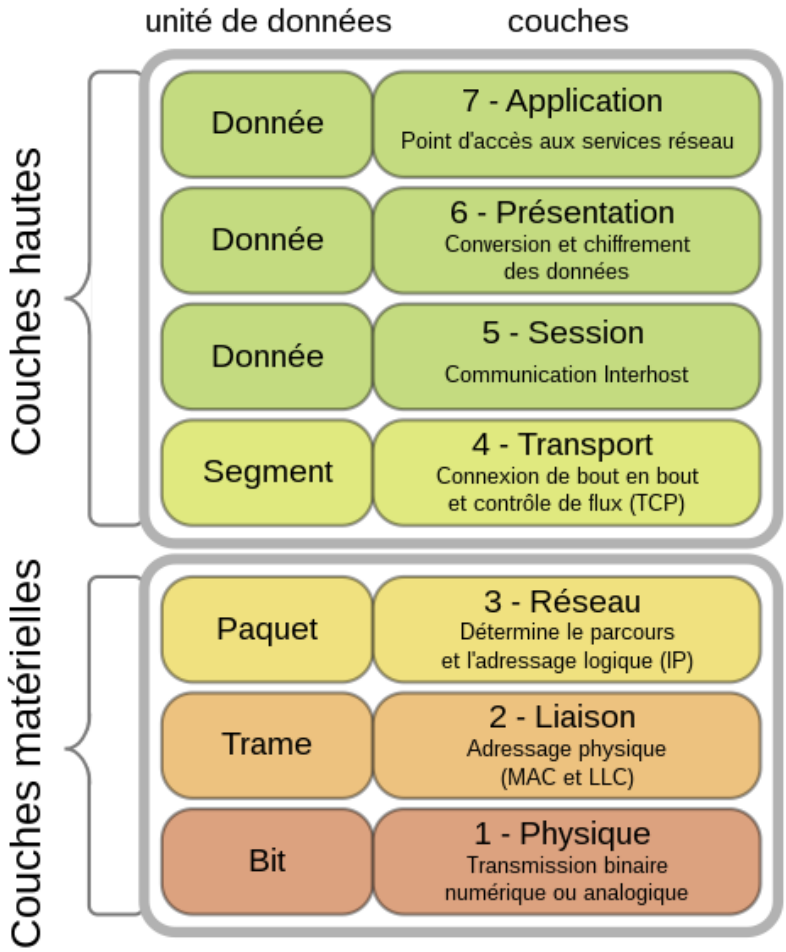
Les communications réseaux en C

Protocoles TCP-UDP/IP

Une histoire de protocoles : modèles OSI et TCP/IP

Le modèle **OSI** (**O**pen **S**ystem **I**nterconnection) est une norme de communication, en réseau, de tous les systèmes informatiques. C'est un modèle théorique basé sur 7 couches :



Source : wikipedia

Tout au-dessus, on retrouve les applications informatiques (C, C++, Java, Web, …) avec

- leurs **données** à échanger entre ordinateurs distants, ainsi que
- les **points d'accès** aux services réseau : les **sockets** (voir plus loin).

Plus on descend dans ces couches, plus les données sont « transformées » (on dit plutôt « encapsulées ») pour se rapprocher et s'adapter à la couche physique (supports de transmission comme les câbles réseaux, les communications sans fil, …)

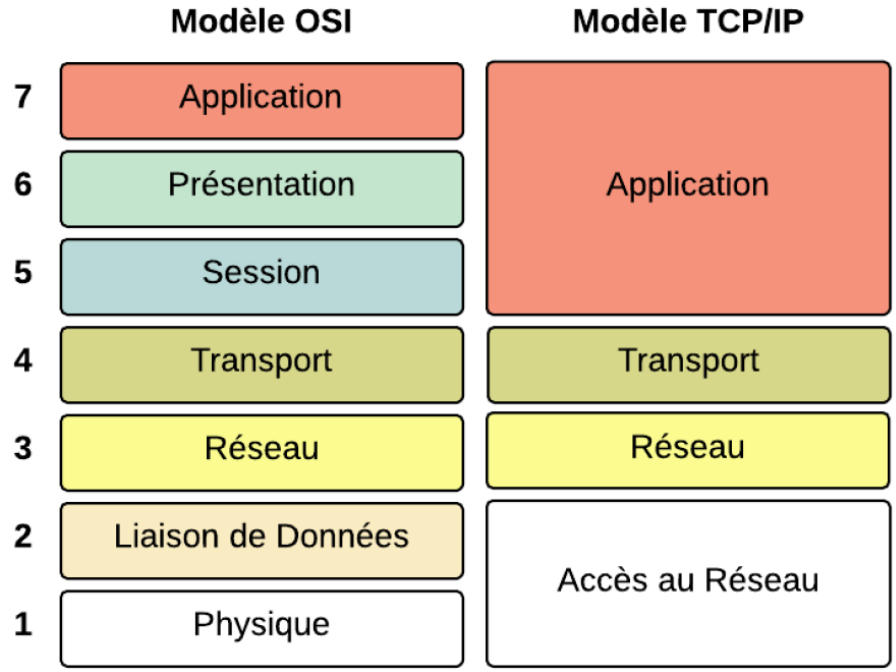
De plus,

- les 3 couches inférieures sont plutôt orientées **communication** et sont souvent fournies par le **système d'exploitation** et le **matériel**
- les 4 couches supérieures sont plutôt orientées **application** et plutôt réalisées par des **bibliothèques** ou un programme spécifique.

Ce modèle OSI est **générique**, indépendant du protocole, mais la plupart des protocoles et des systèmes y adhèrent.

Par contre le **modèle TCP/IP** est basé sur des protocoles standard que l'**Internet** a développés.

Le **modèle TCP/IP** (aussi appelé **modèle Internet**) comporte 4 couches et peut être mis en parallèle avec le modèle OSI :



On remarque que

- les 2 couches les plus « physiques » ont été regroupées
- les 3 couches les plus hautes du modèle OSI ont été regroupées pour former la couche application de modèle TCP/IP

Schématiquement, on retrouve dans les 4 couches du modèle TCP/IP :

Couches	Protocoles
Application **[données utilisateurs]**	**Telnet** : Terminal Network – login à distance **FTP** : File Transfert Protocol – transfert de fichiers **HTTP** : Hyper Text Transfert Protocol – gestion pages Web **SMTP** : Simple Mail Transfert Protocol **DNS** : Domain Name System – gestion du nom des machines **RPC** : Remote Procedure Call …
Transport **[segment/datagramme]**	**TCP** et **UDP**
Réseau ou **Internet** **[datagramme/paquet]**	**IP** : Internet Protocol – transport de datagrammes sans garantie **ICMP** : Internet Control Message Protocol – échange de messages d'erreur et d'informations [ping] **IGMP** : Internet Group Management Protocol – envoi de datagrammes UDP vers plusieurs hôtes
Accès au Réseau ou **Hôte Réseau** **[trame]**	**ARP** (Address Resolution Protocol) / **RARP** (Reverse Address Resolution Protocol) : conversion addresses IP / addresses de certaines interfaces reseaux (comme Ethernet) Drivers et cartes réseaux

TCP (**T**ransport **C**ontrol **P**rotocol) est

- **Orienté connexion** (une connexion doit être établie avant l'échange des données)
- **Fiable** (les paquets de données envoyés sont assurés d'arriver et dans le même ordre que celui dans lequel ils ont été émis)
- Assure un transfert de données par **« flot »** (un peu comme les pipes)

UDP (**U**ser **D**atagram **P**rotocol) est

- **Non orienté connexion** (pas d'établissement de connexion avant l'échange des données)
- **Non Fiable** (certaines données peuvent ne pas arriver à destination et l'ordre d'arrivée des paquets de données peut ne pas être respecté)
- Assure un transfert de données par **« paquets »** appelés datagrammes

Mais que signifie passer d'une couche supérieure à une couche inférieure ?

→ **encapsuler les données**, en partant des données utilisateurs jusqu'à la trame Ethernet envoyée physiquement sur le réseau.

Avec le modèle TCP/IP, les données de l'utilisateur sont encapsulées dans une **trame Ethernet** pouvant contenir de **46 à 1500 octets** :

header Ethernet 14 b	header IP 20 b	header TCP 20 b	header application	**data user**	trailer Ethernet 4

				data user	
			data application		
		segment TCP			
	datagramme IP				
trame Ethernet					

Des couches supérieures vers les couches inférieures :

- « **data application** » représente les données utilisateur et le protocole « utilisateur »
- On ajoute à ce « paquet de bytes » une **entête TCP** (comportant 20 bytes) pour former un **segment TCP**. Cette entête comporte notamment :
  - **Port source**
  - **Port destination**
  - **Numéro de séquence** : permet d'assurer l'ordre des segments TCP
  - **Numéro d'accusé de réception** : permet d'assurer le renvoi d'un segment perdu et donc la fiabilité du protocole
  - **Somme de contrôle** (checksum) : permet d'assurer la détection d'un segment corrompu et donc la fiabilité du protocole.
- On ajoute au segment TCP une **entête IP** (comportant 20 bytes) pour former un **datagramme IP**. Cette entête comporte notamment :
  - **Adresse IP source**
  - **Adresse IP destination**
  - **Time to live (TTL)** : un champ qui limite la durée de vie du datagramme → permet d'éviter qu'un datagramme IP erre indéfiniment sur le réseau
- On ajouter au datagramme IP un « header » et un « trailer » **Ethernet** pour former une **trame Ethernet**

Avec Ethernet,

- la taille maximum d'une trame est limitée et cette limite s'appelle le **MTU** (**M**aximum **T**ransfert **U**nit) qui est donc de **1500 octets**.
- IP devra fragmenter un datagramme de taille supérieure à ce MTU en morceaux de taille inférieure à ce MTU diminué de la taille des entêtes

Adresses IP, ports et sockets

- Une machine est identifiée logiquement sur le réseau à l'aide d'une **adresse IP** (codée sur 32 bits) → Exemple : 10.43.246.115
- Plusieurs processus peuvent utiliser la même adresse IP → pour diriger les paquets de données vers les bonnes applications, on utilise un **numéro de port** (codé sur 16 bits) différent par « application »

Un **numéro de port** est un nombre positif compris entre 0 et 65535 dont les valeurs de 0 à 1023 sont réservées par des protocoles bien connus :

- HTTP → port 80
- FTP → ports 20 et 21
- SSH → port 22
- Telnet → port 23
- SMTP → 25
- …

En **mode connecté (TCP)**,

Une connexion établie associe une adresse IP et un port → les deux ensembles constituent ce que l'on appelle une **socket**, ou un **point de connexion**

En **mode datagramme (UDP)**,

Les paquets de données sont accompagnés des adresses et numéros de port des destinataires

Transmission TCP : une machine à états finis

Principes

Pour rappel, **TCP** est un protocole :

- Orienté **flux** (comme un pipe de communication)
- Orienté **connexion** et **fiable** (comme une communication téléphonique)

Pour ce faire, TCP utilise des accusés de réception (**ACK** pour acknowledgement) :

1. Les données de l'application sont morcelées en segments
2. TCP envoie un segment avec un time-out et attend l'ACK de la cible. Si aucun ACK n'est reçu après le time-out, il y a retransmission
3. Si TCP reçoit des données, il envoie un ACK à la source après un délai assez court

De plus,

- TCP gère un checksum : si un segment arrive avec un checksum erroné, TCP le détruit et n'envoie pas d'ACK à la source
- Selon IP, les segments de données peuvent arriver dans un ordre différent de celui d'émission : TCP est capable de les remettre en ordre (via le numéro de séquence)
- Un datagramme IP qui serait dupliqué est supprimé

Il faut remarquer qu'un routeur peut découper en plusieurs morceaux un segment trop grand. Chaque nouveau segment sera véhiculé indépendamment (donc avec son propre entête IP et TCP).

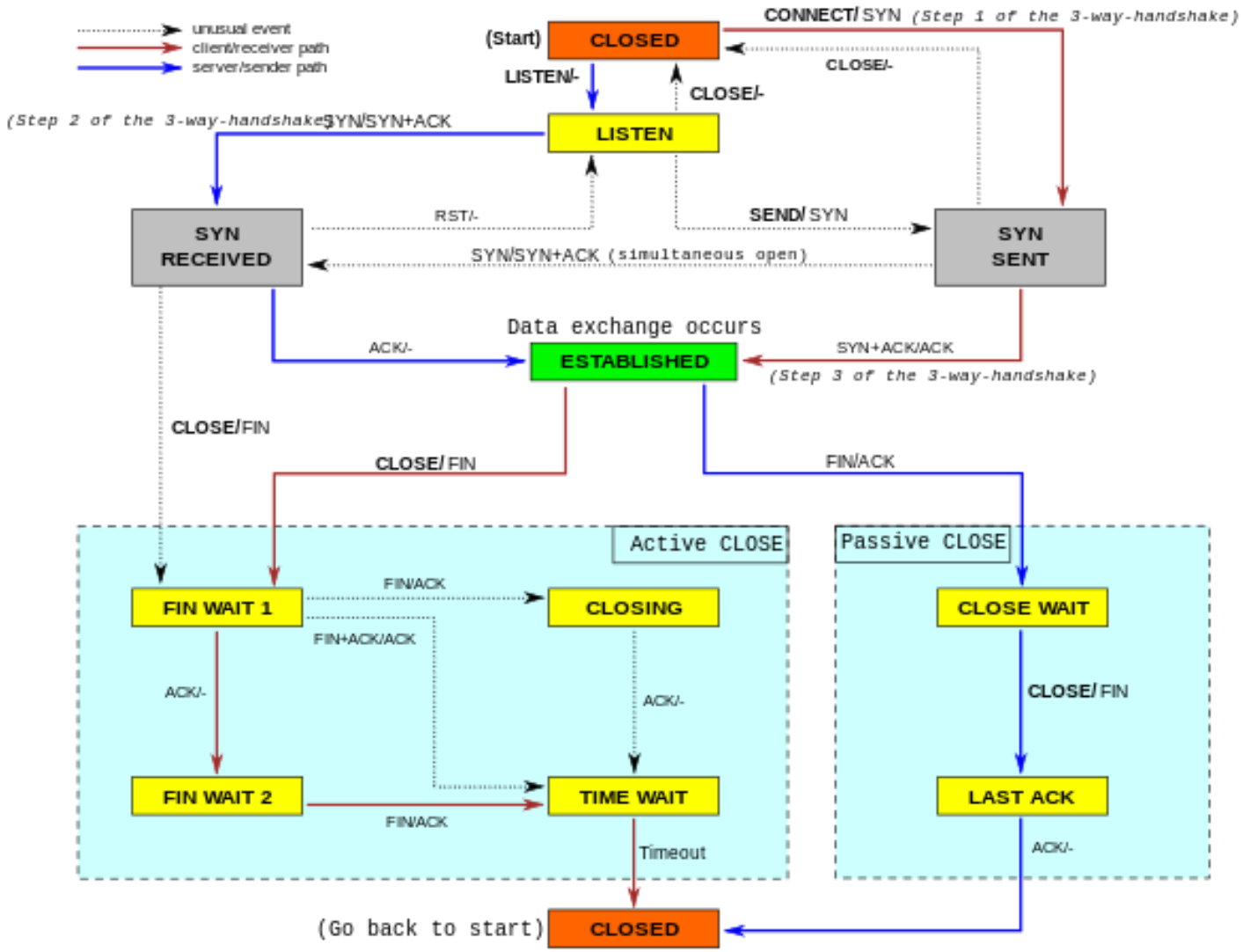
La machine à états de TCP

TCP est un **protocole à états** → une connexion utilisant ce protocole se trouve toujours dans un état donné ou passe d'un état à un autre → il y a donc une machine à états par connexion et non pas une machine à états par intervenant dans la communication.

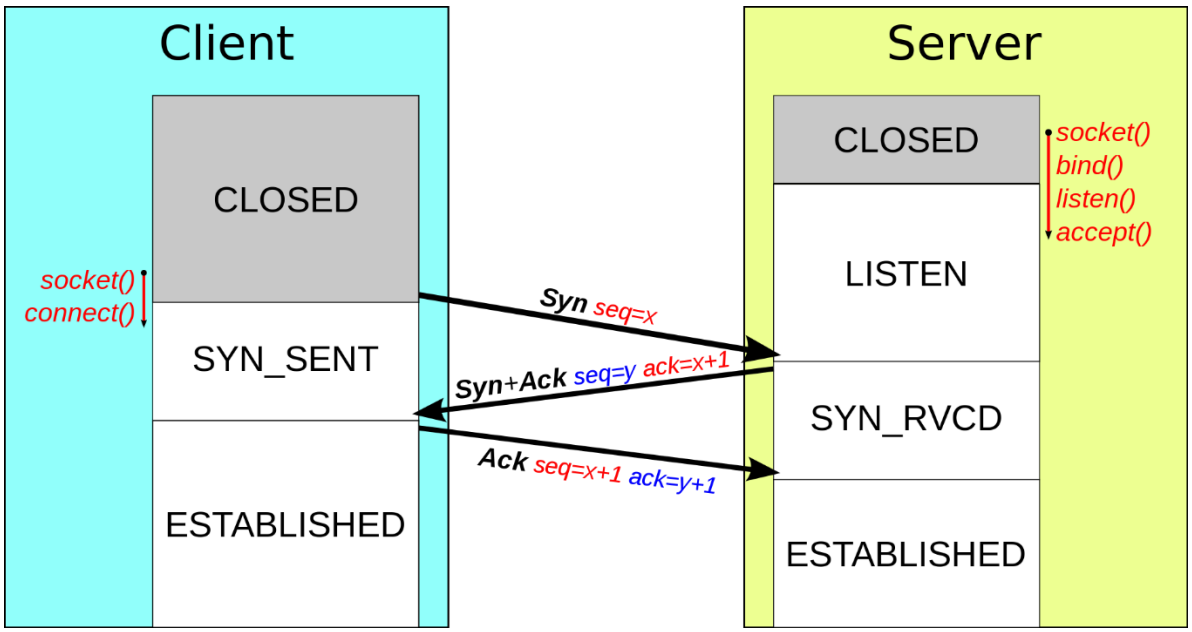
Voici un tableau reprenant les différents états possibles :

Etat	Signification
CLOSED	Connexion fermée, inactive
LISTEN	Le serveur attend une demande de connexion
SYN_RCVD	Une demande de connexion est arrivée
SYN_SENT	L'application a commencé à ouvrir une connexion
ESTABLISHED	La connexion est dans son état normal pour transférer des données
FIN_WAIT1	L'application indique qu'elle a terminé
FIN_WAIT2	Le serveur indique qu'il accepte cette terminaison
TIME_WAIT	Attente que tous les paquets aient disparu
CLOSING	Le client et le serveur ont essayé de fermer simultanément
CLOSE_WAIT	Le serveur reçoit une indication de fermeture
LAST_ACK	Attente que tous les paquets aient disparu

Et voici le diagramme d'états correspondant :



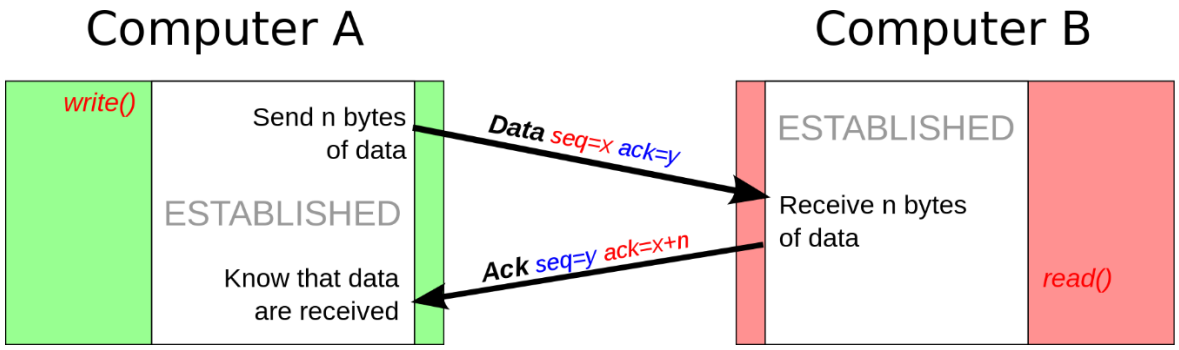
Etablissement d'une connexion :



