TP Développement Réseau : Socket TCP (Windows)

Sommaire

Les objectifs de ce tp sont de mettre en oeuvre la programmation réseau en utilisant l'interface socket pour le mode connecté (TCP) sous Windows.

Remarque : les TP *ont pour but d'établir ou de renforcer vos compétences pratiques. Vous pouvez penser que vous comprenez tout ce que vous lisez ou tout ce que vous a dit votre enseignant mais la répétition et la pratique sont nécessaires pour développer des compétences en programmation. Ceci est comparable au sport ou à la musique ou à tout autre métier demandant un long entraînement pour acquérir l'habileté nécessaire. Imaginez quelqu'un qui voudrait disputer une compététion dans l'un de ces domaines sans pratique régulière. Vous savez bien quel serait le résultat.*

L'interface socket

Pré-requis

La mise en oeuvre de l'interface socket nécessite de connaître :
- L'architecture client/serveur
- L'adressage IP et les numéros de port
- Notions d'API (appels systèmes sous Unix) et de programmation en langage C
- Les protocoles TCP et UDP, les modes connecté et non connecté

Définition

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

Intégration d'IP dans Unix BSD (1981)
Interface de programmation socket de Berkeley (1982) : la plus utilisée et intégrée dans le noyau

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

Socket : mécanisme de communication bidirectionelle entre processus

Manuel du pogrammeur

Le développeur utilisera donc concrètement une interface pour programmer une application TCP/IP grâce par exemple :
- à l'API **Socket BSD** sous Unix/Linux ou
- à l'API **WinSocket** sous Microsoft ©Windows

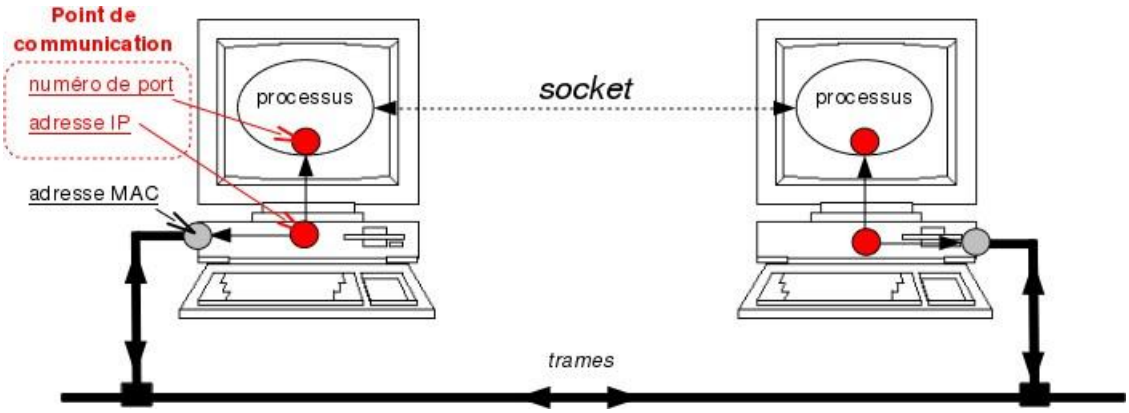
Les pages man principales sous Unix/Linux concernant la programmation réseau sont regroupées dans le chapitre 7 :
- socket(7) : interface de programmation des sockets
- packet(7) : interface par paquet au niveau périphérique
- raw(7) : sockets brutes (raw) IPv4 sous Linux
- ip(7) : implémentation Linux du protocole IPv4
- udp(7) : protocole UDP pour IPv4
- tcp(7) : protocole TCP

L'accès aux pages man se fera donc avec la commande `man`, par exemple : `man 7 socket`

Pour Microsoft ©Windows, on pourra utiliser le service en ligne MSDN :
- Windows Socket 2 : msdn.microsoft.com/en-us/library/ms740673(VS.85).aspx
- Les fonctions Socket : msdn.microsoft.com/en-us/library/ms741394(VS.85).aspx

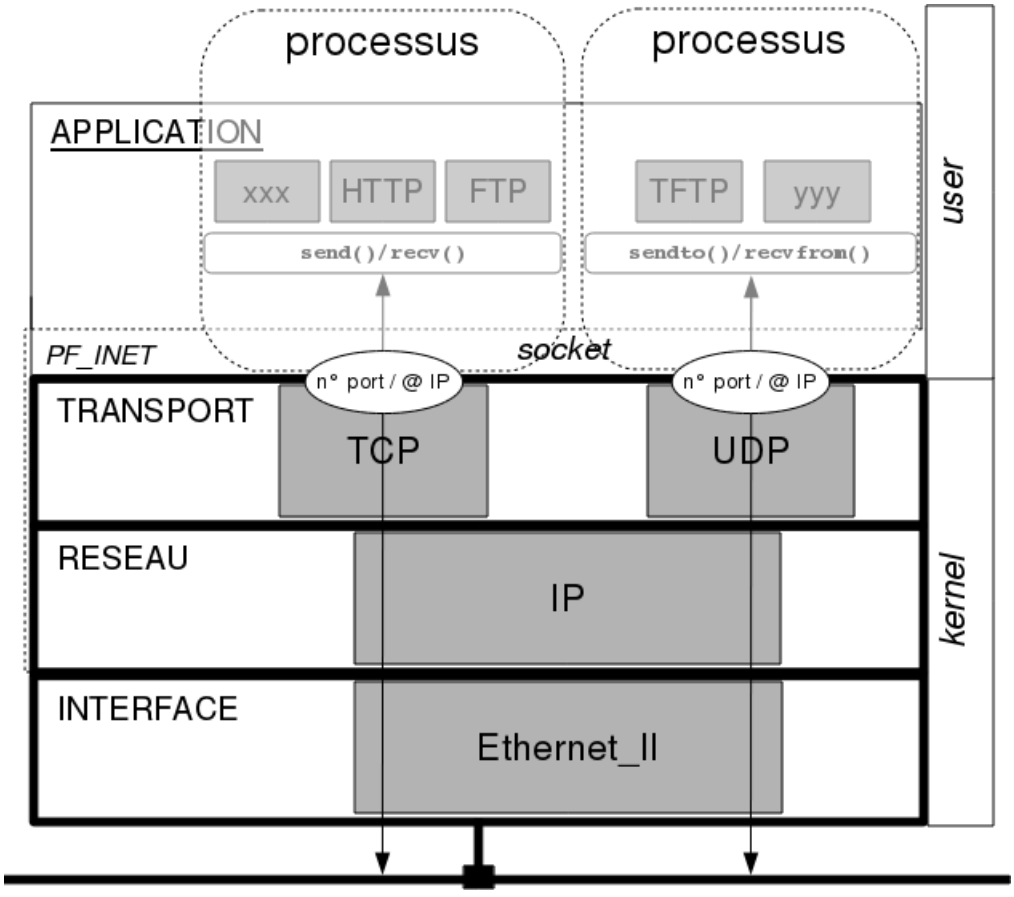
Modèle

Rappel : une socket est un point de communication par lequel un processus peut émettre et recevoir des données.



