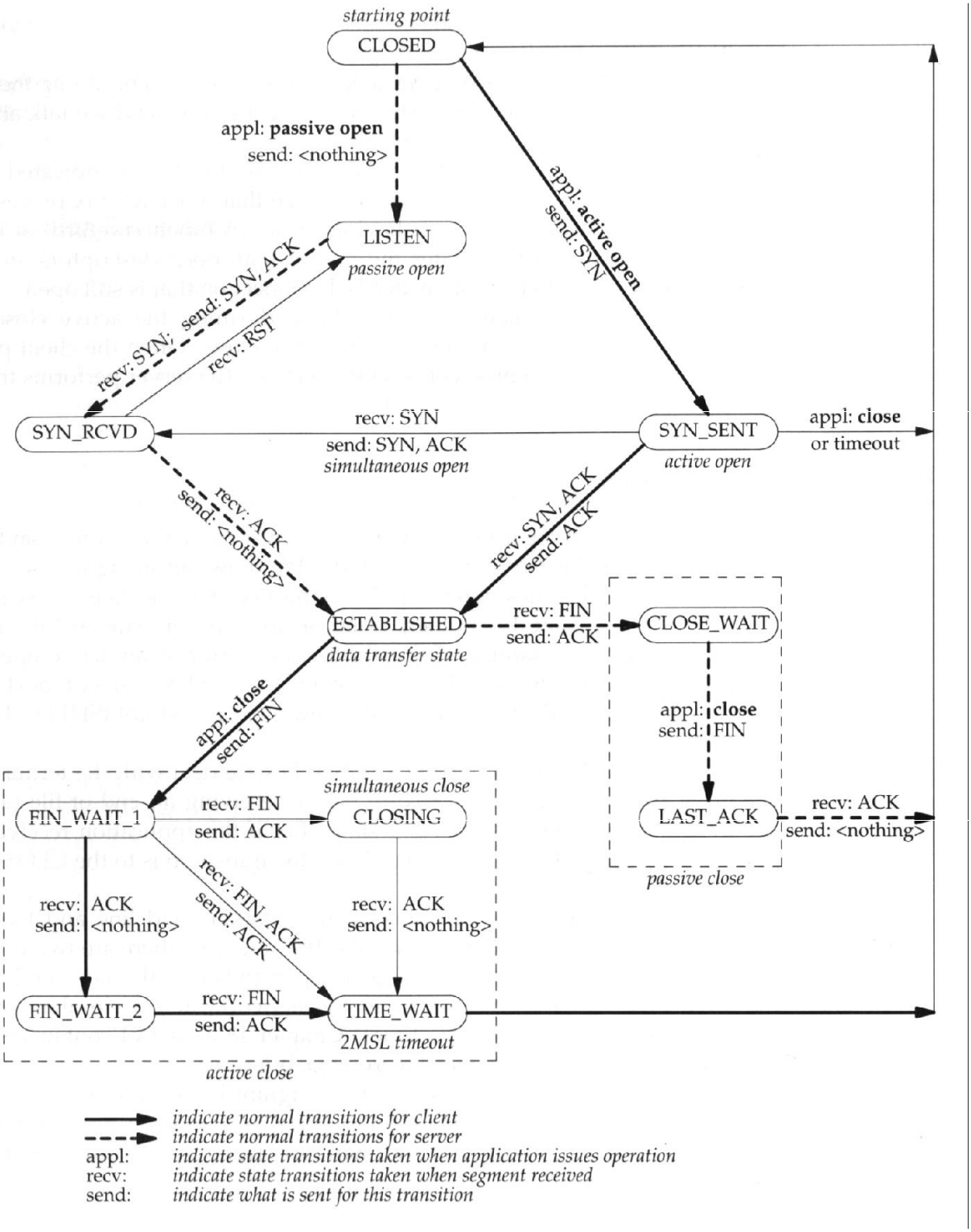
1) TCP (couche transport) est un protocole à états, c'est-à-dire qu'une connexion utilisant ce protocole se trouve toujours dans un état donné ou passe d'un état à un autre.



On peut distinguer :

- L'établissement d'une connexion :
    1) Le serveur se met à l'écoute (listen() -> état ECOUTE) : on parle d'ouverture passive de la connexion.
    2) Le client se connecte (connect()) : on parle d'ouverture active de la connexion. Il envoie un premier segment TCP de type SYN, ce qui permet de communiquer au serveur le premier numéro de séquence qui sera utilisée pour les données émises par le client, la connexion est alors dans l'état SYN_EMIS.
    3) Le serveur envoie au client un ACK et son propre SYN (en un paquet) qui initialise les numéros de séquence des données qu'il enverra, la connexion est alors dans l'état SYN_RECU.
    4) Le client envoie un ACK au serveur, la connexion est alors ETABLIE, cet état correspond au transfert de données entre le serveur et le client.