1. Le serveur ouvre une socket (appel système **socket()**) et se met à l'écoute (appel système **listen()** → état **LISTEN**) : on parle encore d'ouverture passive de la connexion
2. Le client se connecte (appel système **connect()**) : on parle encore d'ouverture active de la connexion → il envoie alors un premier segment TCP de type SYN, ce qui permet de communiquer au serveur le 1er numéro de séquence qui sera utilisé pour les données émises par le client → la connexion est alors dans l'état **SYN_SENT**
3. Le serveur envoie au client un ACK et propre SYN qui initialise les numéros de séquence des données qu'il enverra → la connexion est alors dans l'état **SYN_RCVD**
4. Le client envoie un ACK au serveur : la connexion est alors établie et dans l'état **ESTABLISHED** → cet état correspond en fait au transfert de données entre le serveur et le client

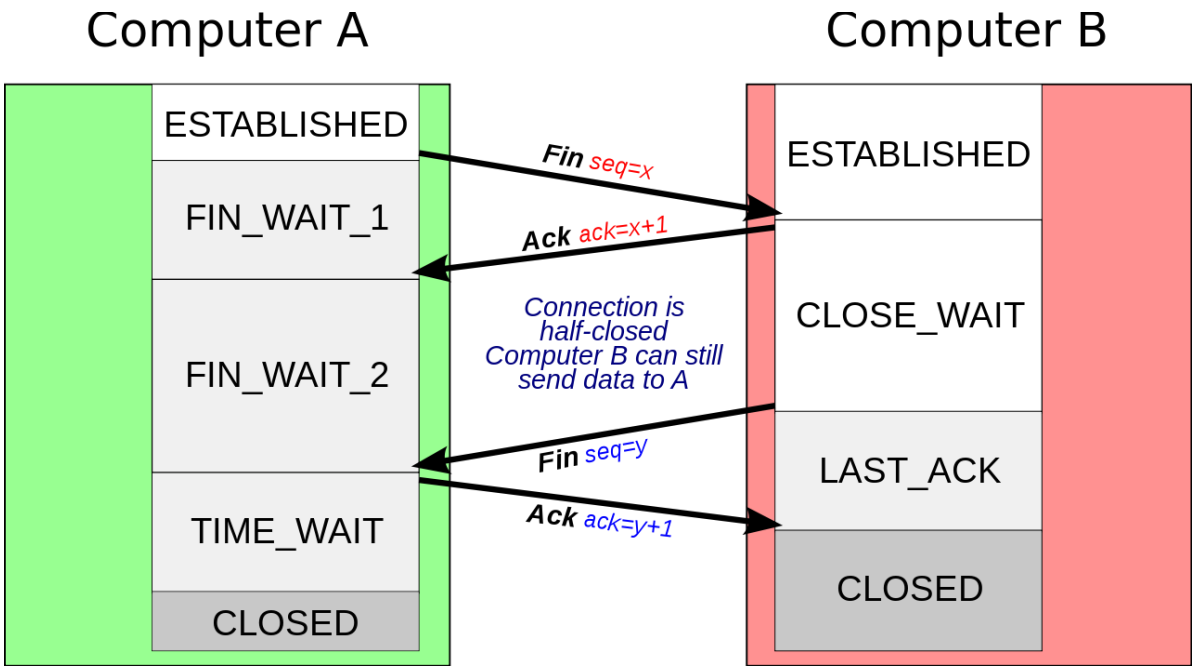
Trois segments ont dû être échangés pour que la connexion soit établie (**three-way handshake**).

Transfert de données :



- Les machines distantes utilisent les appels systèmes write() et read() pour se transférer des données
- Les numéros de séquence sont utilisés afin d'ordonner les segments TCP reçus et détecter les données perdues
- Les checksum permettent la détection des erreurs
- Les acquittements (ACK) permettent la détection des segments perdus ou retardés

Terminaison d'une connexion :



- Le client ferme la connexion (appel système **close()**) : on parle de fermeture active de la connexion → il envoie un premier segment TCP de type FIN → la connexion est alors dans l'état **FIN_WAIT1**
- Le serveur reçoit le segment de fin et réalise une fermeture passive : la connexion est alors dans l'état **CLOSE_WAIT** → on dit encore que la connexion est à demi

fermée parce que des données peuvent encore transiter entre le client et le serveur

- Le serveur envoie un ACK que le client doit recevoir : la connexion est alors dans l'état **FIN_WAIT2**
- 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 **LAST_ACK**
- Lorsque le client reçoit ce segment, il envoie un ACK : la connexion est alors dans l'état **TIME_WAIT** pour attendre que tous les paquets aient bien disparu
- La connexion passe à l'état **CLOSED**

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

Les points de communication : les sockets

Une **socket** est un **point de communication bidirectionnel** par lequel un processus peut émettre ou recevoir des données

Par « point de communication », il faut comprendre l'analogue

- d'un poste téléphonique (dans le cas de **TCP** – mode connecté)
- d'une boîte aux lettres (dans le cas de **UDP** – mode non connecté)

En mode connecté (**TCP** donc), une socket représente donc un **couple (adresse IP,port)**.

Création d'une socket

Pour créer une socket, on utilise l'**appel système**

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol) ;
```

où

- **domain** est le « domaine de communication » → il définit une famille d'adresses (plus exactement le format de adresses possibles). Il peut prendre les valeurs suivantes :
  - **AF_INET** : protocole fondé sur IPv4

    - **AF_INET6** : protocole IPv6 (soumis à des options de compilation particulières)
    - **AF_UNIX** : communication limitée aux processus résidant sur la même machine
    - …
- **type** est le type de socket → mode connecté (TCP) ou mode non connecté (UDP). Il peut prendre les valeurs suivantes :
    - **SOCK_STREAM** : mode connecté (TCP)
    - **SOCK_DGRAM** : mode non connecté (UDP)
    - **SOCK_RAW** : pour dialoguer de manière brute avec le protocole (le plus souvent IP)
- **protocol** est le protocole désiré. Il peut prendre les valeurs
    - **IPPROTO_TCP**
    - **IPPROTO_UDP**
    - **IPPROTO_IP**
    - …

En ce qui nous concerne, le domaine sera **AF_INET** et le protocole sera soit **IPPROTO_TCP**, soit **IPPROTO_UDP**. Dans ce cas, le choix du protocole est implicite et le paramètre **protocol** peut être mis à 0. Le système d'exploitation choisira le seul protocole adapté :

domaine	*type*	*seul protocole utilisable*
AF_INET	SOCK_STREAM	TCP
AF_INET	SOCK_DGRAM	UDP

Le retour de l'appel système est

- Le **descripteur de la socket** (un entier supérieur ou égale à 0) si tout s'est bien passé
- **-1** en cas d'erreur (et errno est positionné)

Remarquez que le descripteur d'une socket est équivalent au descripteur d'un fichier ouvert à l'aide de l'appel système **open()**. L'important ici est de noter qu'actuellement la socket n'est attachée à aucune adresse IP (ni port).

Exemple de création d'une socket en mode connecté (TCP)

Dans l'exemple qui suit, on va simplement **créer une socket** en mode connecté (**TCP**). De plus, on va utiliser la commande **lsof** pour afficher les descripteurs du processus en cours d'exécution.

Le code du programme (fichier **CreationSocket.cpp**) est

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

int main()
{
  int s;

  printf("pid = %d\n",getpid());

  if ((s = socket(AF_INET, SOCK_STREAM, 0)) == -1)
  {
    perror("Erreur de socket()");
    exit(1);
  }

  printf("socket creee = %d\n",s);

  pause();
}
```

dont un exemple d'exécution fournit

```
# CreationSocket &
[1] 89871
pid = 89871
socket creee = 3
# lsof -p 89871 -ad "0-10"
COMMAND     PID    USER   FD   TYPE DEVICE SIZE/OFF   NODE NAME
CreationS 89871 student    0u   CHR  136,0      0t0      3 /dev/pts/0
CreationS 89871 student    1u   CHR  136,0      0t0      3 /dev/pts/0
CreationS 89871 student    2u   CHR  136,0      0t0      3 /dev/pts/0
CreationS 89871 student    3u  sock    0,9      0t0 425381 protocol: TCP
# kill -2 89871
[1]+  Interrompre             CreationSocket
#
```

On observe que

- Le descripteur d'une socket se comporte exactement comme les descripteurs des fichiers ouverts

- L'appel système **socket()** a attribué à la socket le premier descripteur libre dans la table des descripteurs
- Le descripteur 3 associé à la socket correspond à un fichier particulier de type « `socket` » associé au « `protocol : TCP` »

Actuellement,

- L'appel système **socket()** a juste réservé une entrée dans la table des descripteurs
- Aucun dialogue réseau n'a eu lieu
- La socket n'est associée à aucune adresse IP ni port.

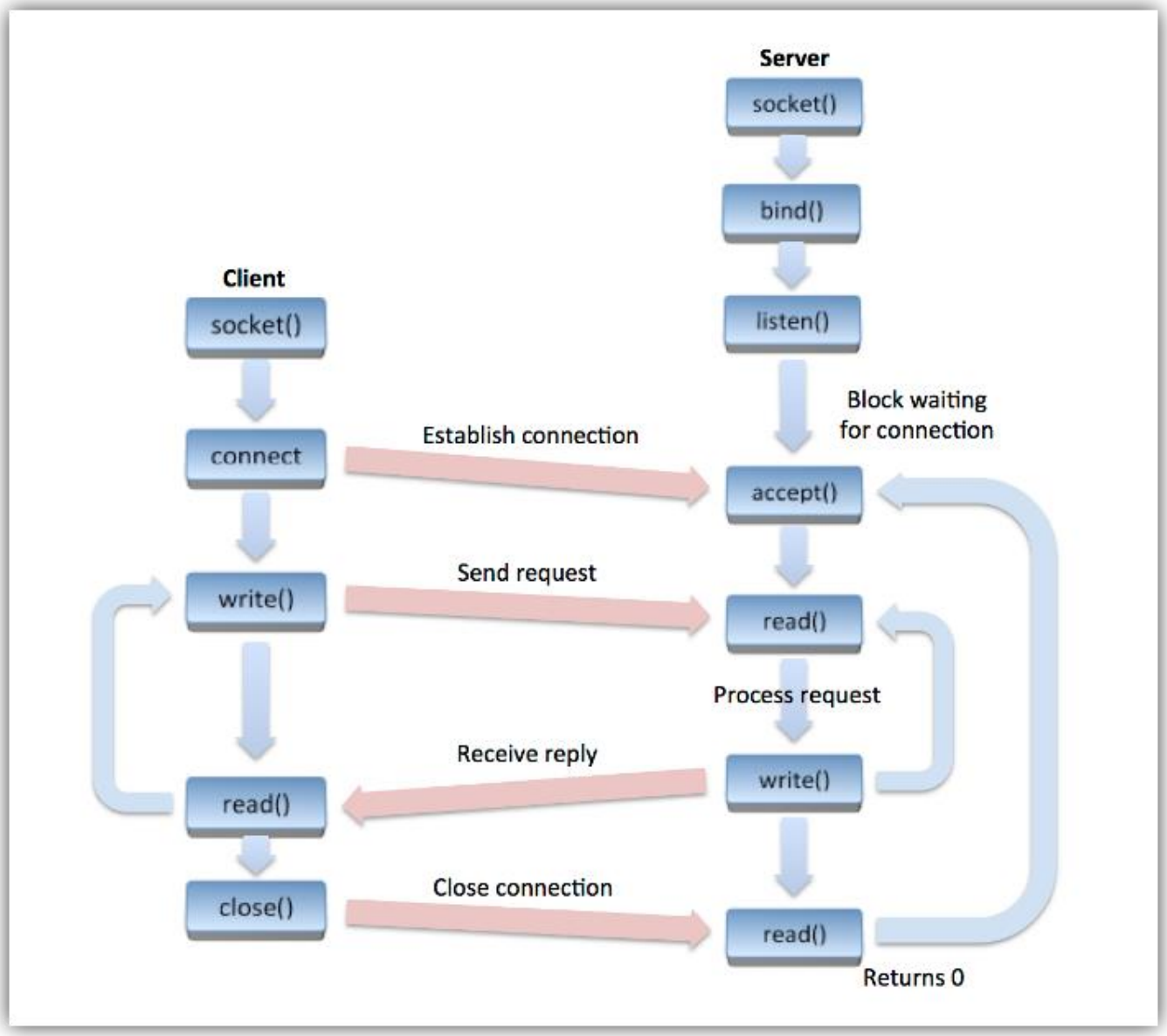
Avant de pouvoir l'utiliser, il faut lui associer un couple (adresse IP, port).

Cycle de vie des sockets dans une communication TCP

Pour communiquer, 2 processus doivent donc créer chacun une **socket**. Parmi ces deux processus, on distingue :

- Le processus « **serveur** » qui va réaliser une **connexion « passive »**, c'est-à-dire qu'il va attendre qu'un autre processus (le « **client** ») se connecte avec lui
- Le processus « **client** » qui va réaliser une **connexion « active »** en décidant de se connecter avec un processus « **serveur** »

Plusieurs **appels systèmes** sont nécessaires pour cela. Le schéma suivant illustre le cycle de vie des sockets en rapport avec ces appels systèmes :



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

Une fois que la socket a été créée, le processus **serveur** doit

- Lier sa socket avec une adresse IP et un port sur lequel il souhaite être contacté → appel système **bind()**
- Prévenir le système d'exploitation qu'il souhaite réaliser une connexion passive et se mettre à l'écoute d'une demande de connexion → appel système **listen()** → ceci va activer la machine à états TCP qui passe à l'état **LISTEN**
- Commencer à attendre et accepter les demandes de connexions → appel système **accept()**

De son côté, une fois la socket créée, le processus **client** doit

- Faire une demande de connexion pour le serveur visé et dont il connaît l'adresse IP et le port → appel système **connect()**
- Le processus client n'a pas besoin de faire appel à **bind()** et de lier sa socket à une adresse IP et un port local (au client) → le système d'exploitation va automatiquement lui attribuer un port

Une fois la connexion établie, les échanges de données (**requêtes** envoyées par le client, **réponses** envoyées par le serveur) se réalisent grâce aux appels système bien connus **write()** et **read()**.

Une fois la communication terminée, l'appel système **close()** permet de fermer la connexion.

Lier une socket à une adresse (IP + port)

Comme déjà mentionné, cette phase

- est **nécessaire** dans le cas du processus **serveur** → il décide sur quelle adresse IP et sur quel port il souhaite être contacté
- n'est **pas nécessaire** mais possible pour le processus **client** → le système d'exploitation va lui attribuer une adresse IP et un port local

Cela se fait via l'appel système

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

où

- **sockfd** est le descripteur de la socket que l'on désire lier
- **addr** est un pointeur vers une structure contenant l'adresse IP et le port souhaité → il s'agit en fait ici d'un pointeur générique → il pointera vers une structure spécifique correspondant à la famille d'adresse utilisée (AF_INET, AF_INET6, …)
- **addrlen** est la taille (en octets) de la structure pointée par **addr**

Dans le cas présent (mode connecté – TCP – domaine AF_INET), l'adresse utilisée sera décrite à l'aide de la structure **sockaddr_in** :

```
struct sockaddr_in {
    sa_family_t    sin_family;  // Famille d'adresses (AF_INET)
    in_port_t      sin_port;    // Numéro de port
    struct in_addr sin_addr;    // Adresse IP
    char           sin_zero[8]; // Remplissage pour alignement
};
```

La structure **in_addr** représente donc une adresse IP et est définie par

```
struct in_addr {
    in_addr_t s_addr; // Adresse IP en format réseau
};
```

Il est possible de remplir ces structures champ par champ mais ce n'est pas chose aisée car les données doivent être encodées sous la forme du réseau (l'ordre des octets pour les différents types de données n'est pas le même d'une machine à l'autre) en utilisant des fonctions de conversion.

Résolution de nom

Le plus simple pour remplir ces structures est d'utiliser le mécanisme de « **résolution de nom** » qui fait correspondre un nom d'hôte convivial (par exemple www.google.be) à une adresse IP numérique (ici 142.251.36.3). Cette résolution de nom est réalisée par le système de noms de domaine (**DNS** – **D**omain **N**ame **S**ystem) qui est une infrastructure distribuée sur Internet. Lorsqu'une application doit résoudre un nom en adresse IP :

1. elle envoie une requête DNS à un serveur DNS.
2. le serveur DNS reçoit la requête, recherche le nom demandé dans sa base de données et renvoie l'adresse IP à l'application
3. Elle peut ensuite utiliser cette adresse IP pour établir la connexion réseau

Dans notre cas, il suffira d'appeler la fonction

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *host,
                const char *service,
                const struct addrinfo *hints,
                struct addrinfo **result);
```

où

- **host** est le nom de la machine hôte (exemples : « www.google.be » , « moon », …) ou une adresse IP (exemple : « 142.251.36.3 », …) fourni sous la forme d'une chaîne de caractères
- **service** est le service (exemple : « http ») ou le numéro de port (exemples : « 80 », « 50000 », …) fourni sous la forme d'une chaîne de caractères
- **hints** est une structure contenant les détails de notre recherche
- **result** est le résultat de la recherche (le résultat de la fonction) fourni sous la forme d'une liste chainée de structures addrinfo

La structure addrinfo est définie par :

```
struct addrinfo {
  int              ai_flags;
  int              ai_family;
  int              ai_socktype;
  int              ai_protocol;
  socklen_t        ai_addrlen;
  struct sockaddr *ai_addr;
  char            *ai_canonname;
  struct addrinfo *ai_next;
};
```

Où

- **ai_family** est la famille d'adresses souhaitées (par exemple AF_INET)
- **ai_socktype** est le type de socket souhaitée (par exemple SOCK_STREAM ou SOCK_DGRAM)
- **ai_protocol** est le protocole utilisé (par exemple IPPROTO_TCP ou IPPROTO_UDP, on peut mettre 0 s'il n'y a pas d'ambiguïté possible)
- **ai_flags** contient un ou des flags (combinés par |) :

- **AI_NUMERICHOST** : si l'argument host contient une adresse IP numérique plutôt qu'un nom d'hôte
- **AI_NUMERICSERV** : si l'argument service contient un numéro de port et non pas un nom de service
- **AI_PASSIVE** : si l'adresse demandée sera utilisée pour y attacher un serveur avec **bind()**
- …

Lors du retour de la fonction, le pointeur **result** est initialisé au début d'une liste chainée de toutes les structures addrinfo correspondant à notre demande. La liste est chainée à l'aide du champ **ai_next**.

Lors de l'appel de cette fonction, **host** ou **service** peut être NULL mais pas les deux simultanément.

Les champs remplis par l'appel de la fonction sont

- **ai_addr** est un pointeur sur la structure d'adresse convoitée (ce que l'on passera au second argument de l'appel système **bind()**)
- **ai_addrlen** est la taille de cette structure (ce que l'on passera en 3ème paramètre de l'appel système **bind()**).

Notons qu'après consultation des résultats de la recherche, la liste chainée doit être libérée par la fonction

```
void freeaddrinfo(struct addrinfo *res);
```

Il est également possible de faire le chemin inverse, c'est-à-dire de retrouver les informations « lisibles par l'être humain » à partir d'une structure sockaddr(_in). Pour cela, on peut utiliser la fonction