Ce point de communication devra être relié à une adresse IP et un numéro de port dans le cas des protocoles Internet.

Une socket est communément représentée comme un point d'entrée initial au niveau TRANSPORT du modèle à couches DoD dans la pile de protocole.



Exemple de processus TCP et UDP

Couche Transport

Rappel : la couche Transport est responsable du transport des messages complets de bout en bout (soit de processus à processus) au travers du réseau.

En programmation, si on utilise comme point d'entrée initial le niveau TRANSPORT, il faudra alors choisir un des deux protocoles de cette couche :
- **TCP** (*Transmission Control Protocol*) est un protocole de transport fiable, en **mode connecté** (RFC 793).
- **UDP** (*User Datagram Protocol*) est un protocole souvent décrit comme étant non-fiable, en **mode non-connecté** (RFC 768), mais plus rapide que TCP.

Numéro de ports

Rappel : un numéro de port sert à identifier un processus (l'application) en cours de communication par l'intermédiaire de son protocole de couche application (associé au service utilisé, exemple : 80 pour HTTP).

Pour chaque port, un numéro lui est attribué (codé sur 16 bits), ce qui implique qu'il existe un maximum de 65 536 ports (2^{16}) par machine et par protocoles TCP et UDP.

L'attribution des ports est faite par le système d'exploitation, sur demande d'une application. Ici, il faut distinguer les deux situations suivantes :
- cas d'un **processus client** : le numéro de port utilisé par le client sera envoyé au processus serveur. Dans ce cas, le processus client peut demander à ce que le système d'exploitation lui attribue n'importe quel port, à condition qu'il ne soit pas déjà attribué.
- cas d'un **processus serveur** : le numéro de port utilisé par le serveur doit être connu du processus client. Dans ce cas, le processus serveur doit demander un numéro de port précis au système d'exploitation qui vérifiera seulement si ce numéro n'est pas déjà attribué.

Une liste des ports dits réservés est disponible dans le chier /etc/services sous Unix/Linux.

Caractéristiques des sockets

Rappel : les sockets compatibles BSD représentent une interface uniforme entre le processus utilisateur (user) et les piles de protocoles réseau dans le noyau (kernel) de l'OS.

Pour dialoguer, chaque processus devra préalablement créer une socket de communication en indiquant : – le **domaine** de communication : ceci sélectionne la famille de protocole à employer. Il faut savoir que chaque famille possède son adressage. Par exemple pour les protocoles Internet IPv4, on utilisera le domaine `PF_INET` ou `AF_INET`.
- le **type** de socket à utiliser pour le dialogue. Pour `PF_INET`, on aura le choix entre : `SOCK_STREAM` (qui correspond à un mode connecté), `SOCK_DGRAM` (qui correspond à un mode non connecté) ou `SOCK_RAW` (qui permet un accès direct aux protocoles de la couche Réseau comme IP, ICMP, ...).
- le **protocole** à utiliser sur la socket. Le numéro de protocole dépend du domaine de communication et du type de la socket. Normalement, il n'y a qu'un seul protocole par type de socket pour une famille donnée (`SOCK_STREAM` ⇒ `TCP` et `SOCK_DGRAM` ⇒ `UDP`). Néanmoins, rien ne s'oppose à ce que plusieurs protocoles existent, auquel cas il est nécessaire de le spécifier (c'est la cas pour `SOCK_RAW` où il faudra préciser le protocole à utiliser).

Une socket appartient à une famille. Il existe plusieurs types de sockets. Chaque famille possède son adressage.

Manipulations

Objectifs

L'objectif de cette partie est la mise en oeuvre d'une communication client/serveur en utilisant une socket TCP sous Windows.

Winsock (WINdows SOCKet) est une bibliothèque logicielle pour Windows dont le but est d'implémenter une interface de programmation inspirée des sockets BSD. Son développement date de 1991 (Microsoft n'a pas implémenté Winsock 1.0.). Il y a peu de di érences avec les sockets BSD, mais Winsock fournit des fonctions additionnelles pour être conforme au modèle de programmation Windows, par exemple les fonctions `WSAGetLastError()`, `WSAStartup()` et `WSACleanup()`.

Pour ce TP, vous aurez besoin d'outils de compilation et fabrication de programmes sous Windows. `MinGW` (*Minimalist GNU for Windows*) est une adaptation des logiciels de développement et de compilation du GNU (`GCC` : *GNU Compiler Collection*) à la plate-forme Windows (Win32). Vous pouvez l'installer indépendamment ou intégrer à un EDI comme `Dev-Cpp` ou `Code::Blocks`.