    Trois segments sont échangés pour que la connexion soir établie. (three-way handshake)

- La terminaison d'une connexion :
    1) Le client ferme la connexion (close()) : on parle de fermeture active de la connexion. Il envoie le premier segment TCP de type FIN, la connexion est alors dans l'état FIN_ATTENTE_1.
    2) Le serveur reçoit le segment de fin et réalise une fermeture passive, la connexion est alors dans l'état ATTENTE_FERMETURE, on dit encore que la connexion est à-demi fermée parce que des données peuvent encore transiter entre le client et le serveur.
    3) Le serveur envoie alors un ACK que le client doit recevoir, la connexion est alors dans l'état FIN_ATTENE_2.
    4) Après un certain temps de latence, la connexion est définitivement fermée par le serveur. Il envoie un segment TCP de type FIN, la connexion passe à l'état DERNIER_ACK. Le client répond par un ACK, la connexion est alors dans l'état TEMPORISATION pour attente que tous les paquets aient disparu. Le temps de latence équivaut à 2 fois les MSL (durée de vie d'un segment). Si vous voyez d'autres temps de latence merci de prévenir ;)
    5) La connexion passe dans l'état FERMEE.

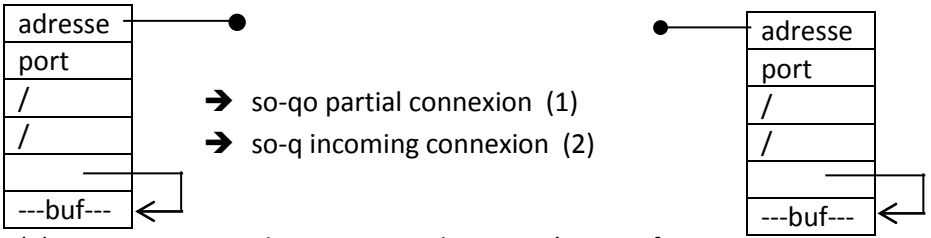
    Quatre segments ont dû être échangés pour que la connexion soit finalement fermée.

La commande netstat permet de visualiser les différentes étapes d'une connexion, ESTABLISHED, CLOSED, etc…

Etat	Signification
CLOSED (FERMEE)	Connexion fermée, inactive
LISTEN (ECOUTE)	Le serveur attend une demande de connexion
SYN_RCVD (SYN_RECU)	Une demande de connexion est arrivée
SYN_SENT (SYN_EMIS)	L'application a commencé à ouvrir une connexion
ESTABLISHED (ETABLIE)	La connexion est dans son état normal pour transférer des données
FIN_WAIT_1 (FIN_ATTENTE_1)	L'application indique qu'elle a terminé
FIN_WAIT_2 (FIN_ATTENTE_2)	Le serveur indique qu'il accepte cette terminaison
TIME_WAIT (TEMPORISATION)	Attente que tous les paquets aient disparu
CLOSING (FERMETURE_EN_COURS)	Le client et le serveur ont essayé de fermer simultanément
CLOSE_WAIT (ATTENTE_FERMETURE)	Le serveur reçoit une indication de fermeture
LAST_ACK (DERNIER_ACK)	Attente que tous les paquets aient disparu

socket () → crée la structure de base

Serveur Client



(1) : connexion pendante tant que le 3way n'est pas fais

(2) : connexion pendante établie (ESTABLISHED)

Serveur :

bind() → rempli le champs adresse et port

listen() →ajoute 2 queues (liste chainée)

accept() →Il prend la 1ere connexion dans les incoming connexion, il crée un 2eme socket. C'est une socket dupliqué (socket d'écoute) de la socket de base (socket de service). Il initialise le pipe dans lequel on peut lire/écrire (send(), recv()).

Client :

connect() → on nous attribue un port temporaire + adresse de la machine

2) Il y a 2 moyens d'implémenter un thread en java, soit par l'utilisation de l'interface Runnable, soit par héritage à partir de la classe Thread (qui elle implémente l'interface Runnable). Dans tous les cas c'est la méthode run() qui contient le code exécuté par le thread en question. Une thread est démarré par la méthode start() (appartient à la classe Thread)qui elle s'occupe d'appeler la méthode run().