```
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo(const struct sockaddr *addr, socklen_t addrlen,
                char *host, socklen_t hostlen,
                char *serv, socklen_t servlen,
                int flags);
```

où

- **addr** est un ponteur vers la structure de l'adresse à analyser

- **addrlen** est la taille de cette structure
- **host** est l'adresse de la chaine de caractères qui sera remplie par la fonction avec le nom de l'hôte ou l'adresse IP
- **hostlen** est la taille de la chaine de caractères précédente
- **serv** est l'adresse de la chaine de caractères qui sera remplie par la fonction avec le nom du service ou le numéro de port
- **servlen** est la taille de cette chaine de caractères
- **flags** contient un ou des flags (combinés avec |) :
    - **NI_NUMERICSERV** si on veut obtenir le numéro de port et non le nom du service
    - **NI_NUMERICHOST** si on veut obtenir l'adresse IP et non le nom de l'hôte
    - …

Pour la taille des chaines de caractères, on peut utiliser les constantes **NI_MAXHOST** et **NI_MAXSERV** pour éviter des débordements mémoire.

Exemple de résolution de nom

Dans l'exemple qui suit, on teste les fonctions **getaddrinfo()** et **getnameinfo()** sur

- L'hôte www.google.be et le service http
- L'adresse IP 192.168.228.167 (machine moon) et le port 80

Le code du programme (fichier **TestResolutionNom.cpp**) est

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

int main()
{
  // Pour la recherche
  struct addrinfo hints;
  struct addrinfo *results;

  // Pour l'affichage des resultats
  char host[NI_MAXHOST];
  char port[NI_MAXSERV];
  struct addrinfo* info;
```

```
  // On fournit l'hote et le service
  memset(&hints,0,sizeof(struct addrinfo)); // initialisation à 0
  hints.ai_family = AF_INET;
  hints.ai_socktype = SOCK_STREAM;
  printf("Pour www.google.be avec le service http :\n");
  if (getaddrinfo("www.google.be","http",&hints,&results) != 0)
    printf("Erreur de getaddrinfo");
  else
  {
    // Affichage du contendu des adresses obtenues au format numérique
    for (info = results ; info != NULL ; info = info->ai_next)
    {
      getnameinfo(info->ai_addr,info->ai_addrlen,
                  host,NI_MAXHOST,
                  port,NI_MAXSERV,
                  NI_NUMERICSERV | NI_NUMERICHOST);
      printf("Adresse IP: %s -- Port: %s\n",host,port);
    }
    freeaddrinfo(results);
  }

  // On fournit l'adresse IP et le port directement
  memset(&hints,0,sizeof(struct addrinfo)); // initialisation à 0
  hints.ai_family = AF_INET;
  hints.ai_socktype = SOCK_STREAM;
  hints.ai_flags = AI_NUMERICSERV | AI_NUMERICHOST;
  printf("Pour 192.168.228.167 avec le port 80 :\n");
  if (getaddrinfo("192.168.228.167","80",&hints,&results) != 0)
    printf("Erreur de getaddrinfo");
  else
  {
    // Affichage du contendu des adresses obtenues au format "hote" et "service"
    for (info = results ; info != NULL ; info = info->ai_next)
    {
      getnameinfo(info->ai_addr,info->ai_addrlen,
                  host,NI_MAXHOST,
                  port,NI_MAXSERV,
                  0);
      printf("Hote: %s -- Service: %s\n",host,port);
    }
    freeaddrinfo(results);
  }

  exit(0);
}
```

dont un exemple d'exécution fournit

```
# TestResolutionNom
Pour www.google.be avec le service http :
Adresse IP: 142.251.39.99 -- Port: 80
Pour 192.168.228.167 avec le port 80 :
Hote: moon -- Service: http
#
```

Exemple de liaison d'une socket avec une adresse : bind()

Dans l'exemple qui suit, on va

- créer une socket
- préparer une adresse (IP + port) en vue d'une **connexion passive** (pour un futur processus **serveur**) pour une communication TCP
- lier la socket créée avec l'adresse construite avec l'appel système **bind()**

Le code du programme (fichier **BindSocket.cpp**) est

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

int main()
{
  int sServeur;

  printf("pid = %d\n",getpid());

  // Creation de la socket
  if ((sServeur = socket(AF_INET, SOCK_STREAM, 0)) == -1)
  {
    perror("Erreur de socket()");
    exit(1);
  }
  printf("socket creee = %d\n",sServeur);

  // Construction de l'adresse
  struct addrinfo hints;
  struct addrinfo *results;
  memset(&hints,0,sizeof(struct addrinfo));
  hints.ai_family = AF_INET;
  hints.ai_socktype = SOCK_STREAM;
  hints.ai_flags = AI_PASSIVE | AI_NUMERICSERV; // pour une connexion passive
  if (getaddrinfo(NULL,"50000",&hints,&results) != 0)
  {
      close(sServeur);
      exit(1);
  }

  // Affichage du contenu de l'adresse obtenue (optionnel, c'est juste pour info)
  char host[NI_MAXHOST];
  char port[NI_MAXSERV];
  struct addrinfo* info;
  getnameinfo(results->ai_addr,results->ai_addrlen,
              host,NI_MAXHOST,port,NI_MAXSERV,
```

```
                NI_NUMERICSERV | NI_NUMERICHOST);
  printf("Mon Adresse IP: %s -- Mon Port: %s\n",host,port);

  // Liaison de la socket à l'adresse
  if (bind(sServeur,results->ai_addr,results->ai_addrlen) < 0)
  {
      perror("Erreur de bind()");
      exit(1);
  }
  freeaddrinfo(results);
  printf("bind() reussi !\n");

  pause();
}
```

dont un exemple d'exécution fournit

```
# BindSocket &
[1] 95519
pid = 95519
socket creee = 3
Mon Adresse IP: 0.0.0.0 -- Mon Port: 50000
bind() reussi !
# netstat -an | grep 50000
#
```

On observe que

- on a passé **NULL** comme premier argument de la fonction **getaddrinfo()**. Combiné avec le flag **AI_PASSIVE** dans la structure **hints** permet d'obtenir une adresse de serveur **0.0.0.0**, que l'on nomme également **INADDR_ANY**, et qui correspond à une (future) écoute sur toutes les interfaces réseaux disponibles
- l'appel de la commande **netstat** permet d'afficher les connexions en cours. Ici en particulier, on a demandé d'afficher les connexions en cours pour le port 50000. On remarque qu'il n'y aucun résultat. Cela signifie simplement que la socket a bien été liée à l'adresse **0.0.0.0:50000** mais que la socket n'est pas encore pas encore « active » sur le réseau.

Attente d'une connexion : listen() et accept()

L'appel système listen()

On se situe toujours au niveau du processus **serveur** qui vient de lier sa socket à une adresse réseau (IP + port). Il doit à présent prévenir le système d'exploitation qu'il va se mettre en attente d'une demande de connexion sur la socket. Pour cela, on doit utiliser l'appel système

```
#include <sys/types.h>
#include <sys/socket.h>

int listen(int sockfd, int count);
```

où

- **sockfd** est la socket sur laquelle on souhaite attendre des connexions
- **count** est un paramètre qui spécifie le nombre maximum de connexions qui peuvent être reçues par le serveur mais qui n'ont pas encore été prises en compte par celui-ci au moyen de l'appel système **accept()** (voir plus loin) : de telles demandes sont appelées **connexions pendantes** et sont enfilées dans une FIFO. La taille maximum de cette FIFO est fixée par la constante **SOMAXCONN**

L'appel de cette fonction **n'est pas bloquant** et retourne

- 0 en cas succès
- -1 en cas d'erreur et ERRNO est positionné. Une valeur particulièrement intéressante de ERRNO est **EADDRINUSE** qui signifie qu'une autre socket est déjà à l'écoute sur le même port

L'appel système accept()

Le noyau étant maintenant prévu, le processus serveur peut effectivement se mettre en attente sur une connexion en utilisant l'appel système **bloquant**

```
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

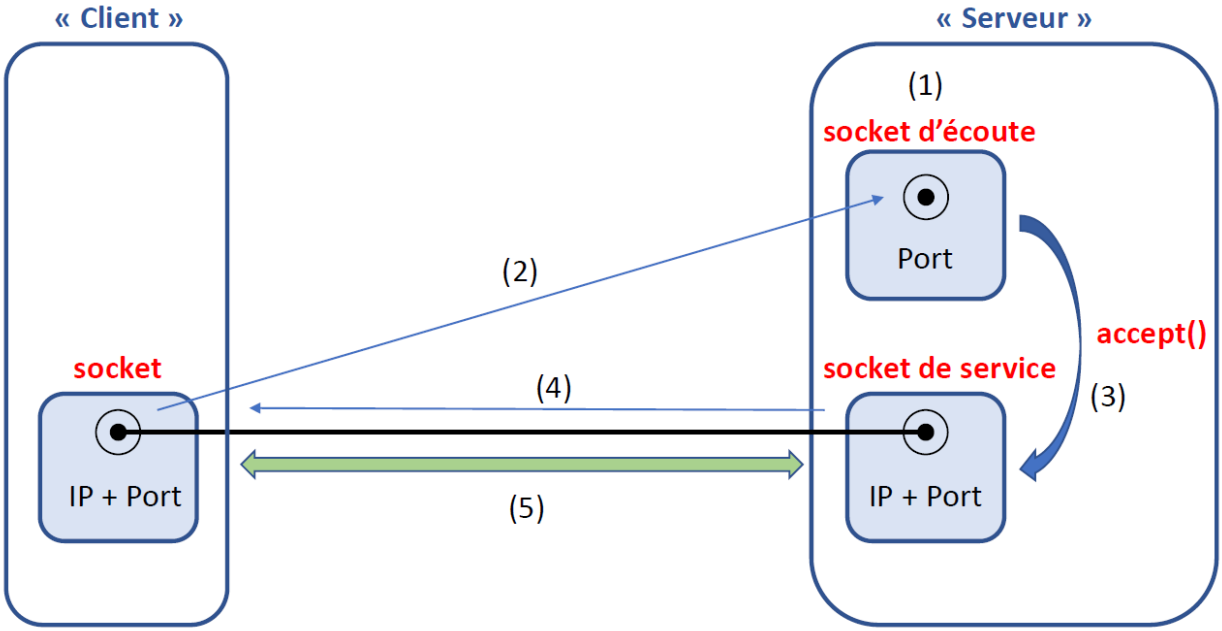
où

- **sockfd** est la socket sur laquelle on désire attendre une connexion
- **addr** est un pointeur vers une structure d'adresse réseau correspondant à celle du client qui vient de se connecter sur le serveur (on parle de **« peer socket »**) → celle-ci sera remplie par la fonction au moment de l'acception de la connexion par le serveur
- **addrlen** est la taille de cette structure → elle est passée par adresse à la fonction car elle va être initialisée par la fonction

Cette fonction est **bloquante** (sauf s'il y a des connexions pendantes) jusqu'au moment où un client se connecte sur le serveur (voir plus loin) et retourne

- -1 en cas d'erreur, mais surtout
- Un **entier positif** correspondant à une **socket dupliquée** → ceci a pour effet d'ouvrir une nouvelle socket côté serveur, appelée **socket de service**, qui est mise en connexion avec le client. La socket originale, appelée **socket d'écoute**, est restée intacte et est prête à servir à nouveau pour une demande de connexion (listen() + accept())

Schématiquement, nous avons



1) Le processus serveur crée une socket, réalise un **bind()** et un **listen()** et se met en attente sur **accept()**
2) Le client réalise un appel système **connect()** (voir plus loin)
3) L'appel système **accept()** se débloque et retourne une **socket de service** mise en connexion avec le client

4) Le client est prévenu et l'appel système **connect()** lui fournit une **socket de service** mise en connexion avec le serveur
5) Les processus client et serveur peuvent communiquer à l'aide des appels systèmes **read()** et **write()** (voir plus loin)

Notez que la **socket dupliquée** (la **socket de service**) permet simplement d'envoyer et de recevoir des données sur la connexion ; elle ne permet pas d'accepter de nouvelles connexions.

Une fois la socket de service obtenue par le serveur, une autre manière d'obtenir les informations (adresse IP + port) sur le client connecté est d'utiliser la fonction

```
#include <sys/socket.h>

int getpeername(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

où

- **sockfd** est la socket de service obtenue lors d'un appel système **accept()**
- **addr** est l'adresse une structure qui va recevoir l'adresse réseau du client connecté
- **addrlen** est la taille de cette structure → doit être initialisée à la taille de la structure

Une fois exécutée, il suffit d'utiliser la fonction **getnameinfo()** vue plus haut pour récupérer les informations souhaitées.

Exemple d'attente de connexion : listen() et accept()

Dans l'exemple qui suit, on reprend l'exemple précédent mais on met le serveur en attente sur le port 50000 grâce aux appels système **listen()** et **accept()**.

Le code du programme (fichier **AttenteConnexion.cpp**) est

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

```c
int main()
{
  int sEcoute;

  printf("pid = %d\n",getpid());

  // Creation de la socket
  if ((sEcoute = socket(AF_INET, SOCK_STREAM, 0)) == -1)
  {
    perror("Erreur de socket()");
    exit(1);
  }
  printf("socket creee = %d\n",sEcoute);

  // Construction de l'adresse
  struct addrinfo hints;
  struct addrinfo *results;
  memset(&hints,0,sizeof(struct addrinfo));
  hints.ai_family = AF_INET;
  hints.ai_socktype = SOCK_STREAM;
  hints.ai_flags = AI_PASSIVE | AI_NUMERICSERV;   // pour une connexion passive
  if (getaddrinfo(NULL,"50000",&hints,&results) != 0)
    exit(1);

  // Affichage du contenu de l'adresse obtenue
  char host[NI_MAXHOST];
  char port[NI_MAXSERV];
  getnameinfo(results->ai_addr,results->ai_addrlen,
              host,NI_MAXHOST,port,NI_MAXSERV,NI_NUMERICSERV | NI_NUMERICHOST);
  printf("Mon Adresse IP: %s -- Mon Port: %s\n",host,port);

  // Liaison de la socket à l'adresse
  if (bind(sEcoute,results->ai_addr,results->ai_addrlen) < 0)
  {
      perror("Erreur de bind()");
      exit(1);
  }
  freeaddrinfo(results);
  printf("bind() reussi !\n");

  // Mise à l'écoute de la socket
  if (listen(sEcoute,SOMAXCONN) == -1)
  {
    perror("Erreur de listen()");
    exit(1);
  }
  printf("listen() reussi !\n");

  // Attente d'une connexion
  int sService;
  if ((sService = accept(sEcoute,NULL,NULL)) == -1)
  {
    perror("Erreur de accept()");
    exit(1);
  }
```

```
  printf("accept() reussi !");
  printf("socket de service = %d\n",sService);

  // Recuperation d'information sur le client connecte
  struct sockaddr_in adrClient;
  socklen_t adrClientLen = sizeof(struct sockaddr_in); // nécessaire
  getpeername(sService,(struct sockaddr*)&adrClient,&adrClientLen);
  getnameinfo((struct sockaddr*)&adrClient,adrClientLen,
              host,NI_MAXHOST,
              port,NI_MAXSERV,
              NI_NUMERICSERV | NI_NUMERICHOST);
  printf("Client connecte --> Adresse IP: %s -- Port: %s\n",host,port);

  pause();
}
```

Dans l'exemple d'exécution qui suit, nous allons

- Lancer le processus **AttentionConnexion** sur la machine **zeus** dont l'adresse IP est **192.168.228.169**
- Nous connecter sur ce processus à partir de la machine **moon** dont l'adresse IP est **192.168.228.167** en utilisant la commande **telnet** (protocole utilisé sur tout réseau TCP/IP, permettant de communiquer avec un serveur distant en échangeant des lignes de texte et en recevant également des lignes de texte)

```
[zeus]# AttenteConnexion &
[1] 27545
pid = 27545
socket creee = 3
Mon Adresse IP: 0.0.0.0 -- Mon Port: 50000
bind() reussi !
listen() reussi !
[zeus]# lsof -p 27545 -ad "0-10"
COMMAND     PID    USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
AttenteCo 27545 student    0u   CHR  136,1      0t0    4 /dev/pts/1
AttenteCo 27545 student    1u   CHR  136,1      0t0    4 /dev/pts/1
AttenteCo 27545 student    2u   CHR  136,1      0t0    4 /dev/pts/1
AttenteCo 27545 student    3u  IPv4 365545      0t0  TCP *:50000 (LISTEN)
[zeus]# netstat -an | grep 50000
tcp        0      0 0.0.0.0:50000           0.0.0.0:*               LISTEN
[zeus]#
```

On observe que

- Le processus est bloqué sur l'appel système **accept()**

- La commande **lsof** permet d'afficher les sockets du processus → on voit que le type de socket est IPv4, le domaine est TCP et qu'elle est en écoute (LISTEN) sur le port 50000
- La commande **netstat** a à présent fourni un résultat : un processus est en attente (état **LISTEN**) sur le port 50000 (adresse 0.0.0.0:50000).

Sur la machine **moon**, nous lançons à présent la commande telnet :

```
[moon]# telnet 192.168.228.169 50000
Trying 192.168.228.169...
Connected to 192.168.228.169.
Escape character is '^]'.
```

On observe que **telnet** s'est bien connecté sur notre processus serveur. Dans un autre terminal de la machine **moon** :

```
[moon]# netstat -an | grep 50000
tcp        0      0 192.168.228.167:46948   192.168.228.169:50000   ESTABLISHED
[moon]#
```

On observe que

- la machine moon a attribué automatiquement le port **46948** au processus **telnet** connecté
- Au niveau du client **telnet**, la socket est dans l'état **ESTABLISHED** et on observe bien que l'adresse distance (**192.168.288.169:50000**) est bien celle de notre processus serveur

Dans le terminal du processus serveur :

```
[zeus]#
accept() reussi !
socket de service = 4
Client connecte --> Adresse IP: 192.168.228.167 -- Port: 46948

[zeus]# lsof -p 27545 -ad "0-10"
COMMAND     PID    USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
AttenteCo 27545 student    0u   CHR  136,1      0t0    4 /dev/pts/1
AttenteCo 27545 student    1u   CHR  136,1      0t0    4 /dev/pts/1
AttenteCo 27545 student    2u   CHR  136,1      0t0    4 /dev/pts/1
AttenteCo 27545 student    3u  IPv4 365545      0t0  TCP *:50000 (LISTEN)
AttenteCo 27545 student    4u  IPv4 365546  0t0 TCP zeus:50000->192.168.228.167:46948
(ESTABLISHED)
[zeus]# netstat -an | grep 50000
tcp        0      0 0.0.0.0:50000           0.0.0.0:*               LISTEN
```

```
tcp        0      0 192.168.228.169:50000   192.168.228.167:46948   ESTABLISHED
[zeus]# kill -2 27545
[1]+  Interrompre                ./AttenteConnexion
[zeus]# netstat -an | grep 50000
tcp        0      0 192.168.228.169:50000   192.168.228.167:46948   TIME_WAIT
[zeus]# netstat -an | grep 50000
[zeus]#
```

On observe que

- L'appel système **accept()** s'est débloqué et a retourné la **socket de service** dont le descripteur vaut **4**
- L'adresse réseau obtenue par la fonction **getpeername()** correspond bien à l'adresse réseau du client connecté (**192.168.228.167:46948**).
- La commande **lsof** indique bien la présence de 2 sockets au niveau du processus serveur : la **socket d'écoute** (**3**) restée dans l'état **LISTEN** et la **socket dupliquée** (**de service**) (**4**) qui correspond à la connexion établie (**ESTABLISHED**) avec le client
- La commande **netstat** nous donne la même information
- L'envoi du signal SIGINT (2) au processus **AttenteConnexion** l'interrompt mais la socket reste bel et bien dans le système dans l'état **TIME_WAIT**. Il faudra attendre 60 secondes pour que cette socket disparaisse complètement (dernier appel de **netstat**) et que le port soit à nouveau réutilisable

Demander d'une connexion : connect()

A partir du moment où notre processus **serveur** est en attente d'une connexion, notre processus **client** peut demander une connexion sur celui-ci. Pour cela, il doit utiliser l'appel système

```
#include <sys/types.h>
#include <sys/socket.h>

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

où

- **sockfd** est une socket créée grâce à l'appel de **socket()** et qui sera associée à la future connexion avec notre processus serveur

- **addr** est l'adresse d'une structure contenant l'adresse réseau du processus serveur que l'on désire contacter
- **addrlen** est la taille de cette structure

Si tout s'est bien passé,

- le retour de la fonction est 0
- si aucun **bind()** n'a été réalisé par le processus appelant (notre **client**), le système attribue automatiquement un port local à la socket désignée par **sockfd**
- la communication est établie avec le processus serveur dont l'appel système **accept()** s'est débloqué (voir plus haut)

En cas d'erreur la fonction retourne -1.