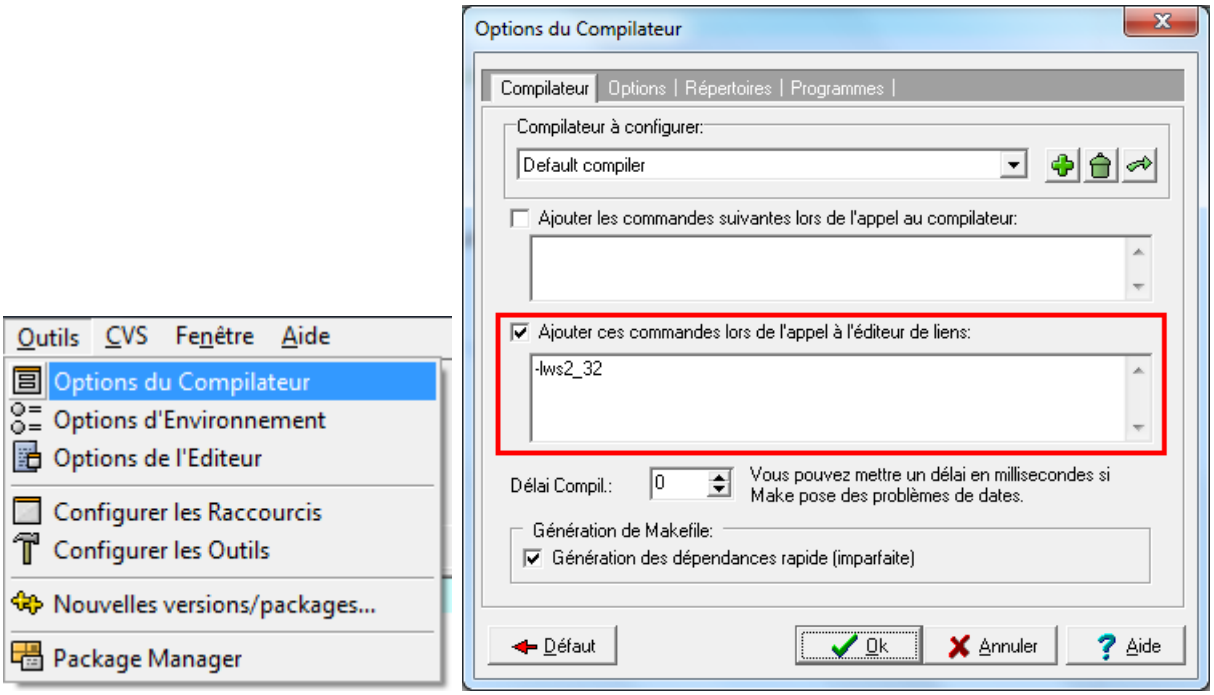
Étape n°0 : préparation

Sous Windows, il faut tout d'abord initialiser avec `WSAStartup()` l'utilisation de la DLL Winsock par le processus. De la même manière, il faudra terminer son utilisation proprement avec `WSACleanup()`.

```c
#include <winsock2.h>

// pour Visual Studio sinon ajouter -lws2_32
#pragma comment(lib, "ws2_32.lib")

int main()
{
    WSADATA WSAData; // variable initialisée par WSAStartup

    WSAStartup(MAKEWORD(2,0), &WSAData); // indique la version utilisée, ici 2.0

    /* ... */

    WSACleanup(); // termine l' utilisation

    return 0;
}
```

Un client TCP en C (itération 0)

Dans cet exemple, on utilise la version 2 de Winsock (`winsock2.h` et `ws2_32.lib`). Vous pouvez aussi utiliser la version 1 (`winsock.h` et `wsock32.lib`). Avec les compilateurs type GCC, il faudra ajouter `-lws2_32` à l'édition des liens.



Exemple : Lien vers la bibliothèque ws2_32. lib dans Dev-Cpp

Étape n°1 : création de la socket (côté client)

Pour créer une socket, on utilisera l'appel `socket()`. On commence par consulter la documentation associée à cet appel : http ://msdn.microsoft.com/en-us/library/ms740506(v=vs.85).aspx.

The socket function creates a socket that is bound to a specific transport service provider.

```
SOCKET WSAAPI socket(
  _In_ int af,
  _In_ int type,
  _In_ int protocol
);

...
```

Extrait de la documentation de l'appel socket

À l'aide d'un éditeur de texte (`notepad++` ou d'un EDI comme `Dev-Cpp` ou `Code::Blocks` sous Windows), tapez (à la main, pas de copier/coller, histoire de bien mémoriser !) le programme suivant dans un fichier que vous nommerez "`clientTCPWin-1.c`" :

```c
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
    WSADATA WSAData; // variable initialisée par WSAStartup

    WSAStartup(MAKEWORD(2,0), &WSAData); // indique la version utilisée, ici 2.0
```

```
    //--- Début de l' étape n° 1 :
    SOCKET descripteurSocket;
    int iResult;

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

    if (descripteurSocket == INVALID_SOCKET)
    {
        printf("Erreur creation socket : %d¥n", WSAGetLastError()); WSACleanup();
        return 1;
    }

    //--Fin de l' étape n° 1 !
    printf("Socket créée avec succès !¥n");

    // On ferme la ressource avant de quitter

    iResult = closesocket(descripteurSocket);
    if (iResult == SOCKET_ERROR)
    {
        printf("Erreur fermeture socket : %d¥n", WSAGetLastError()); WSACleanup();
        return 1;
    }

    WSACleanup(); // termine l' utilisation

    return 0;
}
```

Un client TCP en C (itération 1)

Pour le paramètre `protocol`, on a utilisé la valeur `0` (voir commentaire). On aurait pu préciser le protocole TCP de la manière suivante : `IPPROTO_TCP`.

Étape n°2 : connexion au serveur

Maintenant que nous avons créé une socket TCP, il faut la connecter au processus serveur distant.

Pour cela, on va utiliser l'appel système `connect()`. On commence par consulter la documentation associée à cet appel : http ://msdn.microsoft.com/en-us/library/ms737625(v=vs.85).aspx.

`The connect function establishes a connection to a specified socket.`

```
int connect(
  _In_ SOCKET s,
  _In_ const struct sockaddr *name,
  _In_ int namelen
);
```

. . .

Extrait de la page de documentation de l'appel connect

On rappelle que l'adressage du processus distant dépend du **domaine** de communication (cad la famille de protocole employée). Ici, nous avons choisi le domaine `AF_INET` pour les protocoles Internet IPv4.