- Héritage de Thread :

```
public class test extends Thread {
    // Variable
    ...
    // Propriété
    ...
    // Constructeur
    public test(...) {
            ...
    }
    // Méthode
    @Override
    public void run() {
            // Code d'exécution du thread
    }
    ...
}
public static void main(String[] args) {
    ...
    test t = new test(...);
    t.start(); // Lance le thread
    ...
}
```

- Interface Runnable:

```
public class test implements Runnable
{
    // Variable
    ...
    // Propriété
    ...
    // Constructeur
    public test(...) {
            ...
    }
    // Méthode
    public void run() {
            // Code d'exécution du thread
    }
```

```
        ...
    }
    public static void main(String[] args) {
        ...
        test t = new test(...);
        Thread thread1 = new Thread(t);
        thread1.start(); // Lance le thread
        ...
    }
```

L'arrêt d'un thread peut se faire de plusieurs façons. Soit lorsque celui-ci est terminé, il s'arrête. Soit, au moyen de la méthode stop(), cette méthode est deprecated puisque elle libère toute les ressources utilisé pas le thread (tous les moniteurs). Ces objets peuvent être laissés dans des états incohérents puisque qu'on pourrait interrompre une section critique, etc…

Le meilleur moyen est donc que le thread contient une variable membres indiquant son état (actif ou non). Cette variable est bien entendu accessible depuis l'extérieur de celui-ci. Dans l'exécution du code du thread on tachera alors de tester cette variable au moment voulu et en fonction du test arrêter le thread ou non. Cependant le thread peut être bloqué (sur un wait() par exemple). On utilise donc au préalable la méthode interrupt() qui met fin aux attentes du à un wait(), un join() ou un sleep() en lançant un exception instance de InterruptedException.

Il est possible de contourner l'utilisation d'interrupt() dans le cas d'un blocage sur un wait(). Il suffit de spécifier en paramètre un time-out pour le wait et de tester à intervalle régulier dans une boucle si ce n'est pas la fin du thread ou si le wait() a été exécuter.

Pour remédier au problème d'accès concurrent Java permet de déclarer une méthode comme étant synchronized. Lorsqu'un thread invoque une méthode de ce type sur un objet celui-ci est verrouillé, si un autre invoque une méthode synchronized (la même ou une autre) sur le même objet il y aura un blocage jusqu'à ce que l'exécution de la première méthode invoqué soit terminé. L'exclusion mutuelle est donc assurée. C'est ce qu'on appelle un monitor. Ces monitors sont réentrants c'est-à-dire qu'une méthode synchronized peut appeler une autre méthode synchronized appartenant au même objet sans qu'elle soit bloqué par le verrou du posé par le premier synchronized. Une méthode statique synchronized n'est utilisable que par un thread à la fois.

Exemples :

```
public synchronized void test() { … }
```

Il est également possible de soumettre seulement quelque instructions a la sections critique comme suit :

```
synchronized(<objet à verrouiller>)
{
        … // Code sous section critique
}
```

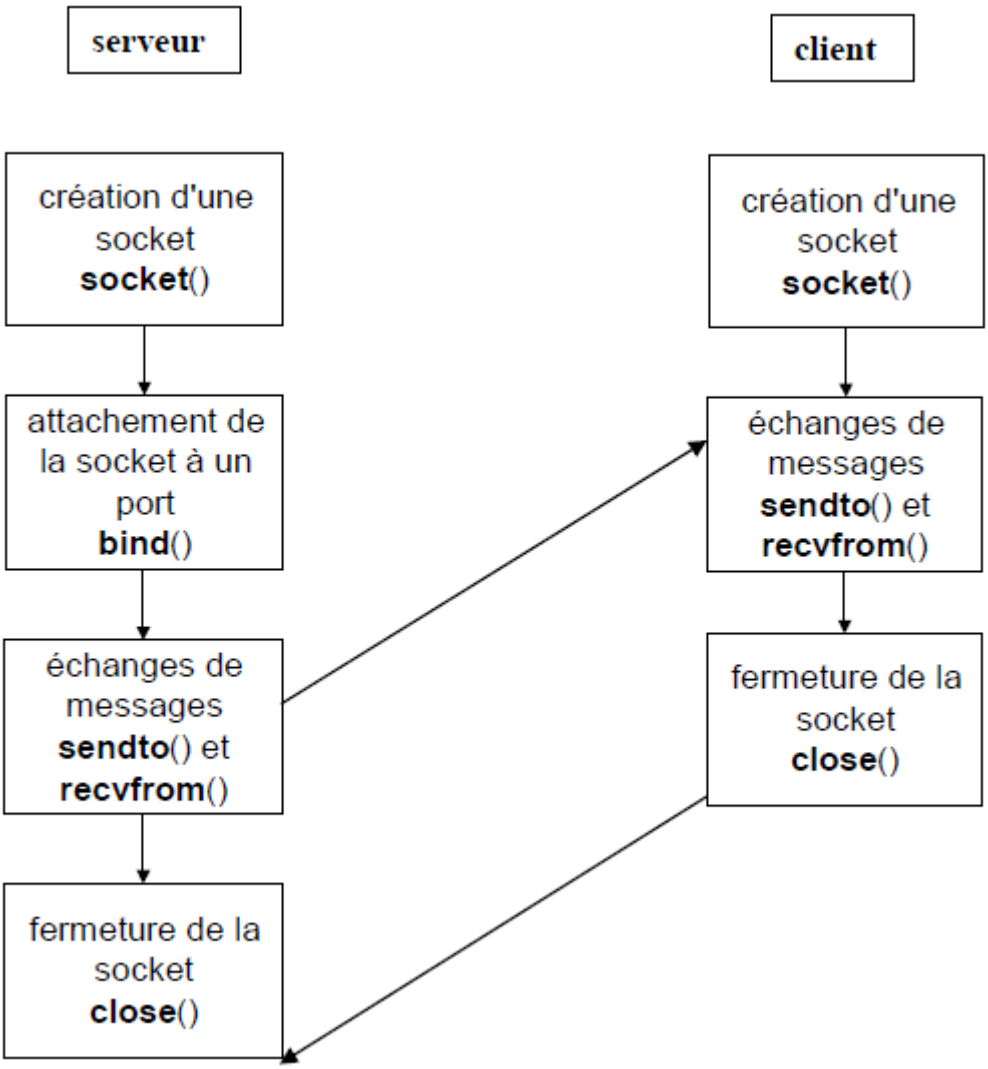
Il est aussi important de pouvoir gérer la synchronisation entre les threads (hors de la section critique) et cela peut se faire au moyen de 2 méthodes dérivées d'Object quoi doivent être appelée au sein d'une méthode synchronized :