Exemple de demande de connexion : connect()

Dans l'exemple qui suit, on reprend notre programme **AttenteConnexion** qui va se mettre en attente d'une connexion sur la machine **zeus** (IP = **192.168.228.169**) sur le port **50000**. Nous créons un processus **client** qui va tenter de se connecter sur notre processus serveur à partir de la machine **moon** (IP = **192.168.226.167**).

Le code du programme **client** (fichier **DemandeConnexion.cpp**) est

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

int main(int argc,char* argv[])
{
  if (argc != 3)
  {
    printf("Erreur, usage :\n");
    printf("DemandeConnexion ipServeur portServeur\n");
    exit(1);
  }

  int sClient;
  printf("pid = %d\n",getpid());

  // Creation de la socket
  if ((sClient = socket(AF_INET, SOCK_STREAM, 0)) == -1)
  {
```

```
    perror("Erreur de socket()");
    exit(1);
  }
  printf("socket creee = %d\n",sClient);

  // Construction de l'adresse du serveur
  struct addrinfo hints;
  struct addrinfo *results;
  memset(&hints,0,sizeof(struct addrinfo));
  hints.ai_family = AF_INET;
  hints.ai_socktype = SOCK_STREAM;
  hints.ai_flags = AI_NUMERICSERV;
  if (getaddrinfo(argv[1],argv[2],&hints,&results) != 0)
    exit(1);

  // Demande de connexion
  if (connect(sClient,results->ai_addr,results->ai_addrlen) == -1)
  {
      perror("Erreur de connect()");
      exit(1);
  }
  printf("connect() reussi !");

  pause();
}
```

Nous observons que

- la socket client est créée avec l'appel système **socket()** comme pour celle du processus serveur
- l'adresse IP et le port du serveur que l'on veut contacter sont fournis en ligne de commande par **argv[1]** et **argv[2]**
- la fonction **getaddrinfo()** permet de construire la structure de l'adresse réseau du serveur que l'on veut contacter

Dans la console du **serveur**, nous avons

```
[zeus]# AttenteConnexion &
[1] 9363
pid = 9363
socket creee = 3
Mon Adresse IP: 0.0.0.0 -- Mon Port: 50000
bind() reussi !
listen() reussi !
[zeus]# netstat -an | grep 50000
tcp        0      0 0.0.0.0:50000           0.0.0.0:*               LISTEN
[zeus]#
accept() reussi !
socket de service = 4
Client connecte --> Adresse IP: 192.168.228.167 -- Port: 47070
```

```
[zeus]# netstat -an | grep 50000
tcp        0      0 0.0.0.0:50000           0.0.0.0:*               LISTEN
tcp        0      0 192.168.228.169:50000   192.168.228.167:47070   ESTABLISHED
[zeus]#
```

tandis que sur la console du **client**, nous avons

```
[moon]# DemandeConnexion 192.168.228.169 50000
pid = 110547
socket creee = 3
Erreur de connect(): Connection refused
[moon]# DemandeConnexion 192.168.228.169 50000 &
[1] 110566
pid = 110566
socket creee = 3
[moon]# lsof -p 110566 -ad "0-10"
COMMAND      PID    USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
DemandeCo 110566 student    0u   CHR  136,0      0t0    3 /dev/pts/0
DemandeCo 110566 student    1u   CHR  136,0      0t0    3 /dev/pts/0
DemandeCo 110566 student    2u   CHR  136,0      0t0    3 /dev/pts/0
DemandeCo 110566 student    3u IPv4 632084 0t0  TCP moon:47070->192.168.228.169:50000
(ESTABLISHED)
[moon]# netstat -an | grep 50000
tcp        0      0 192.168.228.167:47070   192.168.228.169:50000   ESTABLISHED
[moon]#
```

On observe que

- Au niveau serveur, aucune différence par rapport à l'exemple précédent. Celui-ci ne fait aucune différence entre **telnet** et notre programme **DemandeConnexion**
- Lors du premier lancement de **DemandeConnexion**, le serveur n'était pas encore lancé, d'où l'erreur. C'est évident mais il est donc bien nécessaire de toujours lancer le processus serveur avant le processus client…
- Une fois connecté, le système a attribué à la socket du client le descripteur **3** et automatiquement le port **47070** à la socket

Echanger des données en TCP et fermeture d'une connexion

Une fois la connexion établie entre un processus **serveur** et un processus **client**, ils peuvent s'échanger des données à l'aide des appels système

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t send(int sockfd, const void *buf, size_t len, int flags);
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

où

- **sockfd** est la socket de service du serveur ou du client
- **buf** est l'adresse
  - du « paquet de bytes » que l'on désire envoyer dans le cas du **send**
  - d'un buffer de réception dans le cas du **recv**
- **len** est la taille
  - du « paquet de bytes » que l'on désire envoyer dans le cas du **send**
  - du buffer de réception dans le cas du **recv**
- **flags** est un flag qui permet modifier le comportement des fonctions **send** et **recv**. flags est rarement utilisé en pratique sauf dans certains cas qui nous intéressent peu ici. S'il n'est pas utilisé, il peut être choisi égal à **0**.

Mais il faut que remarquer que lorsque **flags** est choisi égal à 0, les appels systèmes **send()** et **recv()** se comportement exactement de la même manière que les appels systèmes bien connus

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
ssize_t read(int fd, void *buf, size_t count);
```

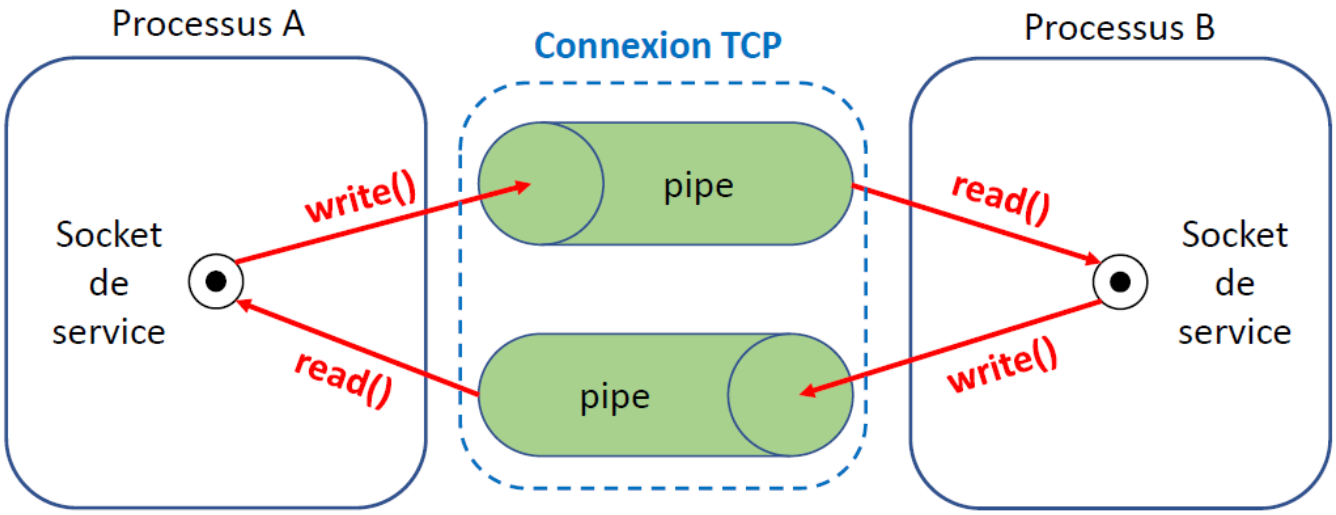
où

- **fd** est le descripteur de la socket de service du client ou du serveur
- **buf** a le même rôle que dans les appels système **send()** et **recv()**
- **count** à le même rôle que **len** dans les appels système **send()** et **recv()**

Il suffira alors, au niveau programmation, de voir la **communication TCP établie** comme une **communication par pipes** classiques, à la différence près que

- une socket est **bidirectionnelle** : on écrit et lit sur le même descripteur de socket
- il y a **2 pipes** : un dans le sens **client → serveur** et un dans le sens **serveur → client**

Schématiquement, cela correspond à



L'appel système close()

Une fois la communication terminée, il est nécessaire de fermer correctement les sockets de service au niveau du **serveur** ET du **client**. Pour cela, on utilise l'appel système

```
#include <unistd.h>

int close(int fd);
```

où **fd** est la socket que l'on désire fermer.

Si cela n'est pas fait, les sockets de service risquent de rester dans des états comme **CLOSE_WAIT**, **FIN_WAIT1**, … Et il faudra alors attendre un certain temps que le système libère correctement le port avant de pouvoir le réutiliser.

Evidemment, il sera également nécessaire de fermer la socket d'écoute du **serveur** lorsqu'il ne souhaitera plus accepter de nouvelles connexions.

Remarque

Une fermeture plus fine des sockets peut être réalisée à l'aide de l'appel système

```
#include <sys/socket.h>

int shutdown(int sockfd, int how);
```

où

- **sockfd** est la socket que l'on désire fermer
- **how** est un flag pouvant prendre les valeurs :
  - **SHUT_RD** : aucune lecture ne sera plus possible → l'appel système **read()** retournera 0
  - **SHUT_WR** : aucune écriture ne sera plus possible → un appel à **write()** provoquera la réception du signal **SIGPIPE** par le processus
  - **SHUT_RDWR** : plus aucune écriture/lecture ne sera plus possible. Cela correspond à l'appel de **close()**

Exemple d'échanges de donnés : read() et write()

Dans l'exemple qui suit,

- Un processus **serveur** va se mettre en écoute sur le port **50000**.
- Le **serveur** répondra au client en concaténant à la chaîne de caractères reçues la chaîne « **[SERVEUR]** » en préfixe
- Une fois répondu, le **serveur** fermera sa socket d'écoute et sa socket de service
- Le processus **client** va simplement se connecter sur le processus **serveur**, envoyer une chaine de caractères, attendre et afficher la réponse du **serveur**, avant de fermer sa socket de service

Voici le code du **serveur** (fichier **Serveur.cpp**) :

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

int main()
{
  int sEcoute;
```

```c
printf("pid = %d\n",getpid());

// Creation de la socket
if ((sEcoute = socket(AF_INET, SOCK_STREAM, 0)) == -1)
{
  perror("Erreur de socket()");
  exit(1);
}
printf("socket creee = %d\n",sEcoute);

// Construction de l'adresse
struct addrinfo hints;
struct addrinfo *results;
memset(&hints,0,sizeof(struct addrinfo));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE | AI_NUMERICSERV;   // pour une connexion passive
if (getaddrinfo(NULL,"50000",&hints,&results) != 0)
  exit(1);

// Affichage du contenu de l'adresse obtenue
char host[NI_MAXHOST];
char port[NI_MAXSERV];
getnameinfo(results->ai_addr,results->ai_addrlen,
            host,NI_MAXHOST,port,NI_MAXSERV,
            NI_NUMERICSERV | NI_NUMERICHOST);
printf("Mon Adresse IP: %s -- Mon Port: %s\n",host,port);

// Liaison de la socket à l'adresse
if (bind(sEcoute,results->ai_addr,results->ai_addrlen) < 0)
{
    perror("Erreur de bind()");
    exit(1);
}
freeaddrinfo(results);
printf("bind() reussi !\n");

// Mise à l'écoute de la socket
if (listen(sEcoute,SOMAXCONN) == -1)
{
  perror("Erreur de listen()");
  exit(1);
}
printf("listen() reussi !\n");

// Attente d'une connexion
int sService;
if ((sService = accept(sEcoute,NULL,NULL)) == -1)
{
  perror("Erreur de accept()");
  exit(1);
}
printf("accept() reussi !\n");
printf("socket de service = %d\n",sService);

// Recuperation d'information sur le client connecte
```

```
  struct sockaddr_in adrClient;
  socklen_t adrClientLen = sizeof(struct sockaddr_in); // nécessaire
  getpeername(sService,(struct sockaddr*)&adrClient,&adrClientLen);
  getnameinfo((struct
sockaddr*)&adrClient,adrClientLen,host,NI_MAXHOST,port,NI_MAXSERV,NI_NUMERICSERV |
NI_NUMERICHOST);
  printf("Client connecte --> Adresse IP: %s -- Port: %s\n",host,port);

  // Lecture sur la socket
  int nb;
  char buffer1[50];
  if ((nb = read(sService,buffer1,5)) == -1)
  {
    perror("Erreur de read()");
    close(sEcoute);
    close(sService);
  }
  buffer1[nb] = 0;
  printf("nbLus = %d Lu: --%s--\n",nb,buffer1);

  // Seconde lecture sur la socket
  char buffer2[50];
  if ((nb = read(sService,buffer2,5)) == -1)
  {
    perror("Erreur de read()");
    close(sEcoute);
    close(sService);
  }
  buffer2[nb] = 0;
  printf("nbLus = %d Lu: --%s--\n",nb,buffer2);

  // Ecriture sur la socket
  char reponse[50];
  sprintf(reponse,"[SERVEUR] %s%s",buffer1,buffer2);

  if ((nb = write(sService,reponse,strlen(reponse))) == -1)
  {
    perror("Erreur de write()");
    close(sEcoute);
    close(sService);
  }
  printf("nbEcrits = %d Ecrit: --%s--\n",nb,reponse);

  // Fermeture de la connexion cote serveur
  close(sService);
  close(sEcoute);

  exit(0);
}
```

On observe que :

- La **lecture** se fait en 2 fois (pour l'exemple !) : une première demande de lecture de 5 bytes suivi d'une seconde lecture de 5 bytes

- La réponse, donc l'**écriture**, se réalise en une seule fois et le nombre de bytes écrits correspond à l'entièreté de la requête du client concaténée avec « [SERVEUR] »
- Une fois lancé le processus **serveur** sera bloqué sur le premier appel système **read()**
- Après avoir répondu, le **serveur** ferme bien ses 2 sockets

Le code du **client** (fichier **Client.cpp**) est :

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

int main(int argc,char* argv[])
{
  if (argc != 3)
  {
    printf("Erreur, usage :\n");
    printf("Client ipServeur portServeur\n");
    exit(1);
  }

  int sClient;
  printf("pid = %d\n",getpid());

  // Creation de la socket
  if ((sClient = socket(AF_INET, SOCK_STREAM, 0)) == -1)
  {
    perror("Erreur de socket()");
    exit(1);
  }
  printf("socket creee = %d\n",sClient);

  // Construction de l'adresse du serveur
  struct addrinfo hints;
  struct addrinfo *results;
  memset(&hints,0,sizeof(struct addrinfo));
  hints.ai_family = AF_INET;
  hints.ai_socktype = SOCK_STREAM;
  hints.ai_flags = AI_NUMERICSERV;
  if (getaddrinfo(argv[1],argv[2],&hints,&results) != 0)
    exit(1);

  // Demande de connexion
  if (connect(sClient,results->ai_addr,results->ai_addrlen) == -1)
  {
      perror("Erreur de connect()");
      exit(1);
```

```
  }
  printf("connect() reussi !\n");

  // Ecriture sur la socket
  int nb;
  if ((nb = write(sClient,"abcdefgh",8)) == -1)
  {
    perror("Erreur de write()");
    close(sClient);
  }
  printf("nbEcrits = %d\n",nb);

  // Lecture sur la socket
  char buffer[50];
  if ((nb = read(sClient,buffer,50)) == -1)
  {
    perror("Erreur de read()");
    close(sClient);
  }
  buffer[nb] = 0;
  printf("nbLus = %d Lu: --%s--\n",nb,buffer);

  // Fermeture de la connexion cote serveur
  close(sClient);

  exit(0);
}
```

On observe que :

- Une fois connecté, le client envoie une seule requête de **8 bytes** correspondant à « **abcdefgh** »
- La lecture de la réponse se réalise en une seule fois

Le processus **serveur** est lancé sur la machine **zeus** (IP = **192.168.228.169**) et dans la console du **serveur** :

```
[zeus]# Serveur
pid = 16681
socket creee = 3
Mon Adresse IP: 0.0.0.0 -- Mon Port: 50000
bind() reussi !
listen() reussi !
accept() reussi !
socket de service = 4
Client connecte --> Adresse IP: 192.168.228.167 -- Port: 47084
nbLus = 5 Lu: --abcde--
nbLus = 3 Lu: --fgh--
nbEcrits = 18 Ecrit: --[SERVEUR] abcdefgh--
[zeus]# Serveur
pid = 16716
socket creee = 3
```

```
Mon Adresse IP: 0.0.0.0 -- Mon Port: 50000
Erreur de bind(): Address already in use
[zeus]# netstat -an | grep 50000
tcp        0      0 192.168.228.169:50000   192.168.228.167:47084   TIME_WAIT
[zeus]#
```

Le processus **client** est lancé sur la machine **moon** (IP = **192.168.228.167**) et sur la console du client :

```
[moon]# Client 192.168.228.169 50000
pid = 112670
socket creee = 3
connect() reussi !
nbEcrits = 8
nbLus = 18 Lu: --[SERVEUR] abcdefgh--
[moon]#
```

On observe que

- Le **client** a envoyé **8 bytes** (le retour de l'appel système **write()** en témoigne) et se met en attente sur l'appel système **read()**
- Au niveau **serveur**,
  - Le premier appel à **read()** demandait de lire **5 bytes**, **8 bytes** étant disponibles à la lecture, l'appel système **read()** a retourné **5**
  - Le second appel système n'était pas bloquant : **read()** demandait à lire **5 bytes**, **3** étaient disponibles, l'appel système a retourné **3**
  - La réponse a été construite en concaténant « **[SERVEUR]** » et « **abcdefgh** » avec un espace entre les 2, donc un total de **18 bytes** qui correspond bien à la valeur de retour de **write()**
- L'appel de **read()** du **client** se débloque et retourne bien la valeur **18**

Remarques importantes :

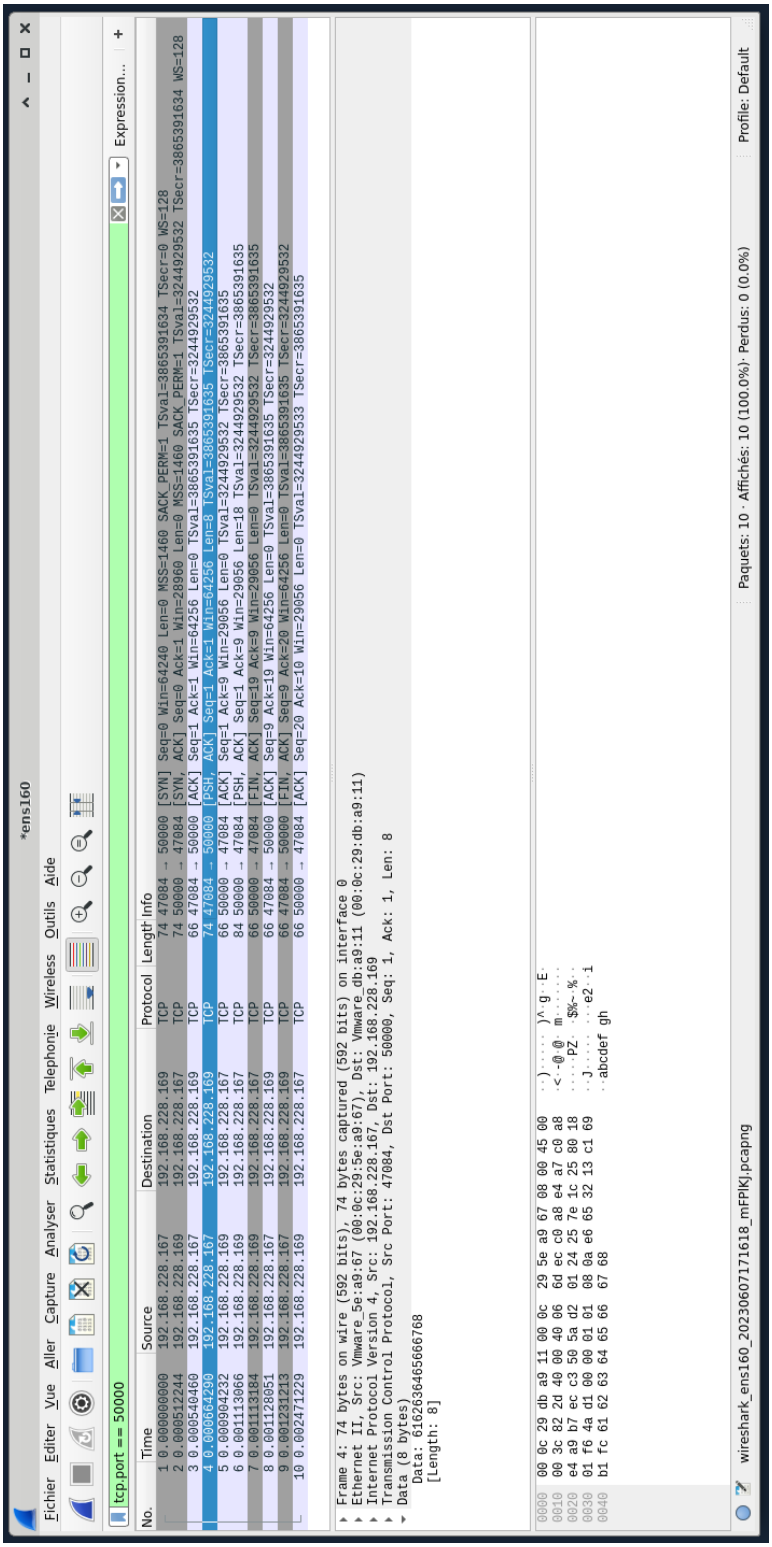
- Comme pour les pipes, la **lecture est destructrice** et l'appel système **lseek** est impossible
- Ci-dessus, on remarque une tentative de relancer le serveur directement après l'arrêt de celui-ci. Ceci n'a pas été possible. En effet, le **port** était toujours « **occupé** » par le **protocole TCP**. Ceci est visible par l'appel de la commande **netstat** qui indique que la socket est dans l'état **TIME_WAIT**

Observation de la communication dans Wireshark

Wireshark est un analyseur de paquets libre et gratuit. Il permet de visualiser le contenu des trames qui passent sur le réseau.