Dans cette famille, un processus sera identifié par :
– une adresse IPv4
– un numéro de port

L'interface socket propose une structure d'adresse générique :

```
struct sockaddr {
        ushort sa_family;
        char    sa_data[14];
};

typedef struct sockaddr SOCKADDR;
```

La structure générique sockaddr

Et le domaine `AF_INET` utilise une structure compatible :

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

struct sockaddr_in {
        short  sin_family;
        u_short sin_port;
        struct in_addr sin_addr;
        char    sin_zero[8];
};

typedef struct sockaddr_in SOCKADDR_IN;
```

La structure compatible sockaddr_in pour AF_INET

Il suffit donc d'initialiser une structure `sockaddr_in` avec les informations distantes du serveur (adresse IPv4 et numéro de port).

Pour écrire ces informations dans la structure d'adresse, il nous faudra utiliser :
– `inet_addr()` pour convertir une **adresse IP** depuis la notation IPv4 décimale pointée vers une forme binaire (dans l'ordre d'octet du réseau)
– `htons()` pour convertir le **numéro de port** (sur 16 bits) depuis l'ordre des octets de l'hôte vers celui du réseau.

L'ordre des octets du réseau est en fait big-endian. Il est donc plus prudent d'appeler des fonctions qui respectent cet ordre pour coder des informations dans les en-têtes des protocoles réseaux.

Éditer le programme suivant dans un fichier que vous nommerez "`clientTCPWin-2.c`" :

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <stdlib.h>

#define PORT 5000
```

```
int main()
{
    WSADATA WSAData; // variable initialisée par WSAStartup

    WSAStartup(MAKEWORD(2,0), &WSAData); // indique la version utilisée, ici 2.0

    SOCKET descripteurSocket;
    int iResult;

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

    if (descripteurSocket == INVALID_SOCKET)
    {
        printf("Erreur creation socket : %d¥n", WSAGetLastError()); WSACleanup();
        return 1;
    }

    //--- Début de l'étape n° 2 :
    struct sockaddr_in pointDeRencontreDistant; // ou SOCKADDR_IN pointDeRencontreDistant;

    // Renseigne la structure sockaddr_in avec les informations du serveur distant
    pointDeRencontreDistant.sin_family = AF_INET;
    // On choisit l'adresse IPv4 du serveur
    pointDeRencontreDistant.sin_addr.s_addr = inet_addr("192.168.52.2"); // à modifier selon ses
        besoins
    // On choisit le numéro de port d'écoute du serveur
    pointDeRencontreDistant.sin_port = htons(PORT); // = 5000

    // Débute la connexion vers le processus serveur distant
    iResult = connect(descripteurSocket, (SOCKADDR *)&pointDeRencontreDistant, sizeof(
        pointDeRencontreDistant));
    if (iResult == SOCKET_ERROR)
    {
        printf("Erreur connexion socket : %d¥n", WSAGetLastError());
        iResult = closesocket(descripteurSocket); // On ferme la ressource avant de quitter if
        (iResult == SOCKET_ERROR)
        {
          printf("Erreur fermeture socket : %d¥n", WSAGetLastError());
        }
        WSACleanup();
        return 1; // On sort en indiquant un code erreur
    }

    //--- Fin de l'étape n° 2 !
    printf("Connexion au serveur réussie avec succès !¥n");

    // On ferme la ressource avant de quitter
    iResult  =  closesocket(descripteurSocket);
    if (iResult == SOCKET_ERROR)
```

```
    {
        printf("Erreur fermeture socket : %d¥n", WSAGetLastError()); WSACleanup();
        return 1;
    }

    WSACleanup(); // termine l' utilisation

    return 0;
}
```

Un client TCP en C (itération 2)

Si vous testez ce client, vous risquez d'obtenir :

```
Erreur connexion socket : 10061
```

Ceci peut s'expliquer tout simplement parce qu'il n'y a pas de processus serveur à cette adresse !

La fonction `WSAGetLastError()` retourne seulement un code d'erreur. L'ensemble des code d'erreurs sont déclarés dans le chier d'en-tête `winsock2.h`. Par exemple ici, le code d'erreur 10061 correspond à l'étiquette `WSAECONNREFUSED` (connexion refusée). Pour obtenir le message associé à un code d'erreur, il faut utiliser la fonction `FormatMessageW()`.

```
wchar_t *s = NULL;
FormatMessageW(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM |
    FORMAT_MESSAGE_IGNORE_INSERTS, NULL, WSAGetLastError(), MAKELANGID(LANG_NEUTRAL,
    SUBLANG_DEFAULT), (LPWSTR)&s, 0, NULL);
fprintf(stderr, "%S¥n", s); LocalFree(s);
```

Affichage du message associé à un code d'erreur

Étape n°3 : vérification du bon fonctionnement de la connexion

Pour tester notre client, il nous faut un serveur ! Pour cela, on va utiliser l'outil réseau `netcat` en mode serveur (`-l`) sur le port 5000 (`-p 5000`) :

```
nc -l -p 5000
```

Puis en exécutant `clientTCPWin-2.exe`, on obtient :

```
Connexion au serveur réussie avec succès !
```

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

Étape n°4 : échange des données

On rappelle qu'une communication TCP est bidirectionnelle *full duplex* et orientée flux d'octets. Il nous faut donc des fonctions pour écrire (envoyer) et lire (recevoir) des octets dans la socket.

Normalement les octets envoyés ou reçus respectent un protocole de couche APPLICATION. Ici, pour les tests, notre couche APPLICATION sera vide! C'est-à-dire que les données envoyées et reçues ne respecteront aucun protocole et ce seront de simples caractères ASCII.

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

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

Les appels `recv()` et `send()` sont spéci ques aux sockets en mode connecté (TCP).