- public final void wait() : cette méthode place le thread en attente d'une notification par un autre thread.
- public final native void notify(): cette méthode choisit un thread en attente sur le monitor et le réveille, la notification est perdu si il n'y a aucun thread actif.

Les timers Java utilise bel et bien la notion de Thread. Il existe 2 sortes de timers prédéfini en Java :

- La class Timer de java.util :
  Cette classe exécute une série de TimeTask. Ces TimeTask sont des objets implémentant Runnable. La classe Timer va donc threader ce différents TimeTask pour l'exécution des différentes actions.
- La class Timer de java.swing :
  Tous les timers de cette classe partage le même thread, ce dernier se « shedule » entre les différentes tâches. Cette classe délivre une notification a un ActionListenner spécifié dans son constructeur tous les X temps, ce délai est également préciser dans le constructeur. La tâche à exécuter doit donc être codé dans la méthode actionPerformed() de l'ActionListener spécifié.

3) UDP est un protocole non fiable et sans connexion, c'est-à-dire sans garantie de réception. Il a une complexité moindre et un trafic réseaux réduit. Il ne garantit pas non plus l'ordre de réception des données il faut donc envoyer de paquet de taille inférieur au MTU afin d'éviter la fragmentation des paquets et numéroté ces paquets si les données à envoyer sont plus grande. Le fait d'être orienté sans connexion permet le multicast. UDP Simplifie également la programmation puisque les applications l'utilisant doivent simplement créer une socket, la faire reconnaitre par le système, échanger la données, et fermé leur socket.