Remarque : pour avoir accès au trafic réseau, il doit être lancé en sudo (sous Linux)

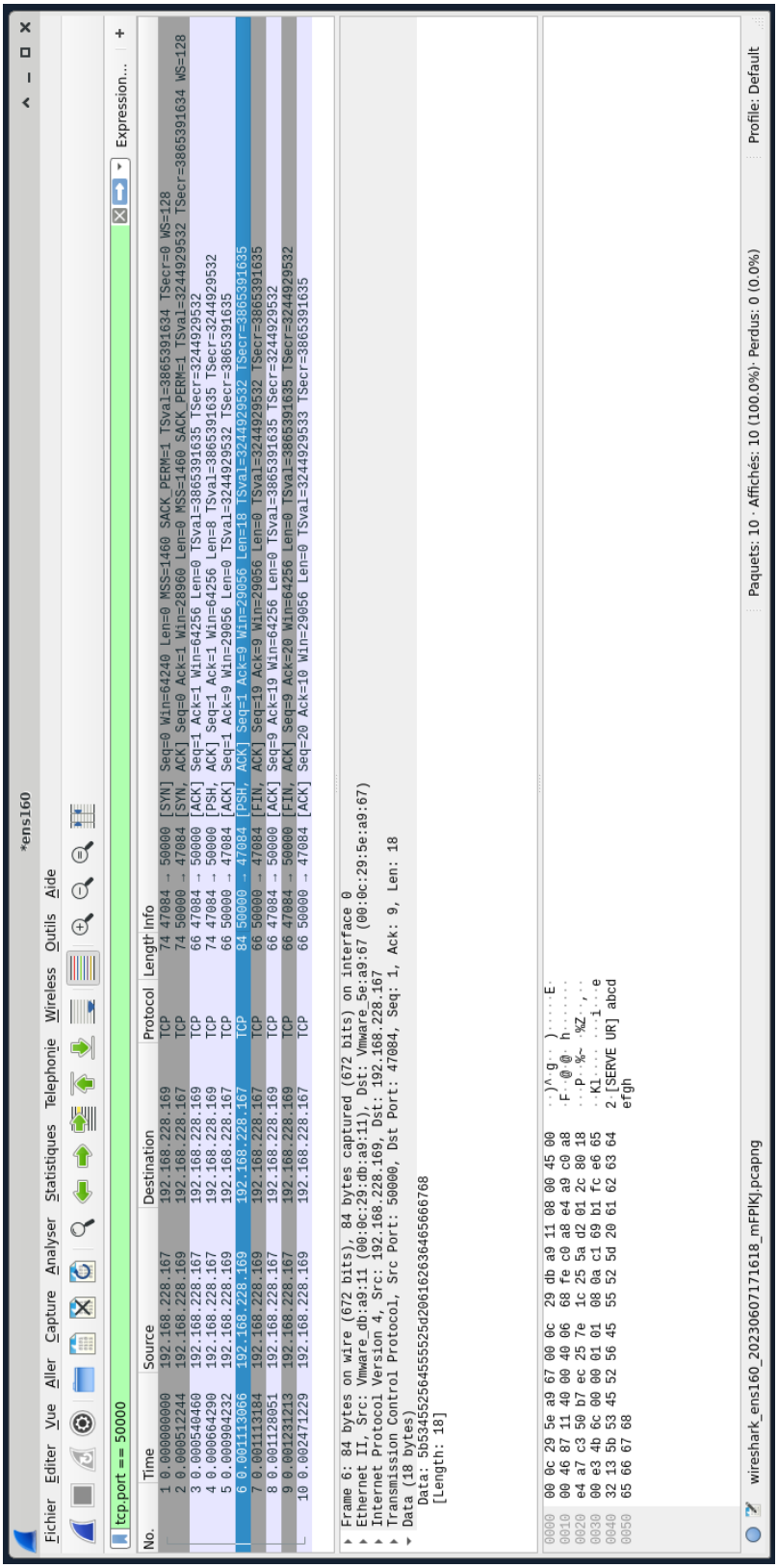
Capture 1 : requête envoyée par le client



On observe :

- Lignes 1 à 3 : établissement de la connexion → **three-way handshake**
- Lignes 3 et 4 : envoi de la requête par le client de l'**ACK** par le serveur

Capture 2 : **réponse envoyée par le serveur**



On observe :

- Lignes 6 et 7 : envoi de la réponse par le serveur et de l'**ACK** par le client
- Lignes 7 à 10 : les 4 paquets propre à la terminaison de la connexion

Accéder aux options des sockets

Dans l'exemple précédent, nous avons observé qu'il est impossible de relancer directement le processus **serveur**. En effet, la socket associée au port 50000 est dans l'état **TIME_WAIT** du protocole à états TCP géré par le système d'exploitation. Il faut alors attendre que le port soit libéré par le système (de l'ordre de la minute).

Il est également possible d'agir par programmation, en accédant aux **options des sockets**. Pour cela, on utilise les appels système

```
#include <sys/types.h>
#include <sys/socket.h>

int getsockopt(int sockfd,int level,int optname,void *optval,socklen_t *optlen);
int setsockopt(int sockfd,int level,int optname,const void *optval,socklen_t optlen);
```

où

- **sockfd** est la socket que l'on désire manipuler
- **level** correspond au niveau auquel s'applique l'option → il représente la couche de protocole correspondant à l'option désirée. Il peut prendre les valeurs
  - **SOL_SOCKET** : s'il s'agit de la socket elle-même
  - **IPPROTO_IP** : correspond à la couche réseau IP
  - **IPPROTO_TCP** : correspond à la couche réseau TCP
- **optname** représente l'option elle-même
- **optval** est l'adresse d'une variable contenant la valeur de l'option elle-même → **getsockopt()** remplira cette variable et **setsockopt()** lira son contenu
- **optlen** correspond à la longueur de la variable pointée par **optval**. La plupart des options sont de type int, il suffit alors d'utiliser sizeof(int)

Dans le cas du niveau **SOL_SOCKET** qui nous intéresse essentiellement ici, il existe de nombreuses valeurs possibles pour **optname** mais les plus fréquemment utilisées pour le programmeur applicatif sont

- **SO_REUSEADDR** : permet de réutiliser directement une socket (et donc un port) déjà affecté
- **SO_BROADCAST** : permet la diffusion de messages en broadcast sur une socket UDP
- …

L'option **SO_REUSEADDR** permet notamment de relancer immédiatement un **serveur** TCP qu'on vient d'arrêter et dont la socket se trouve par exemple dans l'état **TIME_WAIT**. Dans ce cas, on insère l'appel à **setsockopt()** **avant** le **bind()** :

```
…

int main()
{
  // Creation de la socket
  …

  // Construction de l'adresse
  …

  // Liaison de la socket à l'adresse
  int value = 1 ;
  setsockopt(sEcoute,SOL_SOCKET,SO_REUSEADDR,&value,sizeof(int));
  if (bind(sEcoute,results->ai_addr,results->ai_addrlen) < 0)
  {
      perror("Erreur de bind()");
      exit(1);
  }

  …

}
```

L'option **SO_BROADCAST** permet d'effectuer de la diffusion globale, ce qui consiste à envoyer un message **UDP** en direction de tout un sous-réseau. L'ensemble des machines ayant une adresse IP dans ce sous-réseau recevra le paquet de données. Ce mécanisme permet d'arroser un ensemble de machines avec des données.

Adopter une stratégie d'échange de données

Pour rappel, les caractéristiques principales d'une communication **TCP** sont :

- Elle travaille en mode « **connecté** » : un client a dû se connecter sur un serveur avant de pouvoir communiquer avec lui
- Elle est **fiable** : les données sont envoyées et arrivent dans un ordre précis, et ne sont jamais perdues
- Elle est orientée « **flot de données** », comme l'est une communication par pipe

C'est cette dernière caractéristique qui nous préoccupe ici.

Quand on pense à une communication « **client-serveur** », on pense immédiatement à la notion de « **requête** » et de « **réponse** ». Cependant,

- L'envoi d'une donnée par un appel de **write()** peut être lu à l'aide de plusieurs appels de de **read()** (cf. exemple précédent)
- Plusieurs envois de données (par plusieurs appels de **write()**) peuvent être lus par un seul appel de **read()**

Donc peut-on considérer que

- un appel de **write()** correspond à l'envoi d'une « requête » ?
- un appel de **read()** correspond à la réception d'une « requête » ?

La réponse est **non**… Et la cause est justement cet aspect « **flot** » d'une communication typique des communications par pipe.

Différentes stratégies

Supposons qu'un émetteur (client ou serveur) doive envoyer des données (des chaînes de caractères, plusieurs données différentes, des structures de données, etc..) à un récepteur (serveur ou client). Plusieurs stratégies peuvent être envisagées :

- les 2 intervenants décident d'un commun accord de s'envoyer des **« paquets » de taille fixe**. Ainsi les paramètres des appels de **write()** et de **read()** sont identiques et il est ainsi aisé pour les 2 intervenants de se synchroniser.

Inconvénient majeur : Imaginons que les 2 intervenants doivent s'échanger des chaînes de caractères de taille variable mais dont la taille maximale ne dépassera pas 1000 caractères. Les deux intervenants vont s'échanger des paquets de données dont la charge utile est de 1000 bytes systématiquement, sans se soucier du nombre de bytes effectivement utiles pour eux. Donc il faudra véhiculer 1000 bytes sur le réseau pour transmettre un simple « Hello World », ce qui semble réellement exagéré… De plus, si les chaînes de caractères dépassent la taille du MTU (1500), il faudra alors avoir une attention particulière car il faudra alors sans doute faire appel à plusieurs **read()** avant d'obtenir les données complètes.

- Si **l'émetteur ferme la connexion** (appel de **close()** ou **shutdown()**) une fois que les données sont transmises (cas du protocole **HTTP** d'un serveur date-heure classique), il suffit d'attendre que l'appel de **read()** retourne 0

  Inconvénient majeur : cette stratégie nécessite de se reconnecter à chaque fois que l'on souhaite envoyer une nouvelle donnée

Ces 2 stratégies ne semblent donc pas adaptées pour la construction d'un système **« client/serveur » optimisé et orienté « connexion »**.

On peut alors envisager d'ajouter une **« marque de fin de message »** qui sera fournie au destinataire. Cependant, **TCP** ne fournit pas lui-même un marqueur de fin → normal vu qu'il est orienté flot de données → c'est donc au programmeur à y pourvoir !

On peut alors considérer les 2 stratégies suivantes :

- On peut ajouter aux données utiles une **entête précisant le nombre de bytes envoyés**

  Exemple : supposons que l'on désire envoyer la chaîne de caractères **« Hello World »**, on peut construire la charge utile **« 0011Hello World »** qui sera ainsi envoyée (envoi de 15 bytes en tout). L'entête est composé de 4 caractères '0', '0', '1', '1' précisant le nombre de bytes envoyés, ici en l'occurrence 11 (remarquez que le **'\0'** de fin de chaîne n'a pas été transmis → ce n'est pas utile si le récepteur sait qu'il va recevoir une chaîne de caractères). Le récepteur commence par lire 4 bytes (dans tous les cas), puis il sait alors qu'il doit encore lire 11 bytes pour obtenir l'entièreté de la requête

- On peut ajouter un (ou des) **marqueur(s) de fin** (**FTP** ajoute **\r\r\n** par exemple)

  Exemple : supposons que l'on désire envoyer la chaîne de caractère **« Hello World »**, on peut construire la charge utile **« Hello World#) »** qui sera ainsi envoyée (envoi de 13 bytes en tout). Le marqueur de fin ajouté ici (ce n'est qu'un exemple !) est **#)**. Le récepteur peut alors lire sur la socket byte par byte jusqu'au moment où il détecte la séquence **#)** → il sait alors qu'il a reçu l'entièreté de la requête

Construction d'une « librairie de sockets TCP »

On constate qu'il devient fastidieux, au niveau du code (complexité et longueur surtout) d'établir une connexion, attendre une connexion, envoyer et recevoir des données de taille variable, …

Il pourrait alors être intéressant de construire une **librairie de sockets** permettant d'encapsuler tous les appels systèmes nécessaires dans des fonctions simples d'utilisation, tout en s'abstrayant de structure système telle que sockaddr_in.

Voici un exemple de librairie (en C mais cela pourrait être C++) simple que l'on pourrait imaginer de construire (fichier **TCP.h**) :

```
#ifndef TCP_H
#define TCP_H

#define TAILLE_MAX_DATA 10000

int ServerSocket(int port);
int Accept(int sEcoute,char *ipClient);
int ClientSocket(char* ipServeur,int portServeur);
int Send(int sSocket,char* data,int taille);
int Receive(int sSocket,char* data);

#endif
```

où

- **ServerSocket()** est une fonction qui sera appelée par le processus **serveur** (celui qui se mettra donc en attente d'une connexion). Elle prend en entrée le port sur

lequel le processus souhaite attendre et retourne la socket d'écoute ainsi créé. Pour ce faire, cette fonction

- fait un appel à **socket()** pour créer la socket
- construit l'adresse réseau de la socket par appel à **getaddrinfo()**
- fait appel à **bind()** pour lier la socket à l'adresse réseau

- **Accept()** est une fonction qui sera appelée par le processus **serveur**. Elle prend en premier paramètre la socket créée par l'appel de **ServerSocket()** et retourne la socket de service obtenue par connexion avec un **client**. Pour ce faire, cette fonction
  - fait appel à **listen()**
  - fait appel à **accept()**
  - récupère éventuellement l'adresse IP distante du client qui vient de se connecter. Cette adresse IP est placée dans **ipClient** si celui-ci est non NULL. S'il est non NULL, **ipClient** doit pointer vers une zone mémoire capable de recevoir une chaîne de caractères de la taille d'une adresse IP (Exemple : « 192.168.228.167 »)

- **ClientSocket()** est une fonction qui sera appelée par le processus **client** (celui qui souhaite se connecter sur un serveur). Elle prend en entrée l'adresse IP (sous forme d'une chaîne de caractère du type « 192.168.228.169 ») et le port (sous forme d'un int) du **serveur** sur lequel on désire se connecter. Elle retourne la socket de service qui lui va lui permettre de communiquer avec le **serveur**. Pour ce faire, cette fonction
  - fait appel à **socket()** pour créer la socket
  - construit l'adresse réseau de la socket (avec l'IP et le port du serveur) par appel à la fonction **getaddrinfo()**
  - fait appel à **connect()** pour se connecter sur le serveur

- **Send()** est une fonction qui sera utilisée par le processus **client** et le processus **serveur** afin d'envoyer des données. Cette fonction reçoit en paramètre
  - la socket de service
  - l'adresse mémoire d'un « **paquet de bytes** » que l'on désire envoyer
  - la taille de ce paquet de bytes

  et retourne le nombre de bytes qui ont été envoyés à des fins de « test d'erreur ».

- **Receive()** est une fonction qui sera utilisée par le processus **client** et le processus **serveur** afin de recevoir un « paquet de données » envoyé par la fonction **Send()**. Elle reçoit en paramètre
  - la socket de service
  - l'adresse d'un buffer de réception qui va revoir les données lues sur le réseau

  et retourne le nombre de bytes qui ont été lus

Il est à remarquer que les fonctions **Send()** et **Receive()** vont de paire et tiennent compte d'une des **stratégies d'échanges de données** citées plus haut.

Voici un exemple de fonction **Send()** utilisant la stratégie qui consiste à ajouter un marqueur de fin :

```
int Send(int sSocket,char* data,int taille)
{
  if (taille > TAILLE_MAX_DATA)
    return -1;

  // Preparation de la charge utile
  char trame[TAILLE_MAX_DATA+2];
  memcpy(trame,data,taille);
  trame[taille] = '#';
  trame[taille+1] = ')';

  // Ecriture sur la socket
  return write(sSocket,trame,taille+2)-2;
}
```

On observe que

- La macro **TAILLE_MAX_DATA** représente la taille maximum d'un « paquet de bytes » que l'on souhaite envoyer → cela ne veut pas dire que TAILLE_MAX_DATA sont envoyés sur le réseau à chaque requête/réponse
- La variable **trame** contient le « paquet de bytes » que l'on désire envoyer « concaténé » avec le marqueur de fin **#)**
- L'appel de **write()** reçoit bien en paramètre un nombre de bytes correspondant à la taille du « paquet de bytes » que l'on désire envoyé augmenté de la taille du marqueur qui est de 2 ici

La fonction **Receive()** qui correspond à cette fonction **Send()** pourrait alors être :

```
int Receive(int sSocket,char* data)
{
  bool fini = false;
  int nbLus, i = 0;
  char lu1,lu2;

  while(!fini)
  {
    if ((nbLus = read(sSocket,&lu1,1)) == -1)
      return -1;

    if (nbLus == 0) return i;     // connexion fermee par client

    if (lu1 == '#')
    {
      if ((nbLus = read(sSocket,&lu2,1)) == -1)
        return -1;

      if (nbLus == 0) return i;  // connexion fermee par client

      if (lu2 == ')') fini = true;
      else
      {
        data[i] = lu1;
        data[i+1] = lu2;
        i += 2;
      }
    }
    else
    {
      data[i] = lu1;
      i++;
    }
  }

  return i;
}
```

On observe que

- Les bytes sont lus un par un par l'appel système **read()**
- Une fois que les caractères '**#**' puis '**)**' sont lus consécutivement, la boucle de lecture se termine.
- La variable **i** contient à la fin le nombre bytes « utiles » qui ont été lus

- Une fin de connexion anticipée du client est détectée par le test de la valeur de retour de **read()** à 0
- Notez que **data** doit pointer vers un buffer de réception dont la taille est suffisante

La conception des autres fonctions de la librairie sont laissées au bon soin du lecteur 😉

Exemple d'utilisation de librairie de sockets

Dans l'exemple qui suit, nous allons utiliser/tester la librairie de sockets.