Faire communiquer deux processus sans aucun protocole de couche APPLICATION est tout de même difficile ! On va simplement fixer les règles d'échange suivantes :

– le client envoie en premier une chaîne de caractères
– et le serveur lui répondra "ok"

Éditer le programme suivant dans un fichier que vous nommerez "`clientTCPWin-3.c`":

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <stdlib.h>

#define PORT 5000
#define LG_MESSAGE 256

int main()
{
    WSADATA WSAData; // variable initialisée par WSAStartup

    WSAStartup(MAKEWORD(2,0), &WSAData); // indique la version utilisée, ici 2.0 SOCKET

    descripteurSocket;
    int iResult;

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

    if (descripteurSocket == INVALID_SOCKET)
    {
        printf("Erreur creation socket : %d¥n", WSAGetLastError()); WSACleanup();
        return 1;
    }

    struct sockaddr_in pointDeRencontreDistant; // ou SOCKADDR_IN pointDeRencontreDistant;

    // Renseigne la structure sockaddr_in avec les informations du serveur distant
    pointDeRencontreDistant.sin_family = AF_INET;
    // On choisit l' adresse IPv4 du serveur
```

```
pointDeRencontreDistant.sin_addr.s_addr = inet_addr("192.168.52.2"); // à modifier selon ses
    besoins
// On choisit le numéro de port d'écoute du serveur
pointDeRencontreDistant.sin_port = htons(PORT); // = 5000

// Débute la connexion vers le processus serveur distant
iResult = connect(descripteurSocket, (SOCKADDR *)&pointDeRencontreDistant, sizeof(
   pointDeRencontreDistant));
if (iResult == SOCKET_ERROR)
{
    printf("Erreur connexion socket : %d¥n", WSAGetLastError());
    iResult = closesocket(descripteurSocket); // On ferme la ressource avant de quitter if
    (iResult == SOCKET_ERROR)
    {
      printf("Erreur fermeture socket : %d¥n", WSAGetLastError());
    }
    WSACleanup();
    return 1; // On sort en indiquant un code erreur
}

printf("Connexion au serveur réussie avec succès !¥n");

//--- Début de l'étape n° 4 :
char messageEnvoi[LG_MESSAGE]; /* le message de la couche Application ! */ char
messageRecu[LG_MESSAGE]; /* le message de la couche Application ! */ int
ecrits, lus; /* nb d'octets ecrits et lus */

sprintf(messageEnvoi, "Hello world !¥n");
ecrits = send(descripteurSocket, messageEnvoi, (int)strlen(messageEnvoi), 0); // message à
    TAILLE variable
if (ecrits == SOCKET_ERROR)
{
    printf("Erreur envoi socket : %d¥n", WSAGetLastError());
    closesocket(ConnectSocket);
    WSACleanup();
    return 1;
}

printf("Message %s envoyé avec succès (%d octets)¥n¥n", messageEnvoi, ecrits);

/* Reception des données du serveur */
lus = recv(ConnectSocket, messageRecu, sizeof(messageRecu), 0); /* attend un message de
   TAILLE fixe */
if( lus > 0 ) /* réception de n octets */
    printf("Message reçu du serveur : %s (%d octets)¥n¥n", messageRecu, lus);
else if ( lus == 0 ) /* la socket est fermée par le serveur */
    printf("La socket a été fermée par le serveur !¥n");
else /* une erreur ! */
    printf("Erreur lecture socket : %d¥n", WSAGetLastError());
//--- Fin de l'étape n° 4 !

// On ferme la ressource avant de quitter
iResult = closesocket(descripteurSocket);
```

```
    if (iResult == SOCKET_ERROR)
    {
        printf("Erreur fermeture socket : %d¥n", WSAGetLastError()); WSACleanup();
        return 1;
    }

    WSACleanup(); // termine l' utilisation

    return 0;
}
```

Un client TCP en C (itération 3)

On utilise la même procédure de test que précédemment en démarrant un serveur `netcat` sur le port `5000` :

```
nc -l -p 5000
```

Puis, on exécute notre client `clientTCPWin-3.exe` :

```
Connexion au serveur réussie avec succès !
Message Hello world !
 envoyé avec succès (14 octets)

Message reçu du serveur : ok (3
 octets)
```

Dans la console où on a exécuté le serveur `netcat`, cela donne :

```
Hello world !
ok
```

Dans `netcat`, pour envoyer des données au client, il su t de saisir son message et de valider par la touche Entrée.

Que se passe-t-il si le serveur s'arrête (en tapant Ctrl-C par exemple !) au lieu d'envoyer "ok" ?

```
Hello world !
^C
```

Dans la console du client `clientTCPWin-3.exe` :

```
Connexion au serveur réussie avec succès !
Message Hello world !
 envoyé avec succès (14 octets)

La socket a été fermée par le serveur !
```

Notre client a bien détecté la fermeture de la socket côté serveur.

Dans les codes sources ci-dessus, nous avons utilisés l'appel `close()` pour fermer la socket et donc la communication. En TCP, la communication étant bidirectionnelle *full duplex*, il est possible de fermer plus finement l'échange en utilisant l'appel `shutdown()` :

```
The shutdown function disables sends or receives on a socket.

int shutdown(
  _In_ SOCKET s,
  _In_ int how
);

s [in] : A descriptor identifying a socket.
how [in] : A flag that describes what types of operation will no longer be allowed. Possible
     values for this flag are listed in the Winsock2.h header file.


SD_RECEIVE : Shutdown receive operations. 0

SD_SEND : Shutdown send operations. 1

SD_BOTH : Shutdown both send and receive operations. 2

...
```

Extrait de la documentation de l'appel shutdown

Étape n°5 : réalisation d'un serveur TCP

Évidemment, un serveur TCP a lui aussi besoin de créer une socket SOCK_STREAM dans le domaine AF_INET.

Mis à part cela, le code source d'un serveur TCP basique est très différent d'un client TCP dans le principe. On va détailler ces différences étape par étape.

On rappelle qu'un serveur TCP attend des demandes de connexion en provenance de processus client. Le processus client doit connaître au moment de la connexion le numéro de port d'écoute du serveur.

Pour mettre en oeuvre cela, le serveur va utiliser l'appel bind() qui va lui permettre de lier sa socket d'écoute à une interface et à un numéro de port local à sa machine.

