultanément pour un descripteur de socket. **Listen()** crée une file d'attente de taille limitée dans le système.

int accept(int sock, struct sockaddr *sockAddr, int *len); a les paramètres habituels :
- **sock**, désigne le descripteur du socket ;
- un pointeur sur une structure **sockaddr** comprenant:

l'adresse du site distant,
la porte (le numéro de port) du client (processus distant).

Ces valeurs ne sont pas remplies à l'appel de la primitive, mais à l'arrivée du message de demande d'ouverture. Elle correspond aux étapes « Ouverture passive» jusqu'à l'ouverture réussie (la connexion établie).

6. **int connect(int sock, struct sockaddr *sockAddr, int len);** Cette primitive consiste à demander d'établir une connexion à un processus serveur distant dont l'adresse est passée en paramètre. Elle a comme la primitive bind() les paramètres suivants:
sock, désigne le descripteur de socket;
Un pointeur sur la structure « sockaddr » contenant:
* l'adresse du site distant,
* la porte (numéro de port) du processus distant.

La primitive crée un nouveau descripteur de socket identique à celui référencé par **sock** (contenant les mêmes informations) de manière à ce que le serveur puisse, en créant un processus fils, traiter les requêtes (messages) venant sur la connexion en utilisant le descripteur créé et à ce que le processus père puisse venir se mettre en attente d'une nouvelle demande d'ouverture (serveur concurrent) avec le descripteur initial.

Dans le mode de service sur connexion (TCP), les primitives **accept()** et **connect()** se correspondent pour compléter le contenu des deux descripteurs de sockets.

// On rappelle qu'une communication TCP est bidirectionnelle « full duplex » et orientée flux d'octets. Il nous faut donc des fonctions pour écrire (envoyer) et lire (recevoir) des octets dans la socket: 7. et 8.
Envoi de données en mode connecté : **int send(int sock, char * buffer, int nbOctets, int flag);**
Réception de données en mode connecté: **int receive(int numSock, char * buffer, int tailleMax, int flag);**
Envoi de données en mode non connecté :
int sendto(int sock, char * buffer, int nbOctets, int flag, struct sockaddr *sockAddr, int len);
Cette primitive peut envoyer au maximum un datagramme de **nbOctets** de données sur le socket **sock**. Les données du

datagramme sont stockées dans **buffer**. L'adresse de niveau transport du destinataire est indiquée dans **sockAddr**.

Réception de données en mode non connecté:

int recvfrom(int sock, char * buffer, int tailleMax, int flag, struct sockaddr *sockAddr, int *len);

Cette primitive est utilisée pour recevoir un datagramme d'au maximum **tailleMax** de données sur le socket **sock**. Les données du datagramme seront stockées dans **buffer** où le deuxième paramètre est un pointeur sur la chaîne d'octet à recevoir. L'adresse de niveau transport de l'émetteur sera indiquée dans **sockAddr**. **flag** est un ensemble de drapeaux qui peuvent être :

- envoi ou réception de données express aussi appelé hors-bande,
- prendre une copie des données reçues sans que celles-ci soient retirées du tampon à l'arrivée,

11. Fin de connexion: **int close (int sock);** équivalent à une fermeture de fichier. Donc, quel que soit le mode utilisé (sur ou hors connexion), cette primitive libère les ressources attribuées: la connexion, les tampons, le descripteur de socket.

12. Primitives de gestion des adresses et des structures

UNIX possède un grand nombre de primitives de services qui permettent d'écrire ces informations dans la structure d'adresse et de simplifier l'usage du réseau et en particulier les phases initiales.

gethostbyname(name) : Cette fonction permet de connaître l'adresse de la machine. La déclaration de la structure est faite dans le fichier d'include **netdb.h**.

gethostbyaddr(addr, len, type) : Cette fonction rend le nom symbolique correspondant à une adresse. Elle remplit une structure de type ***hostent*** à partir d'une adresse fournie en paramètre.

gethostname(name, namelen) : Elle permet de connaître le nom de la machine locale sur laquelle le programme s'exécute.

inet_aton() pour convertir une adresse IP depuis la notation IPv4 décimale pointée vers une forme binaire (dans l'ordre d'octet du réseau): Bibliothèque **#include <arpa/inet.h>**

htons() pour convertir le numéro de port (sur 16 bits) depuis l'ordre des octets de l'hôte vers celui du réseau: Bibliothèque **#include <arpa/inet.h>**.

htonl() pour convertir une adresse IP (sur 32 bits) depuis l'ordre des octets de l'hôte vers celui du réseau.