Voici le code du programme **serveur** (fichier **ServeurTest.cpp**) :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include "TCP.h"

typedef struct
{
  char nom[20];
  int  age;
  float poids;
} PERSONNE;

int main(int argc,char* argv[])
{
  if (argc != 2)
  {
     printf("Erreur...\n");
     printf("USAGE : ServeurTest portServeur\n");
     exit(1);
  }

  int sServer;

  if ((sServer = ServerSocket(atoi(argv[1]))) == -1)
  {
     perror("Erreur de ServeurSocket");
     exit(1);
  }

  printf("Attente d'une connexion...\n");
  int sService;
  if ((sService = Accept(sServer,NULL)) == -1)
  {
     perror("Erreur de Accept");
```

```
    close(sServer);
    exit(1);
}

printf("Connexion acceptee !\n");

// ***** Reception texte pur ************************************
char buffer[100];
int nbLus;

if ((nbLus = Receive(sService,buffer)) < 0)
{
    perror("Erreur de Receive");
    close(sService);
    close(sServer);
    exit(1);
}

printf("NbLus = %d\n",nbLus);
buffer[nbLus] = 0;
printf("Lu    = --%s--\n",buffer);

// ***** Envoi de texte pur *************************************
char texte[80];
sprintf(texte,"Je vais bien merci ;) !");
int nbEcrits;
if ((nbEcrits = Send(sService,texte,strlen(texte))) < 0)
{
  perror("Erreur de Send");
  close(sService);
  close(sServer);
  exit(1);
}

printf("NbEcrits = %d\n",nbEcrits);
printf("Ecrit    = --%s--\n",texte);

// ***** Reception d'une structure ********************************
PERSONNE p;

if ((nbLus = Receive(sService,(char*)&p)) < 0)
{
    perror("Erreur de Receive");
    close(sService);
    close(sServer);
    exit(1);
}

printf("NbLus = %d\n",nbLus);
printf("Lu    = --%s--%d--%f--\n",p.nom,p.age,p.poids);
```

```
  // ***** Envoi d'une structure ************************************
  strcpy(p.nom,"charlet");
  p.age = 54;
  p.poids = 71.98f;
  if ((nbEcrits = Send(sService,(char*)&p,sizeof(PERSONNE))) < 0)
  {
    perror("Erreur de Send");
    close(sService);
    close(sServer);
    exit(1);
  }

  printf("NbEcrits = %d\n",nbEcrits);
  printf("Ecrit    = --%s--%d--%f--\n",p.nom,p.age,p.poids);

  close(sService);
  close(sServer);

  exit(0);
}
```

Et voici le code du programme **client** (fichier **ClientTest.cpp**) :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include "TCP.h"

typedef struct
{
  char nom[20];
  int  age;
  float poids;
} PERSONNE;

int main(int argc,char* argv[])
{
  if (argc != 3)
  {
     printf("Erreur...\n");
     printf("USAGE : ClientTest ipServeur portServeur\n");
     exit(1);
  }

  int sClient;

  if ((sClient = ClientSocket(argv[1],atoi(argv[2]))) == -1)
```

```
{
    perror("Erreur de ClientSocket");
    exit(1);
}

// ***** Envoi de texte pur ************************************
char texte[80];
sprintf(texte,"Bonjour, comment vas-tu ?");
int nbEcrits;
if ((nbEcrits = Send(sClient,texte,strlen(texte))) == -1)
{
  perror("Erreur de Send");
  close(sClient);
  exit(1);
}

printf("NbEcrits = %d\n",nbEcrits);
printf("Ecrit    = --%s--\n",texte);

// ***** Reception texte pur ***********************************
char buffer[100];
int nbLus;

if ((nbLus = Receive(sClient,buffer)) < 0)
{
  perror("Erreur de Receive");
  close(sClient);
  exit(1);
}

printf("NbLus = %d\n",nbLus);
buffer[nbLus] = 0;
printf("Lu    = --%s--\n",buffer);

// ***** Envoi d'une structure *************************************
PERSONNE p;
strcpy(p.nom,"Wagner");
p.age = 49;
p.poids = 87.21f;
if ((nbEcrits = Send(sClient,(char*)&p,sizeof(PERSONNE))) < 0)
{
  perror("Erreur de Send");
  close(sClient);
  exit(1);
}

printf("NbEcrits = %d\n",nbEcrits);
printf("Ecrit    = --%s--%d--%f--\n",p.nom,p.age,p.poids);

// ***** Reception d'une structure *********************************
```

```
  if ((nbLus = Receive(sClient,(char*)&p)) < 0)
  {
    perror("Erreur de Receive");
    close(sClient);
    exit(1);
  }

  printf("NbLus = %d\n",nbLus);
  printf("Lu    = --%s--%d--%f--\n",p.nom,p.age,p.poids);

  close(sClient);

  exit(0);
}
```

Pour cet exemple,

- Le processus **serveur** est lancé sur la machine **zeus** (IP = **192.168.228.169**) et mis en attente sur le port **50000**
- Le processus **client** est lancé sur la machine **moon** (IP = **192.168.228.167**)

Dans la console du serveur, nous observons

```
[zeus]$ ServeurTest
Erreur...
USAGE : ServeurTest portServeur
[zeus]$ ServeurTest 50000
Attente d'une connexion...
Connexion acceptee !
NbLus = 25
Lu    = --Bonjour, comment vas-tu ?--
NbEcrits = 23
Ecrit    = --Je vais bien merci ;) !--
NbLus = 28
Lu    = --Wagner--49--87.209999--
NbEcrits = 28
Ecrit    = --charlet--54--71.980003--
[zeus]$
```

Tandis que dans la console du client, nous observons

```
[moon]$ ClientTest 192.168.228.169 50000
NbEcrits = 25
Ecrit    = --Bonjour, comment vas-tu ?--
NbLus = 23
Lu    = --Je vais bien merci ;) !--
NbEcrits = 28
Ecrit    = --Wagner--49--87.209999--
NbLus = 28
```

```
Lu    = --charlet--54--71.980003--
[moon]$
```

On observe que

- L'utilisation de **Send()** et **Receive()** est symétrique : **serveur** et **client** l'utilisent de la même manière
- Les fonctions **Send()** et **Receive()** peuvent s'utiliser pour envoyer des chaînes de caractères mais également des structures → en fait n'importe quel « paquets de bytes »
- Dans le cas des chaînes de caractères, il est nécessaire de concaténer **'\0'** au buffer de réception car celui-ci n'est pas transmis par réseau

Un exemple de serveur mono-processus (ou mono-thread)

Nous allons à présent utiliser la librairie de socket développée ci-dessus pour construire

- un **serveur mono-processus** :
  - Étant mono-processus, lorsqu'un client sera connecté, aucun autre client ne pourra se connecter et dialoguer avec lui. En effet, le thread principal ne pourra pas accepter une nouvelle connexion tant qu'il n'aura pas terminé sa conversation avec le client
  - Il tournera en boucle et se contentera d'envoyer comme réponse la requête reçue du client concaténée avec la chaine de caractères « **[SERVEUR]** »
  - Qui s'arrêtera proprement à la réception du **signal SIGINT**
- Un **client**
  - tournant en boucle dans laquelle il envoie une requête au serveur en attend la réponse
  - qui s'arrêtera proprement à la réception du **signal SIGINT**

Voici le code du **serveur** (fichier **Serveur.cpp**) :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>
#include "TCP.h"
```

```c
int sEcoute;
int sService;

void HandlerSIGINT(int s);
void TraitementConnexion();

int main(int argc,char* argv[])
{
  if (argc != 2)
  {
     printf("Erreur...\n");
     printf("USAGE : Serveur portServeur\n");
     exit(1);
  }

  // Armement des signaux
  struct sigaction A;
  A.sa_flags = 0;
  sigemptyset(&A.sa_mask);
  A.sa_handler = HandlerSIGINT;
  if (sigaction(SIGINT,&A,NULL) == -1)
  {
    perror("Erreur de sigaction");
    exit(1);
  }

  // Creation de la socket d'écoute
  if ((sEcoute = ServerSocket(atoi(argv[1]))) == -1)
  {
     perror("Erreur de ServeurSocket");
     exit(1);
  }

  // Mise en boucle du serveur
  printf("Demarrage du serveur.\n");
  while(1)
  {
    printf("Attente d'une connexion...\n");
    if ((sService = Accept(sEcoute,NULL)) == -1)
    {
       perror("Erreur de Accept");
       close(sEcoute);
       exit(1);
    }

    printf("Connexion acceptee !\n");

    // Traitement de la connexion
    TraitementConnexion();
```

```
  }
}

void HandlerSIGINT(int s)
{
  printf("\nArret du serveur.\n");
  close(sEcoute);
  close(sService);
  exit(0);
}

void TraitementConnexion()
{
  char requete[200], reponse[200];
  int nbLus, nbEcrits;

  while (1)
  {
    printf("\tAttente requete...\n");
    // ***** Reception Requete ******************
    if ((nbLus = Receive(sService,requete)) < 0)
    {
      perror("Erreur de Receive");
      HandlerSIGINT(0);
    }

    // ***** Fin de connexion ? *****************
    if (nbLus == 0)
    {
      printf("\tFin de connexion du client.\n");
      close(sService);
      return;
    }

    requete[nbLus] = 0;
    printf("\tRequete recue = %s\n",requete);

    // ***** Traitement de la requete ***********
    sprintf(reponse,"[SERVEUR] %s",requete);

    // ***** Envoi de la reponse ****************
    if ((nbEcrits = Send(sService,reponse,strlen(reponse))) < 0)
    {
      perror("Erreur de Send");
      HandlerSIGINT(0);
    }

    printf("\tReponse envoyee = %s\n",reponse);
  }
}
```

On remarque que :

- Une fois que le **serveur** a accepté une connexion (appel de **Accept()**), il exécute la fonction **TraitementConnexion()**. Tant que cette fonction n'est pas terminée, il ne peut pas remonter dans sa boucle principale et accepter une nouvelle connexion
- La fonction **TraitementConnexion()** traite une conversation complète avec le client connecté. Elle contient une boucle dans laquelle elle
  - reçoit la requête du client
  - prépare la réponse
  - envoie la réponse au client
- Pour terminer le traitement de la connexion, le serveur devra attendre que le client ait mis fin à la connexion. Cela se traduira par un retour égal à 0 de la fonction **Receive()**
- La réception du **signal SIGINT** par le serveur provoque la fermeture de la socket de service et de la socket d'écoute

Le code du **client** (fichier **Client.cpp**) est :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>
#include "TCP.h"

int sClient;

void HandlerSIGINT(int s);

int main(int argc,char* argv[])
{
  if (argc != 3)
  {
      printf("Erreur...\n");
      printf("USAGE : Client ipServeur portServeur\n");
      exit(1);
  }

  // Armement des signaux
  struct sigaction A;
  A.sa_flags = 0;
  sigemptyset(&A.sa_mask);
  A.sa_handler = HandlerSIGINT;
  if (sigaction(SIGINT,&A,NULL) == -1)
  {
```

```
    perror("Erreur de sigaction");
    exit(1);
  }

  // Connexion sur le serveur
  if ((sClient = ClientSocket(argv[1],atoi(argv[2]))) == -1)
  {
     perror("Erreur de ClientSocket");
     exit(1);
  }

  printf("Connecte sur le serveur.\n");

  char requete[200],reponse[200];
  int nbEcrits, nbLus;
  while(1)
  {
    printf("Requete a envoyer (<CTRL-C> four fin) : ");
    fgets(requete,200,stdin);
    requete[strlen(requete)-1] = 0;  // pour retirer le '\n'

    // ***** Envoi de la requete ******************
    if ((nbEcrits = Send(sClient,requete,strlen(requete))) == -1)
    {
      perror("Erreur de Send");
      HandlerSIGINT(0);
    }

    printf("Requete envoyee = %s\n",requete);

    // ***** Attente de la reponse ****************
    if ((nbLus = Receive(sClient,reponse)) < 0)
    {
      perror("Erreur de Receive");
      HandlerSIGINT(0);
    }

    if (nbLus == 0)
    {
      printf("Serveur arrete, pas de reponse reçue...");
      HandlerSIGINT(0);
    }
    reponse[nbLus] = 0;
    printf("Reponse recue = %s\n",reponse);
  }
}

void HandlerSIGINT(int s)
{
  printf("\nArret du client.\n");
```

```
  close(sClient);
  exit(0);
}
```

On observe que :

- Le **client** tourne en boucle jusqu'au moment où l'utilisateur mettra fin à la communication par l'envoi d'un **signal SIGINT** au processus, ce qui provoquera la fermeture de la socket du client
- La fin prématurée du serveur est détectée par le test de la valeur de retour de la fonction **Receive()** qui dans ce cas retournera 0

Pour cet exemple :

- Le **serveur** est lancé sur la machine **zeus** (IP = **192.168.228.169**) et mis en écoute sur le port **50000**
- Le **client** est lancé sur la machine **moon** (IP = **192.168.228.167**)

Sur la console du **serveur**, nous observons :

```
[zeus]$ ./Serveur 50000
Demarrage du serveur.
Attente d'une connexion...
Connexion acceptee !
      Attente requete...
      Requete recue = Hello there !
      Reponse envoyee = [SERVEUR] Hello there !
      Attente requete...
      Requete recue = Bye bye ;)
      Reponse envoyee = [SERVEUR] Bye bye ;)
      Attente requete...
      Fin de connexion du client.
Attente d'une connexion...
Connexion acceptee !
      Attente requete...
      Requete recue = Bonjour !
      Reponse envoyee = [SERVEUR] Bonjour !
      Attente requete...
^C
Arret du serveur.
[zeus]$
```

Tandis que sur la console du **client**, nous observons :

```
[moon]$ Client 192.168.228.169 50000
Connecte sur le serveur.
Requete a envoyer (<CTRL-C> four fin) : Hello there !
Requete envoyee = Hello there !
Reponse recue = [SERVEUR] Hello there !
Requete a envoyer (<CTRL-C> four fin) : Bye bye ;)
Requete envoyee = Bye bye ;)
Reponse recue = [SERVEUR] Bye bye ;)
Requete a envoyer (<CTRL-C> four fin) : ^C
Arret du client.
[moon]$ Client 192.168.228.169 50000
Connecte sur le serveur.
Requete a envoyer (<CTRL-C> four fin) : Bonjour !
Requete envoyee = Bonjour !
Reponse recue = [SERVEUR] Bonjour !
Requete a envoyer (<CTRL-C> four fin) : Au revoir...
Requete envoyee = Au revoir...
Serveur arrete, pas de reponse reçue...
Arret du client.
[moon]$
```

On observe que

- Une première connexion a eu lieu et dans laquelle c'est le **client** qui a mis fin à la communication par **<CTRL-C>** → ce qui a été détecté par le serveur → **« Fin de connexion du client. »**
- Dans la seconde connexion, c'est le **serveur** qui a été interrompu par un **<CTRL-C>** → ce qui a été détecté par le client → **« Serveur arrete, pas de reponse reçue… »**

Dans cet exemple, on remarque essentiellement 3 choses

1. **Impossible de connecter 2 clients simultanément**. Normal, c'est un serveur mono-thread (ou mono-processus) → pour palier à cet inconvénient, il va falloir passer un modèle multi-threads (ou multi-processus) de serveur
2. Le **protocole est rudimentaire**, le serveur se contente de recopier la requête en guise de réponse
3. Nous avons réalisé ici un **serveur de « connexions »** ce qui signifie qu'une fois le client accepté, le serveur entre dans une boucle pour n'en sortir que lorsque la connexion (ou communication) a été traitée entièrement → Dans le cas (pas dans cet exemple) où le serveur accepte une connexion, lit une requête, envoie une

réponse puis se remet en attente directement sur **Accept()**, on parle de **serveur de « requêtes »** (car il ne traite qu'une seule requête par connexion).

Construire son propre protocole de communication

Jusqu'ici, notre **serveur** n'est capable que de réaliser « un écho » de ce qui a été envoyé par le **client**. Cependant, un serveur digne de ce nom doit être capable de répondre à plusieurs requêtes (« demandes ») différentes d'un client : login, achat d'un article, récupération d'un solde, paiement, logout… **Serveur** et **client** doivent donc se mettre d'accord sur un mécanisme d'échange de données : le **protocole de communication**

On distingue deux types de protocole :

- **Protocole sans état** : les requêtes peuvent arriver dans n'importe ordre, le serveur n'a aucune mémoire de ce qui s'est passé avant (cas de HTTP → voir plus tard)
- **Protocole avec états** : les requêtes ne peuvent pas arriver dans n'importe quel ordre et le **serveur** possède un « **état** », il se souvient des requêtes arrivées précédemment et peut agir en conséquence → exemple : sur un serveur d'achat de marchandises où il est nécessaire de se logger (avec un login et un mot de passe), il est impossible d'envoyer une requête d'achat tant que l'on ne s'est pas loggé. Une fois loggé (requête de login acceptée), le serveur mémorise l'identité du client qui vient de se connecter et peut en tenir compte dans les échanges suivants.

Comment construire les requêtes et les réponses

Des requêtes différentes (et du coup des réponses différentes) entrainent

- des données de types différents
- un nombre de paramètres différents par requête (réponse)
- la nécessité d'avoir un moyen de distinguer une requête d'une autre

On pourrait imaginer

- Utiliser les structures du C comportant autant de champs que de paramètres de requête (exemple d'un login : nom + mot de passe) → **inconvénient** : client et serveur doivent être écrits en C et compilés avec le même compilateur, ce qui est fort limitatif

- Utiliser une structure du C pouvant servir d'entête (contenant le type de requête et la taille des données) concaténé avec un paquet de bytes constituant les données → même inconvénient que le cas précédent

Aucune de ces deux propositions ne semble satisfaisante sauf si client et serveur sont écrits en C

Si on souhaite être plus général et permettre une communication entre machines différentes sans tenir compte du langage utilisé, il est nécessaire de ne pas utiliser les spécificités du langage dans la construction des requêtes/réponses. Il faut donc utiliser des mécanismes/données compréhensibles par tous les intervenants sans tenir compte du langage ou de la machine → on préférera utiliser des chaînes de caractères ayant un format particulier et répondant aux besoins → on parle de **trame de requête/réponse**

Idéalement, la **trame** d'une requête/réponse doit idéalement contenir

- Une **entête** composée d'un identifiant de la requête/réponse et éventuellement d'un numéro de version du protocole (si on souhaite pouvoir faire évoluer un protocole et gérer plusieurs versions du protocole)
- Un ensemble de données : les **paramètres de la requête**
- Les éléments de l'entête et les paramètres « doivent » (d'autres solutions pourraient s'envisager) être séparés par un « **séparateur** » qui peut être un caractère quelconque mais qui doit être choisi à l'avance par tous les intervenants de la communication

Exemple de protocole à états

Imaginons un **serveur de calculs** (simple) permettant de réaliser en ligne les opérations classiques **(+,-,*,/) sur des entiers**. Pour pouvoir utiliser ce serveur, il sera nécessaire au client de se logger à l'aide d'un couple « login / mot de passe ». Une fois terminé, le client devra se délogger.