```
The bind function associates a local address with a socket.

int bind(
  _In_ SOCKET s,
  _In_ const struct sockaddr *name,
  _In_ int namelen
);

...
```

Extrait de la documentation de l'appel bind

On rappelle que l'adressage d'un processus (local ou distant) dépend du **domaine** de communication (cad la famille de protocole employée). Ici, nous avons choisi le domaine AF_INET pour les protocoles Internet IPv4.

Dans cette famille, un processus sera identifié par : –
une adresse IPv4

– un numéro de port

Rappel : l'interface socket propose une structure d'adresse générique `sockaddr` *et le domaine* `AF_INET` *utilise une structure compatible* `sockaddr_in`.

Il suffit donc d'initialiser une structure `sockaddr_in` avec les informations **locales du serveur** (adresse IPv4 et numéro de port).

Pour écrire ces informations dans la structure d'adresse, il nous faudra utiliser :
– `htonl()` pour convertir une adresse IP (sur 32 bits) depuis l'ordre des octets de l'hôte vers celui du réseau
– `htons()` pour convertir le numéro de port (sur 16 bits) depuis l'ordre des octets de l'hôte vers celui du réseau.

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

Éditer le programme suivant dans un fichier que vous nommerez "`serveurTCPWin-1.c`" :

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> /* pour sleep */

#define PORT 5000

int main()
{
    WSADATA WSAData; // variable initialisée par WSAStartup

    WSAStartup(MAKEWORD(2,0), &WSAData); // indique la version utilisée, ici 2.0 SOCKET

    socketEcoute;
    int iResult;

    //--- Début de l'étape n° 5 :
    // Crée un socket de communication
    socketEcoute = socket(AF_INET, SOCK_STREAM, 0); /* 0 indique que l'on utilisera le
        protocole par défaut associé à SOCK_STREAM soit TCP */

    if (socketEcoute == INVALID_SOCKET)
    {
        printf("Erreur creation socket : %d¥n", WSAGetLastError()); WSACleanup();
        return 1;
    }

    // On prépare l'adresse d'attachement locale
    struct sockaddr_in pointDeRencontreLocal; // ou SOCKADDR_IN pointDeRencontreLocal;

    // Renseigne la structure sockaddr_in avec les informations locales du serveur
    pointDeRencontreLocal.sin_family = AF_INET;
```

```
    pointDeRencontreLocal.sin_addr.s_addr = htonl(INADDR_ANY); // toutes les interfaces
        locales disponibles
    // On choisit le numéro de port d'écoute du serveur
    pointDeRencontreLocal.sin_port = htons(PORT); // = 5000

    iResult = bind(socketEcoute, (SOCKADDR *)&pointDeRencontreLocal, sizeof(
        pointDeRencontreLocal));
    if (iResult == SOCKET_ERROR)
    {
        printf("Erreur bind socket : %d¥n", WSAGetLastError());
        closesocket(socketEcoute);
        WSACleanup();
        return 1;
    }

    //--- Fin de l'étape n° 5 !
    printf("Socket attachée avec succès !¥n");

    // On s'endort ... (cf. test)
    sleep(2);

    // On ferme la ressource avant de quitter
    iResult = closesocket(socketEcoute);
    if (iResult == SOCKET_ERROR)
    {
        printf("Erreur fermeture socket : %d¥n", WSAGetLastError()); WSACleanup();
        return 1;
    }

    WSACleanup(); // termine l'utilisation

    return 0;
}
```

Un serveur TCP en C (itération 1)

Le test est concluant :

```
Socket attachée avec succès !
```

Attention, tout de même de bien comprendre qu'un numéro de port identifie un processus communiquant ! Exécutons deux fois le même serveur et on obtient alors :

```
bind: Address already in use
```

Explication : l'attachement local au numéro de port 5000 du deuxième processus échoue car ce numéro de port est déjà attribué par le système d'exploitation au premier processus serveur.

Étape n°6 : mise en attente des connexions

Maintenant que le serveur a créé et attaché une socket d'écoute, il doit la placer en attente passive, c'est-à-dire capable d'accepter les demandes de connexion des processus clients.

Pour cela, on va utiliser l'appel `listen()` :

```
The listen function places a socket in a state in which it is listening for an incoming
    connection.

int listen(
  _In_ SOCKET s,
  _In_ int backlog
);


s [in] : A descriptor identifying a bound, unconnected socket.
backlog [in] : The maximum length of the queue of pending connections. If set to SOMAXCONN, the
    underlying service provider responsible for socket s will set the backlog to a maximum
    reasonable value. There is no standard provision to obtain the actual backlog value.
...
```

Extrait de la documentation de l'appel listen

Si la le est pleine, le serveur sera dans une situation de DOS (Deny Of Service) car il ne peut plus traiter les nouvelles demandes de connexion.

Éditer le programme suivant dans un fichier que vous nommerez "serveurTCPWin-2.c":

```c
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> /* pour sleep */

#define PORT 5000

int main()
{
    WSADATA WSAData; // variable initialisée par WSAStartup

    WSAStartup(MAKEWORD(2,0), &WSAData); // indique la version utilisée, ici 2.0 SOCKET

    socketEcoute;
    int iResult;

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

    if (socketEcoute == INVALID_SOCKET)
    {
        printf("Erreur creation socket : %d¥n", WSAGetLastError()); WSACleanup();
        return 1;
    }

    // On prépare l'adresse d'attachement locale
    struct sockaddr_in pointDeRencontreLocal; // ou SOCKADDR_IN pointDeRencontreLocal;
```

```
    // Renseigne la structure sockaddr_in avec les informations locales du serveur
    pointDeRencontreLocal.sin_family = AF_INET;
    pointDeRencontreLocal.sin_addr.s_addr = htonl(INADDR_ANY); // toutes les interfaces locales
        disponibles
    // On choisit le numéro de port d'écoute du serveur
    pointDeRencontreLocal.sin_port = htons(PORT); // = 5000

    iResult = bind(socketEcoute, (SOCKADDR *)&pointDeRencontreLocal, sizeof(
        pointDeRencontreLocal));
    if (iResult == SOCKET_ERROR)
    {
        printf("Erreur bind socket : %d¥n", WSAGetLastError());
        closesocket(socketEcoute);
        WSACleanup();
        return 1;
    }

    printf("Socket attachée avec succès !¥n");

    //--- Début de l'étape n° 6 :
    // On fixe la taille de la file d'attente (pour les demandes de connexion non encore
        traitées)
    if (listen(socketEcoute, SOMAXCONN) == SOCKET_ERROR)
    {
        printf("Erreur listen socket : %d¥n", WSAGetLastError());
    }

    //--- Fin de l'étape n° 6 !
    printf("Socket placée en écoute passive ...¥n");

    // On ferme la ressource avant de quitter
    iResult = closesocket(socketEcoute);
    if (iResult == SOCKET_ERROR)
    {
        printf("Erreur fermeture socket : %d¥n", WSAGetLastError());
        WSACleanup();
        return 1;
    }

    WSACleanup(); // termine l'utilisation

    return 0;
}
```

Un serveur TCP en C (itération 2)

Étape n°7 : accepter les demandes connexions

Cette étape est cruciale pour le serveur. Il lui faut maintenant accepter les demandes de connexion en provenance des processus client.