Programmation UDP en C :

```
int main()        // Serveur
{
        int hSocket, tailleSockaddr_in, nbreRecv;
        struct hostent * infosHost;
        struct in_addr adresseIPClient;
        struct sockaddr_in adresseSocketServeur, adresseSocketClient;
        char msgClient[TAILLEMAX];
        /* 1. Creation de la socket */
        hSocket = socket(AF_INET, SOCK_DGRAM, 0);
        if (hSocket == -1)
                exit(1);
        /* 2. Acquisition des informations sur l'ordinateur local */
        if ( (infosHost = gethostbyname("boole"))==0)
                exit(1);
        memcpy(&adresseIPServeur, infosHost->h_addr, infosHost->h_length);
        printf("Adresse IP = %s\n",inet_ntoa(adresseIPServeur));
        /* 3. Preparation de la structure sockaddr_in */
        tailleSockaddr_in = sizeof(struct sockaddr_in);
        memset(&adresseSocketServeur, 0, tailleSockaddr_in);
        adresseSocketServeur.sin_family = AF_INET;
        adresseSocketServeur.sin_port = htons(PORT);
        memcpy(&adresseSocketServeur.sin_addr, infosHost->h_addr,infosHost->h_length);
        /* 4. Le systeme prend connaissance de l'adresse et du port de la socket (coté serveur) */
        if (bind(hSocket, (struct sockaddr *)&adresseSocketServeur, sizeof(struct sockaddr_in)) == -1)
                exit(1);
        /* 5.Reception d'un message client */
        if ((nbreRecv = recvfrom(hSocket, msgClient, MAXSTRING, 0, &adresseSocketClient,
        &tailleSockaddr_in)) == -1)
        {
                close(hSocket); /* Fermeture de la socket */
                exit(1);
        }
        msgClient[nbreRecv+1]=0;
        printf("Message recu par le serveur = %s\n", msgClient);
        printf("Adresse de l'emetteur = %u\n", adresseSocketClient.sin_addr.s_addr);
        adresseIPClient = adresseSocketClient.sin_addr;
        printf("Adresse de l'emetteur = %s\n", inet_ntoa(adresseIPClient));

        /* 6. Fermeture des sockets */
        close(hSocket);
        return 0;
}
```

```
int main()        // Client
{
        int hSocket, tailleSockaddr_in;
        struct hostent * infosOther;
        struct sockaddr_in adresseSocketServeur;
        char msgClient[TAILLEMAX];
        /* 1. Creation de la socket */
        hSocket = socket(AF_INET, SOCK_DGRAM, 0);
        if (hSocket == -1)
                exit(1);
        /* 2. Acquisition des informations sur l'ordinateur local */
        if ( (infosOther = gethostbyname("boole"))==0)
                exit(1);
        memcpy(&adresseIPServeur, infosOther->h_addr, infosOther->h_length);
        printf("Adresse IP other = %s\n",inet_ntoa(adresseIPServeur));
        /* 3. Preparation de la structure sockaddr_in */
        tailleSockaddr_in = sizeof(struct sockaddr_in);
        memset(&adresseSocketServeur, 0, tailleSockaddr_in);
        adresseSocketServeur.sin_family = AF_INET;
        adresseSocketServeur.sin_port = htons(PORT_SERVEUR);
        memcpy(&adresseSocketServeur.sin_addr, infosOther->h_addr,infosOther->h_length);
        // Le client peut faire un bind() afin de fixer son port
        /* 4. Envoi d'un message au serveur */
        printf("Message a envoyer au serveur : ");
        gets(msgClient);
        if (sendto(hSocket, msgClient,MAXSTRING,0,&adresseSocketServeur,tailleSockaddr_in)==-1)
        {
                close(hSocket);
                exit(1);
        }
        /* 5. Fermeture des sockets */
        close(hSocket);
        return 0;
}
```

Remarque : Il faut faire attention que le client est toujours bien le même, en effet un 2eme client peut se connecter sur le même port, le serveur capte alors les messages de plusieurs clients. Pour éviter de répondre à des messages d'un autre client il doit vérifier l'adresse du paquet à chaque fois. Il faut également faire attention au fait que UDP est un protocole non fiable et un paquet peut-être perdu sans qu'on le remarque. Il est donc conseillé en c de combiné un recvfrom() avec un alarm() et SIGALRM.

Programmation UDP en Java :

Le package réseaux de Java propose une classe DatagramPacket qui matérialise un datagramme reçu ou envoyé. Elle possède des méthodes permettant de connaitre l'adresse ip et le port du client ainsi que récupérer les donnée et leur taille. Et une autre classe DatagramSocket qui représente un point d'envoi ou de réception d'un datagramme, elle dispose d'une méthode void send(DatagramPacket p) et d'une méthode void receive(DatagramPacket p).

```
DatagramSocket socket;
try
{
	socket = new DatagramSocket(port); // Si on ne défini pas de port on nous en attribue un

	// Réception

	byte[] buf = new byte[256];
	DatagramPacket paquet = new DatagramPacket(buf, 256);
	socket.receive(paquet);

	InetAddress adresse = paquet.getAdresse();
	int port = paquet.getPort();

	// Réponse

	String chaine = "reponse";

	buf = chaine.getBytes();

	paquet = new DatagramPacket(buf, buf.lenght, adresse, port);
	socket.send(paquet);

}
catch(SocketException | IOException e) {}
```

Paramétrage des sockets UDP est utilisé pour poser un timeout sur un recvfrom ou pour programmer un multicast (voir question 4).

Paramétrage pour Time out :

```
/* Paramétrage de la socket sur un time-out de 5 secondes */
struct timeval temps.tv_sec = 5; temps.tv_usec = 0;
setsockopt(hSocket, SOL_SOCKET, SO_RCVTIMEO, &temps, sizeof(temps));
do
{
	/* Envoi d'un message au serveur */
	printf("Message a envoyer au serveur : ");
	gets(msgClient);
	if (sendto(hSocket, msgClient, MAXSTRING, 0,
	&adresseSocketServeur, tailleSockaddr_in) == -1)
	{
		close(hSocket);
		exit(1);
	}
	/* 8.Reception d'un message serveur */
	memset(msgServeur, 0, MAXSTRING);
	if ((nbreRecv = recvfrom(hSocket, msgServeur, MAXSTRING, 0, &adresseSocketServeur,
	&tailleSockaddr_in) ) == -1)
	{
		if (errno == EWOULDBLOCK)
		{
			cpt++;
			puts("Time-out sur le recvfrom !");
			if (cpt<3) continue;
			else
			{
				puts("Le serveur semble etre mort ...");  break;
			}
		}
		else
		{
			printf("Erreur sur le recvfrom de la socket %d\n", errno);
			close(hSocket); exit(1);
		}
	}
	else
	{
		if (cpt>0) cpt=0;
			printf("Recvfrom socket OK\n");
	}
} while (strcmp(msgClient, "SHUTDOWN!"));
```