La requête de login nécessite 2 paramètres : le login et le mot de passe. On peut donc imaginer une trame de requête de la forme

```
"LOGIN#wagner#abc123"
```

où on observe que

- La trame de la requête comporte 3 champs

- Le 1er champ correspond au nom de la requête → on pourrait aussi imaginer de remplacer ce nom par un numéro de requête afin de gagner en nombre de bytes transmis mais au détriment de la lisibilité de la requête par le commun des mortels
- Les 2ème et 3ème champs correspondent aux paramètres de la requête : le login « wagner » et le mot de passe « abc123 »
- Les champs de la requête sont séparés par le séparateur **#**

A la réception d'une telle requête, le serveur

1. devrait vérifier la validité du couple « login/mot de passe » (base de données ou autre)
2. construire et envoyer une réponse au client pour le notifier de son statut

On pourrait donc imaginer une réponse du serveur de la forme

```
"LOGIN#ok"
```

en cas de succès ou

```
"LOGIN#ko#Mauvais identifiants !"
```

en cas d'échec. Ici (cas d'une communication en mode connecté TCP), inutile de préciser le login car la réponse est envoyée directement au bon client qui vient d'envoyer sa requête de login.

En cas de succès, le client est « **loggé** » et **la communication, tant au niveau serveur que client, se trouve dans l'état « loggé »**. Dans cet état, le client pourra envoyer ses requêtes « mathématiques », ce qui ne serait pas possible dans le cas contraire.

Pour une requête de calcul, on pourrait imaginer la trame suivante

```
"OPER#*#5#12"
```

On observe que

- il s'agit bien d'une requête pour une opération mathématique
- l'opération souhaitée est la multiplication *****
- les 2 opérandes sont **5** et **12** → les paramètres de la requête sont quant eux **'*'**, **'5'** et '**12**'

Pour ce type de requête, le nombre de paramètres est 3. **Client** et **serveur** se sont mis d'accord là-dessus (**mise en place du protocole**). Si ce n'est pas le cas, la requête est mal formée et le **serveur** devrait répondre par un message d'erreur

Enfin, la réponse à ce type de requête pourrait avoir la forme

```
"OPER#ok#60"
```

où on voit apparaître le résultat de l'opération mathématique. Mais on pourrait également avoir ceci :

```
"OPER#ko#Division par zero !"
```

en cas d'erreur (requête mal formée, division par 0, …)

Le requête de logout peut se faire avec ou sans accusé de réception.

Notre protocole ainsi construit et que l'on pourrait intituler « **SMOP** » (pour « **S**imple **M**ath **O**perations **P**rotocol ») se résume à

Protocole SMOP	**Format de la requête**	**Format de la réponse**
Login	`"LOGIN#user#password"`	`"LOGIN#ok"` si succès `"LOGIN#ko#raison"` si échec
Opération	`"OPER#operation#oper1#oper2"`	`"OPER#ok#resultat"` si succès `"LOGIN#ko#raison"` si échec
Logout	`"LOGOUT"`	`"LOGOUT#ok"` ou rien

Construction d'un serveur multi-threads

Nous allons à présent voir comment notre **serveur** pourrait répondre et communiquer avec plusieurs clients simultanément. Pour cela, il va être nécessaire de « **threader** » notre application **serveur**. Ainsi,

- un thread va être chargé d'accepter de nouvelles connexions (appel de **Accept()**)
  → ce thread porte le nom de « **thread serveur** »
- plusieurs autres threads vont recevoir les sockets de service produites par le thread serveur et s'occuper de la communication avec les clients correspondants
  → ces threads portent le nom de « **threads clients** »
- Pendant que les **threads clients** sont en communication avec les clients connectés, le **thread serveur** peut se remettre en attente d'une nouvelle connexion

On est clairement en présence d'un modèle de thread **« Producteur / Consommateur »** :

- le **thread serveur** fait office de **« producteur de tâches »**
- les **threads clients** font office de **« consommateurs de tâches »**
- les **« tâches »** sont la gestion d'une communication avec un client connecté

Il faut également se rappeler que le modèle **Producteur / Consommateur** présente deux variantes :

1. **A la demande** : dès qu'une connexion sera accepté par le **thread serveur**, celui-ci va créer un **thread client** et lui refiler la socket de service obtenue → le nombre de thread clients pourra donc augmenter fortement en fonction du nombre de demandes de connexion mais aucun client ne sera mis en file d'attente
2. **En Pool** : le **thread serveur** crée à l'avance un certain nombre fixé de **threads clients** (le « pool de threads »). Dès qu'il accepte une connexion, il place la socket de service obtenue dans une file d'attente et réveille un des threads du pool. Si un un des threads clients est disponible, la connexion est prise en compte directement. Sinon le client devra attendre qu'un des threads clients se libère → aucune inflation du nombre de threads mais mise en file d'attente de clients si trop de demande de connexions

Exemple de serveur multi-threads « à la demande »

Dans l'exemple qui suit :

- Nous allons créer un serveur multi-threads **« à la demande »**
- Le protocole mis en place sera celui donné en exemple ci-dessus : **SMOP** → il s'agit d'un **protocole à états** → celui-ci sera géré dans les fichiers **SMOP.cpp** et **SMOP.h**
- Le serveur sera un **« serveur de connexions »**, c'est-à-dire que c'est le même thread qui va s'occuper du même client tout le long de la communication (du login au logout)
- Client et serveur vont utiliser la librairie de sockets décrites plus haut (sous la forme des fichiers **TCP.cpp** et **TCP.h**)

Voici le code du **serveur** (fichier **Serveur.cpp**) :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
```

```c
#include <signal.h>
#include <pthread.h>
#include "TCP.h"
#include "SMOP.h"

void HandlerSIGINT(int s);
void TraitementConnexion(int sService);
void* FctThreadClient(void* p);

int sEcoute;

int main(int argc,char* argv[])
{
  if (argc != 2)
  {
     printf("Erreur...\n");
     printf("USAGE : Serveur portServeur\n");
     exit(1);
  }

  // Armement des signaux
  struct sigaction A;
  A.sa_flags = 0;
  sigemptyset(&A.sa_mask);
  A.sa_handler = HandlerSIGINT;
  if (sigaction(SIGINT,&A,NULL) == -1)
  {
    perror("Erreur de sigaction");
    exit(1);
  }

  // Creation de la socket d'écoute
  if ((sEcoute = ServerSocket(atoi(argv[1]))) == -1)
  {
     perror("Erreur de ServeurSocket");
     exit(1);
  }

  // Mise en boucle du serveur
  int sService;
  pthread_t th;
  char ipClient[50];
  printf("Demarrage du serveur.\n");
  while(1)
  {
    printf("Attente d'une connexion...\n");
    if ((sService = Accept(sEcoute,ipClient)) == -1)
    {
      perror("Erreur de Accept");
      close(sEcoute);
```

```
      SMOP_Close();
      exit(1);
    }

    printf("Connexion acceptée : IP=%s socket=%d\n",ipClient,sService);

    // Creation d'un thread "client" s'occupant du client connecté
    int *p = (int*)malloc(sizeof(int));
    *p = sService;
    pthread_create(&th,NULL,FctThreadClient,(void*)p);
  }
}

void* FctThreadClient(void* p)
{
  int sService = *((int*)p);
  free(p);
  printf("\t[THREAD %p] Je m'occupe de la socket
%d\n",pthread_self(),sService);

  TraitementConnexion(sService);

  pthread_exit(NULL);
}

void HandlerSIGINT(int s)
{
  printf("\nArret du serveur.\n");
  close(sEcoute);
  SMOP_Close();
  exit(0);
}

void TraitementConnexion(int sService)
{
  char requete[200], reponse[200];
  int nbLus, nbEcrits;
  bool onContinue = true;

  while (onContinue)
  {
    printf("\t[THREAD %p] Attente requete...\n",pthread_self());
    // ***** Reception Requete ******************
    if ((nbLus = Receive(sService,requete)) < 0)
    {
      perror("Erreur de Receive");
      close(sService);
      HandlerSIGINT(0);
    }
```

```
    // ***** Fin de connexion ? *****************
    if (nbLus == 0)
    {
      printf("\t[THREAD %p] Fin de connexion du client.\n",pthread_self());
      close(sService);
      return;
    }

    requete[nbLus] = 0;
    printf("\t[THREAD %p] Requete recue = %s\n",pthread_self(),requete);

    // ***** Traitement de la requete ***********
    onContinue = SMOP(requete,reponse,sService);

    // ***** Envoi de la reponse ****************
    if ((nbEcrits = Send(sService,reponse,strlen(reponse))) < 0)
    {
      perror("Erreur de Send");
      close(sService);
      HandlerSIGINT(0);
    }

    printf("\t[THREAD %p] Reponse envoyee = %s\n",pthread_self(),reponse);

    if (!onContinue)
      printf("\t[THREAD %p] Fin de connexion de la socket
%d\n",pthread_self(),sService);
  }
}
```

On observe que

- Le **thread serveur** est en fait le thread principal → on pourrait imaginer de créer un thread spécifique à cette fonction → cela libèrerait le thread principal pour d'autres fonctions de gestion de l'application
- Dès qu'une connexion est acceptée, la socket de service obtenue est passé en paramètre à la fonction **pthread_create** pour la création du **thread client** qui va s'occuper de cette socket
- Le **threads clients** exécutent la fonction **TraitementConnexion()** qui contient une boucle dans laquelle le thread
  - lit une requête
  - traite la requête en appelant la fonction **SMOP()** responsable de la gestion du protocole
  - envoie la réponse au client avant de remonter dans sa boucle
- Le **thread client** ne sortira de sa boucle que lorsque le retour de la fonction **SMOP()** passera à false, indiquant que la communication doit se terminer

- Une fois qu'un **thread client** sort de sa fonction **TraitementConnexion()**, il se termine → caractéristique du modèle **« à la demande »**

Voyons à présent comment le protocole est géré. Voici le fichier **SMOP.h** :

```
#ifndef SMOP_H
#define SMOP_H

#define NB_MAX_CLIENTS 100

bool SMOP(char* requete, char* reponse,int socket);
bool SMOP_Login(const char* user,const char* password);
int  SMOP_Operation(char op,int a,int b);
void SMOP_Close();

#endif
```

Et le fichier **SMOP.cpp** :

```
#include "SMOP.h"
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

//***** Etat du protocole : liste des clients loggés ****************
int clients[NB_MAX_CLIENTS];
int nbClients = 0;

int  estPresent(int socket);
void ajoute(int socket);
void retire(int socket);

pthread_mutex_t mutexClients = PTHREAD_MUTEX_INITIALIZER;

//***** Parsing de la requete et creation de la reponse *************
bool SMOP(char* requete, char* reponse,int socket)
{
  // ***** Récupération nom de la requete *****************
  char *ptr = strtok(requete,"#");

  // ***** LOGIN ****************************************
  if (strcmp(ptr,"LOGIN") == 0)
  {
      char user[50], password[50];
      strcpy(user,strtok(NULL,"#"));
      strcpy(password,strtok(NULL,"#"));
      printf("\t[THREAD %p] LOGIN de %s\n",pthread_self(),user);
      if (estPresent(socket) >= 0)  // client déjà loggé
      {
```

```
        sprintf(reponse,"LOGIN#ko#Client déjà loggé !");
        return false;
      }
      else
      {
       if (SMOP_Login(user,password))
       {
         sprintf(reponse,"LOGIN#ok");
         ajoute(socket);
       }
       else
       {
         sprintf(reponse,"LOGIN#ko#Mauvais identifiants !");
         return false;
       }
    }
  }

  // ***** LOGOUT ****************************************
  if (strcmp(ptr,"LOGOUT") == 0)
  {
    printf("\t[THREAD %p] LOGOUT\n",pthread_self());
    retire(socket);
    sprintf(reponse,"LOGOUT#ok");
    return false;
  }

  // ***** OPER ******************************************
  if (strcmp(ptr,"OPER") == 0)
  {
    char op;
    int a,b;
    ptr = strtok(NULL,"#");
    op = ptr[0];
    a = atoi(strtok(NULL,"#"));
    b = atoi(strtok(NULL,"#"));
    printf("\t[THREAD %p] OPERATION %d %c %d\n",pthread_self(),a,op,b);
    if (estPresent(socket) == -1) sprintf(reponse,"OPER#ko#Client non loggé
!");
    else
    {
      try
      {
        int resultat = SMOP_Operation(op,a,b);
        sprintf(reponse,"OPER#ok#%d",resultat);
      }
      catch(int) { sprintf(reponse,"OPER#ko#Division par zéro !"); }
    }
  }
```

```
  return true;
}

//***** Traitement des requetes ***********************************
bool SMOP_Login(const char* user,const char* password)
{
  if (strcmp(user,"wagner")==0 && strcmp(password,"abc123")==0) return true;
  if (strcmp(user,"charlet")==0 && strcmp(password,"xyz456")==0) return true;
  return false;
}

int SMOP_Operation(char op,int a,int b)
{
  if (op == '+') return a+b;
  if (op == '-') return a-b;
  if (op == '*') return a*b;
  if (op == '/')
  {
    if (b == 0) throw 1;
    return a/b;
  }
  return 0;
}

//***** Gestion de l'état du protocole ****************************
int estPresent(int socket)
{
  int indice = -1;
  pthread_mutex_lock(&mutexClients);
  for(int i=0 ; i<nbClients ; i++)
     if (clients[i] == socket) { indice  = i; break; }
  pthread_mutex_unlock(&mutexClients);
  return indice;
}

void ajoute(int socket)
{
  pthread_mutex_lock(&mutexClients);
  clients[nbClients] = socket;
  nbClients++;
  pthread_mutex_unlock(&mutexClients);
}

void retire(int socket)
{
  int pos = estPresent(socket);
  if (pos == -1) return;
  pthread_mutex_lock(&mutexClients);
  for (int i=pos ; i<=nbClients-2 ; i++)
     clients[i] = clients[i+1];
```

```
  nbClients--;
  pthread_mutex_unlock(&mutexClients);
}

//***** Fin prématurée *********************************************
void SMOP_Close()
{
  pthread_mutex_lock(&mutexClients);
  for (int i=0 ; i<nbClients ; i++)
      close(clients[i]);
  pthread_mutex_unlock(&mutexClients);
}
```

On observe que :

- C'est la fonction **SMOP()** qui gère le protocole :
  - Elle parse la requête grâce à la fonction **strtok** (attention qu'il faudrait utiliser la version ré-entrante de la fonction ici !!! → **strtok_s()**)
  - Elle traite la requête en faisant appel aux autres fonctions
  - Elle crée la trame de la réponse
- Le vecteur **clients** représente la liste des sockets des clients loggés → il représente la « **mémoire du protocole** » et donc sont **état** → c'est ici qu'on voit qu'un client est loggé ou pas
- Le vecteur **clients**, ainsi **nbClients** (le nombre de clients loggés) sont globaux et manipulés par plusieurs threads simultanément → il sont donc protégés par un **mutex** → Les fonction **ajoute()**, **retire()**, **estPresent()** manipulent de manière atomique ces variables globales
- **SMOP_Login()** et **SMOP_Operation()** sont les fonctions qui représentent la logique « métier » du protocole
- La fonction **SMOP_Close()** permet de fermer toutes les sockets des clients connectés en cas de fin prématurée du serveur

Le fait de séparer le code du serveur et le code du protocole pourrait permettre d'aller plus loin et de construire un serveur générique et donc indépendant du protocole. Mais cela est laissé au bon soin du lecteur 😉…

Le code du **client** (fichier **Client.cpp**) est

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
```

```c
#include <signal.h>
#include "TCP.h"

int sClient;

void HandlerSIGINT(int s);

void Echange(char* requete, char* reponse);
bool SMOP_Login(const char* user,const char* password);
void SMOP_Logout();
void SMOP_Operation(char op,int a,int b);

int main(int argc,char* argv[])
{
  if (argc != 3)
  {
     printf("Erreur...\n");
     printf("USAGE : Client ipServeur portServeur\n");
     exit(1);
  }

  // Armement des signaux
  struct sigaction A;
  A.sa_flags = 0;
  sigemptyset(&A.sa_mask);
  A.sa_handler = HandlerSIGINT;
  if (sigaction(SIGINT,&A,NULL) == -1)
  {
    perror("Erreur de sigaction");
    exit(1);
  }

  // Connexion sur le serveur
  if ((sClient = ClientSocket(argv[1],atoi(argv[2]))) == -1)
  {
     perror("Erreur de ClientSocket");
     exit(1);
  }

  printf("Connecte sur le serveur.\n");

  // Phase de login
  char user[50],password[50];
  printf("user: "); fgets(user,50,stdin);
  user[strlen(user)-1] = 0;
  printf("password: "); fgets(password,50,stdin);
  password[strlen(password)-1] = 0;

  if (!SMOP_Login(user,password))
    exit(1);
```

```
  while(1)
  {
    int a,b;
    char op;

    printf("Operation (<CTRL-C> four fin) : ");
    fflush(stdin);
    scanf("%d %c %d",&a,&op,&b); // pas ouf !

    SMOP_Operation(op,a,b);
  }
}

//***** Fin de connexion ********************************************
void HandlerSIGINT(int s)
{
  printf("\nArret du client.\n");
  SMOP_Logout();
  close(sClient);
  exit(0);
}

//***** Gestion du protocole SMOP ***********************************
bool SMOP_Login(const char* user,const char* password)
{
  char requete[200],reponse[200];
  bool onContinue = true;

  // ***** Construction de la requete *********************
  sprintf(requete,"LOGIN#%s#%s",user,password);

  // ***** Envoi requete + réception réponse **************
  Echange(requete,reponse);

  // ***** Parsing de la réponse ***************************
  char *ptr = strtok(reponse,"#"); // entête = LOGIN (normalement...)
  ptr = strtok(NULL,"#"); // statut = ok ou ko
  if (strcmp(ptr,"ok") == 0) printf("Login OK.\n");
  else
  {
    ptr = strtok(NULL,"#"); // raison du ko
    printf("Erreur de login: %s\n",ptr);
    onContinue = false;
  }

  return onContinue;
}

//******************************************************************
```

```
void SMOP_Logout()
{
  char requete[200],reponse[200];
  int nbEcrits, nbLus;

  // ***** Construction de la requete *********************
  sprintf(requete,"LOGOUT");

  // ***** Envoi requete + réception réponse **************
  Echange(requete,reponse);

  // ***** Parsing de la réponse ***************************
  // pas vraiment utile...
}

//*****************************************************************
void SMOP_Operation(char op,int a,int b)
{
  char requete[200],reponse[200];

  // ***** Construction de la requete *********************
  sprintf(requete,"OPER#%c#%d#%d",op,a,b);

  // ***** Envoi requete + réception réponse **************
  Echange(requete,reponse);

  // ***** Parsing de la réponse ***************************
  char *ptr = strtok(reponse,"#"); // entête = OPER (normalement...)
  ptr = strtok(NULL,"#"); // statut = ok ou ko
  if (strcmp(ptr,"ok") == 0)
  {
    ptr = strtok(NULL,"#"); // résultat du calcul
    printf("Résultat = %s\n",ptr);
  }
  else
  {
    ptr = strtok(NULL,"#"); // raison du ko
    printf("Erreur: %s\n",ptr);
  }
}

//***** Echange de données entre client et serveur ******************
void Echange(char* requete, char* reponse)
{
  int nbEcrits, nbLus;

  // ***** Envoi de la requete ****************************
  if ((nbEcrits = Send(sClient,requete,strlen(requete))) == -1)
  {
    perror("Erreur de Send");
```

```
    close(sClient);
    exit(1);
  }

  // ***** Attente de la reponse *************************
  if ((nbLus = Receive(sClient,reponse)) < 0)
  {
    perror("Erreur de Receive");
    close(sClient);
    exit(1);
  }

  if (nbLus == 0)
  {
    printf("Serveur arrete, pas de reponse reçue...\n");
    close(sClient);
    exit(1);
  }
  reponse[nbLus] = 0;
}
```