Pour cela, il va utiliser l'appel `accept()` :

The accept function permits an incoming connection attempt on a socket. SOCKET

```
accept(
  _In_     SOCKET s,
  _Out_    struct sockaddr *addr,
  _Inout_ int *addrlen
);
```

s [in] : A descriptor that identifies a socket that has been placed in a listening state with the listen function. The connection is actually made with the socket that is returned by accept.
addr [out] : An optional pointer to a buffer that receives the address of the connecting entity, as known to the communications layer. The exact format of the addr parameter is determined by the address family that was established when the socket from the sockaddr structure was created.
addrlen [in, out] : An optional pointer to an integer that contains the length of structure pointed to by the addr parameter.

...

Extrait de la documentation de l'appel accept

Explication : imaginons qu'un client se connecte à notre socket d'écoute. L'appel `accept()` va retourner une nouvelle socket connectée au client qui servira de socket de dialogue. La socket d'écoute reste inchangée et peut donc servir à accepter des nouvelles connexions.

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

On va pour l'instant ignorer ce problème et mettre en oeuvre un serveur basique : c'est-à-dire mono-client (ou plus exactement un client après l'autre)!

Concernant le dialogue, on utilisera les mêmes fonctions `recv()`/`send()`) que le client. Éditer

le programme suivant dans un fichier que vous nommerez "`serveurTCPWin-3.c`" :

```c
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> /* pour sleep */

#define PORT 5000
#define LG_MESSAGE 256

int main()
{
    WSADATA WSAData; // variable initialisée par WSAStartup

    WSAStartup(MAKEWORD(2,0), &WSAData); // indique la version utilisée, ici 2.0 SOCKET

    socketEcoute;
```

```
int iResult;

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

if (socketEcoute == INVALID_SOCKET)
{
    printf("Erreur creation socket : %d¥n", WSAGetLastError()); WSACleanup();
    return 1;
}

// On prépare l'adresse d'attachement locale
struct sockaddr_in pointDeRencontreLocal; // ou SOCKADDR_IN pointDeRencontreLocal;

// Renseigne la structure sockaddr_in avec les informations locales du serveur
pointDeRencontreLocal.sin_family = AF_INET;
pointDeRencontreLocal.sin_addr.s_addr = htonl(INADDR_ANY); // toutes les interfaces locales
    disponibles
// On choisit le numéro de port d'écoute du serveur
pointDeRencontreLocal.sin_port = htons(PORT); // = 5000

iResult = bind(socketEcoute, (SOCKADDR *)&pointDeRencontreLocal, sizeof(
    pointDeRencontreLocal));
if (iResult == SOCKET_ERROR)
{
    printf("Erreur bind socket : %d¥n", WSAGetLastError());
    closesocket(socketEcoute);
    WSACleanup();
    return 1;
}

printf("Socket attachée avec succès !¥n");

// On fixe la taille de la file d'attente (pour les demandes de connexion non encore
    traitées)
if (listen(socketEcoute, SOMAXCONN) == SOCKET_ERROR)
{
    printf("Erreur listen socket : %d¥n", WSAGetLastError());
}

printf("Socket placée en écoute passive ...¥n");

//--- Début de l'étape n° 7 :
SOCKET socketDialogue;
struct sockaddr_in pointDeRencontreDistant;
int longueurAdresse = sizeof(pointDeRencontreDistant);
char messageEnvoi[LG_MESSAGE]; /* le message de la couche Application ! */ char
messageRecu[LG_MESSAGE]; /* le message de la couche Application ! */ int
ecrits, lus; /* nb d'octets ecrits et lus */

// boucle d'attente de connexion : en théorie, un serveur attend indéfiniment !
```

```
while(1)
{
    memset(messageEnvoi, 0x00, LG_MESSAGE*sizeof(char));
    memset(messageRecu, 0x00, LG_MESSAGE*sizeof(char));
    printf("Attente d' une demande de connexion (quitter avec Ctrl-C)¥n¥n");
    // c' est un appel bloquant
    socketDialogue = accept(socketEcoute, (SOCKADDR *)&pointDeRencontreDistant, &
        longueurAdresse);

    if (socketDialogue == INVALID_SOCKET)
    {
      printf("Erreur accept socket : %d¥n", WSAGetLastError()); closesocket(socketEcoute);
      WSACleanup();
      return 1;
    }

    // On réception les données du client (cf. protocole !)
    // ici appel bloquant
    lus = recv(socketDialogue, messageRecu, sizeof(messageRecu), 0); /* attend un message
        de TAILLE fixe */
    if( lus > 0 ) /* réception de n octets */
        printf("Message reçu du client : %s (%d octets)¥n¥n", messageRecu, lus);
    else if ( lus == 0 ) /* la socket est fermée par le serveur */
        printf("socket fermé¥n");
    else /* une erreur ! */
        printf("Erreur lecture socket : %d¥n", WSAGetLastError());

    // On envoie des données vers le client (cf. protocole !)
    sprintf(messageEnvoi, "ok¥n");
    ecrits = send(socketDialogue, messageEnvoi, (int)strlen(messageEnvoi), 0); // message à
        TAILLE variable
    if (ecrits == SOCKET_ERROR)
    {
        printf("Erreur envoi socket : %d¥n", WSAGetLastError()); closesocket(socketDialogue);
        WSACleanup();
        return 1;
    }

    printf("Message %s envoyé (%d octets)¥n¥n", messageEnvoi, ecrits);

    // On ferme la socket de dialogue et on se replace en attente ...
    closesocket(socketDialogue);
}

//--- Fin de l' étape n° 7 !
// On ferme la ressource avant de quitter
iResult = closesocket(socketEcoute);
if (iResult == SOCKET_ERROR)
{
    printf("Erreur fermeture socket : %d¥n", WSAGetLastError());
    WSACleanup();
```

```
        return 1;
    }

    WSACleanup(); // termine l' utilisation

    return 0;
}
```