4) Il possible d'envoyer un message à plusieurs destinataire en une seule opération en utilisant une adresse multicast de classe D et le protocole de transport UDP. Une adresse de classe D ont un espace d'adresse compris entre 224.0.0.0 et 239.255.255.255. Une telle adresse n'identifie pas une seule interface, mais un ensemble d'interfaces qui, au travers de l'application qu'ils exécutent, ont manifesté leur souhait de recevoir les données adressées à cette adresse : on dit encore que ces interfaces participent à la session multicast ou qu'ils ont rejoint le groupe multicast. Lorsqu'on envoie un message à une telle adresse, ce sont donc tous les membres du groupe qui le reçoivent. Dans une application de type chat, on utilisera les adresse Multicast et le protocole UPD. Il sera nécessaire de dédié un thread à l'écoute permanente de nouveaux messages sur le réseau et un autre thread sera dédié à l'envoie de message.

Le multicast en C :

Pour rejoindre un groupe multicast, la socket doit être paramétrée grâce à la fonction setsockopt() qui utilise notamment les constantes suivantes :

Constantes de setsockopt()	Action
IP_ADD_MEMBERSHIP	Pour rejoindre un groupe multicast
IP_DROP_MEMBERSHIP	Pour quitter un groupe multicast
IP_MULTICAST_TTL	Pour spécifier le TTL d'un message multicast sortant
IP_MULTICAST_IF	Pour spécifier l'adresse à utiliser pour les messages sortants
IP_MULTICAST_LOOP	Pour spécifier si un message multicast sortant doit être aussi envoyé à la machine émettrice elle-même (loopback)

A la différence de l'envoi de messages multicast, la réception de tels messages implique :

- un bind de la socket locale au port utilisé pour le groupe multicast.
- l'adhésion au groupe multicast proprement dit.

En résumé, lorsqu'un datagramme multicast arrive sur une machine la couche IP reconnaît une adresse multicast et vérifie si certaines applications ont :

- joint le groupe multicast.
- si oui, la couche UDP cherche une socket liée au port multicast spécifié.
- si il la trouve, le message y est délivré.

```
Setsockopt(hsocket, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mreq, sizeof(mreq)) ;

Struct ip_mreq
{
        Struct in_addr imr_multiaddr; //IP multicast address of group
        Struct in_addr imr_interface; //locl ip address of interface
}
```

EXEMPLE p 288 prog TCP IP

Par défaut, les datagrammes IP multicast sont envoyés avec un TTL de 1

Pour modifier TTL
unsigned char ttl = 2;
Setsockopt(hSocket, IPPPROTO_IP, IP_MULTICAST_TTL, &ttl, sizeof(ttl)) ;

Si on veut qu'il puisse aussi communiquer: un datagramme IP ne peut être utilisé par une socket liée à une adresse multicast. Nous serons donc obligés d'utiliser deux sockets, une d'envoi et une de réception.

Le Multicast en Java :

Le package java.net fournit une classe MulticastSocket, dérivée de la classe DataGramSocket dont on se sert en Java pour les communications basées sur UDP.

Le constructeur réalise le bind sur le port spécifié :
public MulticastSocket(int port) throws IOException

Pour rejoindre et quitter un groupe multicast on utilise les méthodes :
public void joinGroup(InetAddress mcastaddr) throws IOException
public void leaveGroup(InetAddress mcastaddr) throws IOException
Dont le paramètre permet de préciser l'adresse du groupe multicast considéré.

Il est dés lors possible d'envoyer et recevoir des datagrammes au moyen des méthodes send() et receive() qui utilise des objets instances de la classe DatagramPacket.
Pour que les messages Java soit compréhensibles par le client C et inversement à chaque, on doit faire un getBytes() du message pour n'envoyer que les bytes utiles au message.