On observe que

- Le protocole est géré par les fonctions **SMOP_Login()**, **SMOP_Operation()** et **SMOP_Logout()** qui pourraient être placée dans une librairie « SMOP côté client »
- Ces 3 fonctions utilisent la fonction **Echange()** qui permet simplement d'envoyer une requête quelconque au serveur et d'attendre la réponse

Pour cet exemple,

- Le **serveur** a été lancé sur la machine **zeus** (IP = **192.168.228.169**) et mis en attente sur le port **50000**
- Deux **clients** ont été lancés sur la machine **moon** (IP = **192.168.228.167**)
- Un troisième **client** a été lancé sur la machine **zeus** (en **localhost**)

Sur la console du **serveur**, nous observons

```
[zeus]$ ./Serveur 50000
Demarrage du serveur.
Attente d'une connexion...
Connexion acceptée : IP=192.168.228.167 socket=4
Attente d'une connexion...
      [THREAD 0x7fb9104d7700] Je m'occupe de la socket 4
      [THREAD 0x7fb9104d7700] Attente requete...
Connexion acceptée : IP=192.168.228.167 socket=5
Attente d'une connexion...
      [THREAD 0x7fb90fcd6700] Je m'occupe de la socket 5
      [THREAD 0x7fb90fcd6700] Attente requete...
```

```
Connexion acceptée : IP=127.0.0.1 socket=6
Attente d'une connexion...
      [THREAD 0x7fb90f4d5700] Je m'occupe de la socket 6
      [THREAD 0x7fb90f4d5700] Attente requete...
      [THREAD 0x7fb9104d7700] Requete recue = LOGIN#wagner#abc123
      [THREAD 0x7fb9104d7700] LOGIN de wagner
      [THREAD 0x7fb9104d7700] Reponse envoyee = LOGIN#ok
      [THREAD 0x7fb9104d7700] Attente requete...
      [THREAD 0x7fb90fcd6700] Requete recue = LOGIN#charlet#xyz456
      [THREAD 0x7fb90fcd6700] LOGIN de charlet
      [THREAD 0x7fb90fcd6700] Reponse envoyee = LOGIN#ok
      [THREAD 0x7fb90fcd6700] Attente requete...
      [THREAD 0x7fb90f4d5700] Requete recue = LOGIN#wagner#abc123
      [THREAD 0x7fb90f4d5700] LOGIN de wagner
      [THREAD 0x7fb90f4d5700] Reponse envoyee = LOGIN#ok
      [THREAD 0x7fb90f4d5700] Attente requete...
      [THREAD 0x7fb90fcd6700] Requete recue = LOGOUT
      [THREAD 0x7fb90fcd6700] LOGOUT
      [THREAD 0x7fb90fcd6700] Reponse envoyee = LOGOUT#ok
      [THREAD 0x7fb90fcd6700] Fin de connexion de la socket 5
      [THREAD 0x7fb9104d7700] Requete recue = OPER#*#5#3
      [THREAD 0x7fb9104d7700] OPERATION 5 * 3
      [THREAD 0x7fb9104d7700] Reponse envoyee = OPER#ok#15
      [THREAD 0x7fb9104d7700] Attente requete...
      [THREAD 0x7fb90f4d5700] Requete recue = OPER#/#8#0
      [THREAD 0x7fb90f4d5700] OPERATION 8 / 0
      [THREAD 0x7fb90f4d5700] Reponse envoyee = OPER#ko#Division par zéro !
      [THREAD 0x7fb90f4d5700] Attente requete...
      [THREAD 0x7fb90f4d5700] Requete recue = LOGOUT
      [THREAD 0x7fb90f4d5700] LOGOUT
      [THREAD 0x7fb90f4d5700] Reponse envoyee = LOGOUT#ok
      [THREAD 0x7fb90f4d5700] Fin de connexion de la socket 6
^C
Arret du serveur.
[zeus]$
```

Sur la console du premier **client** (sur **moon**), nous avons

```
[moon]$ Client 192.168.228.169 50000
Connecte sur le serveur.
user: wagner
password: abc123
Login OK.
Operation (<CTRL-C> four fin) : 5 * 3
Résultat = 15
Operation (<CTRL-C> four fin) : 5 + 3
Serveur arrete, pas de reponse reçue...
[moon]$
```

Et sur la console du second **client** (sur **moon**), nous avons

```
[moon]$ Client 192.168.228.169 50000
Connecte sur le serveur.
user: charlet
password: xyz456
Login OK.
Operation (<CTRL-C> four fin) : ^C
Arret du client.
[moon]$
```

Et finalement sur la console du troisième **client** (sur **zeus**), nous avons

```
[zeus]$ ./Client localhost 50000
Connecte sur le serveur.
user: wagner
password: abc123
Login OK.
Operation (<CTRL-C> four fin) : 8 / 0
Erreur: Division par zéro !
Operation (<CTRL-C> four fin) : ^C
Arret du client.
[zeus]$
```

Une fois que les 3 clients ont été loggés (et avant les logout), nous avons sur une autre console de la machine **serveur** :

```
[zeus]$ ps -u student | grep Serveur
30867 pts/1    00:00:00 Serveur
[zeus]$ netstat -an | grep 50000
tcp        0      0 0.0.0.0:50000           0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:47742         127.0.0.1:50000         ESTABLISHED
tcp        0      0 127.0.0.1:50000         127.0.0.1:47742         ESTABLISHED
tcp        0      0 192.168.228.169:50000   192.168.228.167:47326   ESTABLISHED
tcp        0      0 192.168.228.169:50000   192.168.228.167:47328   ESTABLISHED
[zeus]$ lsof -p 30867 -ad "0-10"
COMMAND   PID    USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
Serveur 30867 student    0u   CHR  136,1      0t0    4 /dev/pts/1
Serveur 30867 student    1u   CHR  136,1      0t0    4 /dev/pts/1
Serveur 30867 student    2u   CHR  136,1      0t0    4 /dev/pts/1
Serveur 30867 student    3u  IPv4 436008      0t0  TCP *:50000 (LISTEN)
Serveur 30867 student    4u  IPv4 436009      0t0  TCP
                                  zeus:50000->192.168.228.167:47326 (ESTABLISHED)
Serveur 30867 student    5u  IPv4 436080      0t0  TCP
                                  zeus:50000->192.168.228.167:47328 (ESTABLISHED)
Serveur 30867 student    6u  IPv4 436134      0t0  TCP
                                  localhost:50000->localhost:47742 (ESTABLISHED)
[zeus]$
```

Nous observons que :

- La communication du **premier client** a été interrompue suite à la fin du serveur
- La communication des **2ème et 3ème clients** a été interrompue par une demande de LOGOUT provenant des clients (via une <CTRL-C>)
- La commande **netstat** nos informe sur les connexions établies : la ligne en « gras noir » est celle du client « **localhost** » vers le serveur
- La commande **lsof** nous informe sur les sockets ouvertes du serveur : la **3** est la socket d'écoute tandis que les **4**, **5** et **6** sont les sockets de service

Exemple de serveur multi-threads « en pool »

Dans l'exemple qui suit :

- Nous allons créer un serveur multi-threads **« en pool »**
- Le protocole mis en place sera toujours **SMOP**, donc identique à l'exemple précédent (les fichiers SMOP.h et SMOP.cpp ne seront donc pas recopiés ci-dessous)
- Le serveur sera toujours un **« serveur de connexions »**
- Le **client** est totalement identique à l'exemple précédent, il ne se rend même pas compte que l'on va changer de modèle de serveur (le fichier **Client.cpp** ne sera donc pas recopié ici)
- La seule chose qui change est le **serveur** et donc le fichier **Serveur.cpp**

Voici le code du **serveur** (fichier **Serveur.cpp**) :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>
#include <pthread.h>
#include "TCP.h"
#include "SMOP.h"

void HandlerSIGINT(int s);
void TraitementConnexion(int sService);
void* FctThreadClient(void* p);

int sEcoute;

// Gestion du pool de threads
```

```c
#define NB_THREADS_POOL 2
#define TAILLE_FILE_ATTENTE 20
int socketsAcceptees[TAILLE_FILE_ATTENTE];
int indiceEcriture=0, indiceLecture=0;
pthread_mutex_t mutexSocketsAcceptees;
pthread_cond_t  condSocketsAcceptees;

int main(int argc,char* argv[])
{
  if (argc != 2)
  {
     printf("Erreur...\n");
     printf("USAGE : Serveur portServeur\n");
     exit(1);
  }

  // Initialisation socketsAcceptees
  pthread_mutex_init(&mutexSocketsAcceptees,NULL);
  pthread_cond_init(&condSocketsAcceptees,NULL);
  for (int i=0 ; i<TAILLE_FILE_ATTENTE ; i++)
    socketsAcceptees[i] = -1;

  // Armement des signaux
  struct sigaction A;
  A.sa_flags = 0;
  sigemptyset(&A.sa_mask);
  A.sa_handler = HandlerSIGINT;
  if (sigaction(SIGINT,&A,NULL) == -1)
  {
    perror("Erreur de sigaction");
    exit(1);
  }

  // Creation de la socket d'écoute
  if ((sEcoute = ServerSocket(atoi(argv[1]))) == -1)
  {
     perror("Erreur de ServeurSocket");
     exit(1);
  }

  // Creation du pool de threads
  printf("Création du pool de threads.\n");
  pthread_t th;
  for (int i=0 ; i<NB_THREADS_POOL ; i++)
    pthread_create(&th,NULL,FctThreadClient,NULL);

  // Mise en boucle du serveur
  int sService;
  char ipClient[50];
  printf("Demarrage du serveur.\n");
```

```
  while(1)
  {
    printf("Attente d'une connexion...\n");
    if ((sService = Accept(sEcoute,ipClient)) == -1)
    {
      perror("Erreur de Accept");
      close(sEcoute);
      SMOP_Close();
      exit(1);
    }

    printf("Connexion acceptée : IP=%s socket=%d\n",ipClient,sService);

    // Insertion en liste d'attente et réveil d'un thread du pool
    // (Production d'une tâche)
    pthread_mutex_lock(&mutexSocketsAcceptees);
    socketsAcceptees[indiceEcriture] = sService; // !!!
    indiceEcriture++;
    if (indiceEcriture == TAILLE_FILE_ATTENTE) indiceEcriture = 0;
    pthread_mutex_unlock(&mutexSocketsAcceptees);
    pthread_cond_signal(&condSocketsAcceptees);
  }
}

void* FctThreadClient(void* p)
{
  int sService;

  while(1)
  {
    printf("\t[THREAD %p] Attente socket...\n",pthread_self());

    // Attente d'une tâche
    pthread_mutex_lock(&mutexSocketsAcceptees);
    while (indiceEcriture == indiceLecture)
      pthread_cond_wait(&condSocketsAcceptees,&mutexSocketsAcceptees);

    sService = socketsAcceptees[indiceLecture];
    socketsAcceptees[indiceLecture] = -1;
    indiceLecture++;
    if (indiceLecture == TAILLE_FILE_ATTENTE) indiceLecture = 0;
    pthread_mutex_unlock(&mutexSocketsAcceptees);

    // Traitement de la connexion (consommation de la tâche)
    printf("\t[THREAD %p] Je m'occupe de la socket %d\n",
      pthread_self(),sService);

    TraitementConnexion(sService);
  }
}
```

```
void HandlerSIGINT(int s)
{
  printf("\nArret du serveur.\n");
  close(sEcoute);
  pthread_mutex_lock(&mutexSocketsAcceptees);
  for (int i=0 ; i<TAILLE_FILE_ATTENTE ; i++)
    if (socketsAcceptees[i] != -1) close(socketsAcceptees[i]);
  pthread_mutex_unlock(&mutexSocketsAcceptees);
  SMOP_Close();
  exit(0);
}

void TraitementConnexion(int sService)
{
  char requete[200], reponse[200];
  int nbLus, nbEcrits;
  bool onContinue = true;

  while (onContinue)
  {
    printf("\t[THREAD %p] Attente requete...\n",pthread_self());
    // ***** Reception Requete ******************
    if ((nbLus = Receive(sService,requete)) < 0)
    {
      perror("Erreur de Receive");
      close(sService);
      HandlerSIGINT(0);
    }

    // ***** Fin de connexion ? *****************
    if (nbLus == 0)
    {
      printf("\t[THREAD %p] Fin de connexion du client.\n",pthread_self());
      close(sService);
      return;
    }

    requete[nbLus] = 0;
    printf("\t[THREAD %p] Requete recue = %s\n",pthread_self(),requete);

    // ***** Traitement de la requete ***********
    onContinue = SMOP(requete,reponse,sService);

    // ***** Envoi de la reponse ****************
    if ((nbEcrits = Send(sService,reponse,strlen(reponse))) < 0)
    {
      perror("Erreur de Send");
      close(sService);
      HandlerSIGINT(0);
```

```
    }

    printf("\t[THREAD %p] Reponse envoyee = %s\n",pthread_self(),reponse);

    if (!onContinue)
      printf("\t[THREAD %p] Fin de connexion de la socket %d\n",pthread_self(),sService);
  }
}
```

où nous observons que :

- Avant d'entrer dans sa boucle principale, le **thread serveur** (le thread principal à nouveau ici) crée son **pool de threads**. Ici pour l'exemple, la taille du pool a été fixé à **2** afin d'observer l'attente d'un 3ème client
- Une fois que le **thread serveur** a accepté une connexion, il place la socket de service obtenue dans la **file d'attente** représentée par la vecteur global **socketsAcceptees**. Pour cela, il utilise la variable globale **indiceEcriture** qui lui indique où écrire cette socket. Ces variables étant globales, elles sont protégées par le **mutex mutexSocketsAcceptees**. On retrouve le paradigme classique de réveil des threads utilisant la fonction **pthread_cond_signal** sur la **variable de condition condSocketsAcceptees**
- Une fois démarré, un **thread client** entre dans une boucle infinie dans laquelle
  - Il attend, via le paradigme d'attente basé sur les **variables de condition**) qu'il y ait une socket de service disponible (cela est détectée par le fait que **indiceEcriture** et **indiceLecture** sont différents)
  - Une fois réveillé, il va chercher la prochaine socket de service en attente à l'indice spécifié par **indiceLecture**
  - Il traite la communication en appelant la fonction **TraitementConnexion()** qui n'a pas changé par rapport à l'exemple précédent
- Une fois la communication terminée avec un client, un **thread client** se remet en attente sur la **variable de condition** → classique dans le **modèle en pool**
- **Important** : il faut remarquer que, afin de garder le code le plus lisible possible, rien n'a été géré dans le cas où le nombre de sockets acceptées dans la file d'attente dépasse **TAILLE_FILE_ATTENTE** (20 dans l'exemple) → cela ferait ici planter le serveur → il faudrait refuser toute nouvelle connexion qui ne peut pas être mise en file d'attente

Pour cet exemple,

- Le **serveur** a été lancé sur la machine **zeus** (IP = **192.168.228.169**) et mis en attente sur le port **50000**
- Trois **clients** ont été lancés sur la machine **moon** (IP = **192.168.228.167**)

Sur la console du **serveur**, nous observons

```
[zeus]$ ./Serveur 50000
Création du pool de threads.
Demarrage du serveur.
Attente d'une connexion...
      [THREAD 0x7f139700a700] Attente socket...
      [THREAD 0x7f1396809700] Attente socket...
Connexion acceptée : IP=192.168.228.167 socket=4
Attente d'une connexion...
      [THREAD 0x7f139700a700] Je m'occupe de la socket 4
      [THREAD 0x7f139700a700] Attente requete...
Connexion acceptée : IP=192.168.228.167 socket=5
Attente d'une connexion...
      [THREAD 0x7f1396809700] Je m'occupe de la socket 5
      [THREAD 0x7f1396809700] Attente requete...
Connexion acceptée : IP=192.168.228.167 socket=6
Attente d'une connexion...
      [THREAD 0x7f139700a700] Requete recue = LOGIN#wagner#abc123
      [THREAD 0x7f139700a700] LOGIN de wagner
      [THREAD 0x7f139700a700] Reponse envoyee = LOGIN#ok
      [THREAD 0x7f139700a700] Attente requete...
      [THREAD 0x7f1396809700] Requete recue = LOGIN#charlet#xyz456
      [THREAD 0x7f1396809700] LOGIN de charlet
      [THREAD 0x7f1396809700] Reponse envoyee = LOGIN#ok
      [THREAD 0x7f1396809700] Attente requete...
      [THREAD 0x7f139700a700] Requete recue = OPER#+#5#9
      [THREAD 0x7f139700a700] OPERATION 5 + 9
      [THREAD 0x7f139700a700] Reponse envoyee = OPER#ok#14
      [THREAD 0x7f139700a700] Attente requete...
      [THREAD 0x7f139700a700] Requete recue = LOGOUT
      [THREAD 0x7f139700a700] LOGOUT
      [THREAD 0x7f139700a700] Reponse envoyee = LOGOUT#ok
      [THREAD 0x7f139700a700] Fin de connexion de la socket 4
      [THREAD 0x7f139700a700] Attente socket...
      [THREAD 0x7f139700a700] Je m'occupe de la socket 6
      [THREAD 0x7f139700a700] Attente requete...
      [THREAD 0x7f139700a700] Requete recue = LOGIN#wagner#abc123
      [THREAD 0x7f139700a700] LOGIN de wagner
      [THREAD 0x7f139700a700] Reponse envoyee = LOGIN#ok
      [THREAD 0x7f139700a700] Attente requete...
      [THREAD 0x7f1396809700] Requete recue = LOGOUT
      [THREAD 0x7f1396809700] LOGOUT
```

```
      [THREAD 0x7f1396809700] Reponse envoyee = LOGOUT#ok
      [THREAD 0x7f1396809700] Fin de connexion de la socket 5
      [THREAD 0x7f1396809700] Attente socket...
      [THREAD 0x7f139700a700] Requete recue = LOGOUT
      [THREAD 0x7f139700a700] LOGOUT
      [THREAD 0x7f139700a700] Reponse envoyee = LOGOUT#ok
      [THREAD 0x7f139700a700] Fin de connexion de la socket 6
      [THREAD 0x7f139700a700] Attente socket...
^C
Arret du serveur.
[zeus]$
```

Sur la console du **premier client** nous avons

```
[moon]$ Client 192.168.228.169 50000
Connecte sur le serveur.
user: wagner
password: abc123
Login OK.
Operation (<CTRL-C> four fin) : 5 + 9
Résultat = 14
Operation (<CTRL-C> four fin) : ^C
Arret du client.
[moon]$
```

Sur celle du **second client** :

```
[moon]$ Client 192.168.228.169 50000
Connecte sur le serveur.
user: charlet
password: xyz456
Login OK.
Operation (<CTRL-C> four fin) : ^C
Arret du client.
[moon]$
```

Et sur celle du **troisième client** :

```
[moon]$ Client 192.168.228.169 50000
Connecte sur le serveur.
user: wagner
password: abc123      // Attente…
Login OK.
Operation (<CTRL-C> four fin) : ^C
Arret du client.
[moon]$
```

Nous observons que

- Les 2 threads du **pool de threads** clients ont été créés avant que le **thread serveur** n'ait pu accepté une première connexion
- Dans la console du 3ème client, il y a un délai invisible ici entre le moment où l'utilisateur entre son mot de passe et le moment où il peut envoyer sa première requête de calcul → les 2 threads du pool sont actuellement occupés par les 2 premiers clients → il faut attendre que le 1er client quitte la communication pour que le thread du pool libéré puisse s'occuper du 3ème client
- Les threads du pool ne s'arrêtent jamais → il faut attendre l'arrêt du serveur (par un <CTRL-C>) pour que celui-ci ferme toutes les sockets ouvertes (celles dans la **file d'attente** et celles éventuellement encore en cours d'utilisation par protocole **SMOP** pour une communication en cours avec un client) et termine le processus, terminant ainsi tous les threads