Un serveur TCP en C (itération 3)

Testons notre serveur `serveurTCPWin-3.exe` avec notre client :

```
Socket attachée avec succès ! Socket
placée en écoute passive ...
Attente d' une demande de connexion (quitter avec Ctrl-C)

Message reçu : Hello world !
 (14 octets)

Message ok
 envoyé (3 octets)

Attente d' une demande de connexion (quitter avec Ctrl-C)

Message reçu : Hello world !
 (14 octets)

Message ok
 envoyé (3 octets)

Attente d' une demande de connexion (quitter avec Ctrl-C)

^C
```

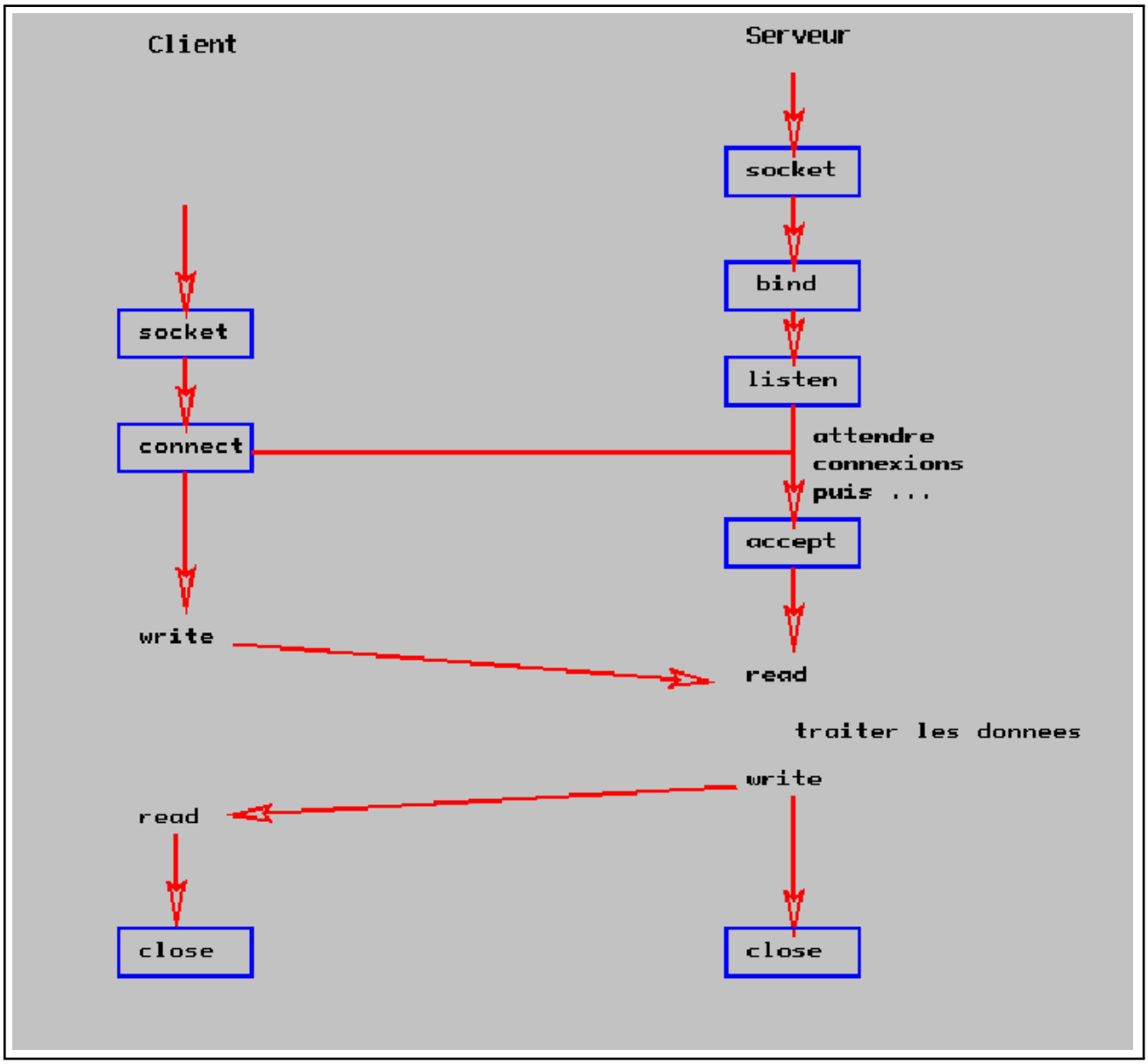
On va exécuter deux clients à la suite :

```
Connexion au serveur réussie avec succès !
Message Hello world !
 envoyé avec succès (14 octets)

Message reçu du serveur : ok (3
 octets)

Connexion au serveur réussie avec succès !
Message Hello world !
 envoyé avec succès (14 octets)

Message reçu du serveur : ok (3
 octets)
```

Il est évidemment possible de tester notre serveur avec des clients TCP existants comme `telnet` ou `netcat`.

Bilan

L'échange entre un client et un serveur TCP peut être maintenant schématisé de la manière suivante :



Les appels systèmes utilisés dans un échange TCP

Questions de révision

L'idée de base des questions de révision est de vous donner une chance de voir si vous avez identifié et compris les points clés de ce TP.

Question 1. Qu'est-ce qu'une *socket* ?

Question 2. Quelles sont les trois caractéristiques d'une *socket* ?

Question 3. Quelles sont les deux informations qui définissent un point de communication en IPv4 ?

Question 4. Comment le serveur connaît-il le port utilisé par le client ?

Question 5. Comment le client connaît-il le port utilisé par le serveur ?

Question 6. À quelle couche du modèle DoD est reliée l'interface de programmation *socket* ?

Question 7. Quel protocole de niveau Transport permet d'établir une communication en mode connecté ?

Question 8. Quel protocole de niveau Transport permet d'établir une communication en mode non- connecté ?

Question 9. Quel est l'ordre des octets en réseau ?

Question 10. À quels protocoles correspond le domaine `PF_INET` ou `AF_INET` ? Est-ce le seul utilisable avec l'interface socket ? En citer au moins un autre.