Exemple pour rejoindre un groupe mulitcast :
```
MulticastSocket socketGroupe;
InetAddress adresseGroupe;
try
{
        adresseGroupe = InetAddress.getByName("10.59.5.9");
        socketGroupe = new MulticastSocket(5000);
        socketGroupe.joinGroup(adresseGroupe);
}
catch (UnknownHostException e){ }
catch (IOException e){ }
```

Envoi d'un message :
```
DatagramPacket envoi = new DatagramPacket(msg.getBytes(),msg.length(), addressGroupe, 5000) ;
socketGroupe.send(envoi) ;
```

Exemple du code du thread de réception d'un message :
```
try
{
```

```
            Byte[] buffer = new byte[1000] ;
            DatagramPacket recu = new DatagramPacket(buffer, buffer.length) ;
            socketGroupe.receive(recu) ;
            System.out.println(new String(buffer.trim()) ;
}
catch (IOException e){ }
```

5) JDBC nous permet, au moyen d'un driver, de communiquer avec la base de données au moyen du langage SQL, il permet également la réception, pour traitement, des résultats des requêtes. C'est le rôle du driver JDBC de communiquer avec le SGBD visé, il sert de pont entre l'applic Java et ce SGBD. On se sert d'un DriverManager pour localiser les différents drivers possibles. On en distingue 4 types :
    - Type 1 : Le driver Java mappe simplement un driver natif.
    - Type 2 : Le driver est partiellement écrit en Java et partiellement avec du code natif, il utilise en fait une librairie native.
    - Type 3 : Le driver est cette fois entièrement en Java et il utilise un middleware qui réalise l'accès au SGBD, la communication driver-middleware est donc indépendante du protocole d'accès à la base à travers le SGBD.
    - Type 4 : Le driver, entièrement écrit en Java, implémente le protocole d'accès à la BD, il est donc propriétaire dans le sens qu'il est dédié à un type de SGBD donné.

    Ces drivers sont uniques pour chaque SGBD et sont défini selon la syntaxe habituelle des classe et des packages. Cette classe est chargé par le DriverManager de façon dynamique (introspection) au moment de la création de la connexion qui se fait au moyen de la méthode public static Connection get Connection(String url, String user, String password)
    L'URL se compose comme tel : jdbc : <nom du sous protocole> :<alias du non de la BD pour ce protocole>.

    Cette méthode fourni un objet Connection (Connection est en fait une interface qui est implémenté par un classe dans le driver) qui représente la connexion demandée qui comprend toute les commandes SQL en cours et leur résultat. Elle dispose de plusieurs méthodes :
    - void close() qui permet de fermer la connexion avec le SGBD
    - Statement createStatement(…) qui permet de créer une instruction à exécuter sur la BD
    - void setAutoCommit(boolean autoCommit) qui sert à activé ou non la validations automatique des changement fait la BD. Si sa valeur est à false les validations sont à faire manuellement avec les méthodes : void commit() pour valider les modifications et void rollback() pour les annuler.

    Pour exécuter une commande SQL on utilise un objet Statement que l'on crée avec la méthode createStatement décrite ci-dessus. Cette objet contient 3 méthodes :
    - ResultSet executeQuerry(String sql) permet de faire une requête de sélection dont le résultat se matérialise par un ResultSet.
    - boolean execute(String sql) permet du faire une requête quelconque dont le résultat se matérialise par un ResultSet ou non.
    - int executeUpdate(String sql) qui permet d'effectuer une requête de mise à jour.

    Le ResultSet représente en quelque sorte le curseur en SGBD. Il contient une liste de résultats de la requête demandé, il peut être parcourus grâce à la méthode boolean next() la fin du cruseur est détecté par la valeur de retour false du next(). On peut récupérer les valeurs grâce aux méthodes getXXX(String columnName) (getInt(), getBoolean(), get String(), etc…). Comme on

peut le deviner, le driver JDBC se charge de convertir les données lues dans les tables au format utilisable en Java.

Il est possible de paramétrer le ResultSet lors de la création d'un Statement avec le 2eme constructeur Statement createStatement(int resultSetType, int resultSetConcurrency). Le premier paramètre est le type de déplacement autorisé et prend comme valeur un des constante de la classe ResultSet :

- TYPE_FORWARD_ONLY séquentiel uniquement
- TYPE_SCROLL_SENSITIVE quelconque, les changements apportés par d'autres opérations sont pris en compte
- TYPE_SCROLL_INSENSITIVE quelconque, les changements apportés par d'autres opérations ne sont pas pris en compte

Le 2 eme paramètre détermine si le curseur est read-only ou read-write, au moyen des constantes :

- CONCUR_READ_ONLY read-only
- CONCUR_UPDATABLE read-write

Il est possible de faire des mises à jour au moyen d'un curseur avec les méthodes void updateString(String columnName, String x) etc… pour que ces chagement soit propagé dans la BD on utilise la méthode void updateRow().

Ou une insertion avec la méthode void moveToInsertRow() cette méthode place le curseur sur un tuple d'insertion, sur lequel on fait des updates pour inséré des valeurs dans les champs et on utilise la méthode void insertRow() pour confirmer l'insertion.

Ou enfin une suppression avec la méthode deleteRow() qui supprime le tuple courant.

Il est également possible de faire des requêtes dynamiques avec un PreparedStatement. Avec ce type de Statement on peut écrire une requête sql avec de paramètre dynamique définit par « ? ». on peut définir la paramètre grâce au méthode void setXXX(int positionParametre, XXX valeur) (setBoolean, setByte).

On peut dire que JDBC est de hauts niveaux car il nous permet de communiquer avec tous les types de base de données de la même façon, en suivant le même schéma. (Que ce soit des fichiers csv ou bien un BD relationnel ou bien une BD orienté objet, la programmation des accès dans l'application se fait de la même façon). Mais si on le compare à JPA, JDBC est de bas niveau car toutes les requêtes se font au moyen de SQL.

JPA est un moyen d'accès aux BD de plus haut niveau, il permet d'interroger la base de données de manière plus intuitive. JPA ne nécessite pas de connaissance en SQL. De plus chaque entité est représentée par une classe qui s'occupe de géré la correspondance avec les tuples (représenté par les objets) de la table concernée.

6) Fonctionnement de base du serveur :
    - Lance les threads chargés de gérer les requêtes d'un client (en attente sur un wait())
    - Se met à l'écoute sur un port donné
    - A chaque connexion, il lit la requête sous forme d'un objet sérialisé
    - Charge un thread du pool du traitement de la requête

Afin de conserver la généricité ainsi que garantir la gestion des requêtes même si tous les threads du pool sont occupés il est bien de placé dans une file le traitement correspondant à la requête que le premier thread disponible exécutera. Ce traitement sera donc un objet Runnable qui contient la méthode run() exécuté par le thread qui le prendra en charge.

Pour réaliser un serveur générique il faut définir un certain comportement, certaine règle afin de garantir un fonctionnement correct et cohérent. Pour cela nous allons créer une série d'interface que devront implémenter les différents objets de notre serveur.

La file (qui pourrait être autre chose, imaginons par exemple un système de priorité, etc…) est pensé de manière abstraite comme implémentant l'interface SourceTaches . Un objet implémentant cette interface contient une série d'objet Runnable à exécuter auquel on pourra accéder grâce aux méthodes ci-dessous :

```
public interface SourceTaches
{
        // Fournit une tâche à exécuter
        public Runnable getTache() throws InterruptedException;
        // Détermine si il y a encore des tâches à exécuter ou non
        public boolean existTaches();
        // Permet d'ajouter une tâche a exécuter à la file (liste, etc...)
        public void recordTache(Runnable r);
}
```

Il faut pouvoir associer une requête reçue à un objet Runnable, on peut donc imaginer une interface Requete comme suit :

```
public interface Requete
{
        // Fournit le traitement associé à la requête
        // On passe en paramètre la socket ainsi qu'un objet permettant de
        // garder une trace des actions
        public Runnable createRunnable(Socket s, ConsoleServeur cs);
}
```

ConsoleServeur est en fait une interface implémenté objet permettant de garder une trace des actions effectuer (cela peut être une fenêtre GUI, ou bien un simple fichier trace) :

```
public interface ConsoleServeur {
        // Permet de conserver une trace d'une action quelconque
        public void traceEvenements(String trace);
}
```

Il faut également définir une interface Reponse :

```
public interface Reponse {
        // Fournit le code de la réponse
        public int getCode();
}
```

Le thread serveur est lancé au démarrage, il crée la socket et lance les threads de son pool puis boucle sur la connexion d'un client et la réception d'une requête. Dés réception de la requête il crée l'objet Runnable correspondant (grâce à la méthode createRunnable()) et ajoute cet objet a la liste des requêtes en attente (objet implémentant l'interface SourceTache).

Les threads du pool sont simples, ils bouclent et attendent qu'une tâche soit mise à disposition (méthode getTache()).

On peut imaginer un système de notification :

- attente sur un wait() des threads du pool
- chaque ajout de tâche dans la liste fait un notify() qui réveille un thread du pool
- ce dernier exécute la tâche qui lui est donné
- il vérifie s'il n'y a pas une autre tâche à exécuter (si tous les threads étaient occupés)
- si il y en a une autre il repart à l'étape d'exécution sinon il se remet en attente sur un wait()

C'est le rôle de l'objet implémentant l'interface SourceTaches de faire cela. En effet cet objet met en attente les threads qui appellent la méthode getTache() si il n'y a pas de tâches à effectuer, et relance un thread, notify(), quand une nouvelle tâche est ajouté.

RequeteX est une classe représentant l'objet transmis sur le réseau. Elle implémente les interfaces Requete et Serializable. Elle contient des constantes représentant les différentes requêtes possibles. Une de ces constantes en passer au constructeur pour déterminer le type de requête. Cet objet redéfini également la méthode createRunnable() qui fournit l'exécution à faire par le serveur.

ReponseX est la classe représentant la réponse de la requête. Elle implémente Reponse et Serializable. Elle contient les constantes pour identifier le type de réponse ainsi que une charge utile sous forme de string qui contient la réponse.