Notes sur les réseaux

par Emmanuel Delahaye (Espace personnel d'Emmanuel Delahaye)

Date de publication : 27 janvier 2008

Dernière mise à jour : 26 septembre 2010

Notes brutes concernant les réseaux

Votre avis et vos suggestions sur cet article nous intéressent !
Alors après votre lecture, n'hésitez pas :

I - Dialogue Terminal Modem

II - Réalisation d'un projet réseau

La realisation d'un projet réseau est basée sur l'utilisation des sockets (BSD, Windows). Il y a quelques nuances selon le système utilisé, mais dans l'ensemble, les fonctions et comportements sont identiques. Le but de cet article est de montrer la réalisation de clients / serveurs en UDP/IP et TCP/IP.

II-A - Sockets

II-A-1 - Introduction

Les sockets sont des flux de données (octets) très similaires aux flux d'entrée / sortie standards ou aux fichiers, mais qui permettent de réaliser des connexions de données bidirectionnelles entre des machines (locales ou distantes) via un réseau de données (boucle, réseau local, Internet, X.25 etc.). Ils sont mis en oeuvre via une série de fonctions regroupées sous le nom de API sockets ou sockets BSD ou sockets.

II-A-2 - Protocoles

Les connexions de données sont gérées par différent protocoles (niv 3 et 4) et différentes liaisons (niv. 2) et interfaces physiques (niv. 1). Les sockets ignorent les interfaces (gérées par les drivers systèmes), mais connaissent les protocoles de niveau 3 et 4 (IP, UNIX / TCP, UDP, etc.). Ensuite, ils savent travailler en mode non connecté (datagrams, simple, pas de vérification, ordre indéterminé) ou connecté (paquets, plus complexe, données vérifiées, integrité des données, ordre garanti).

Par exemple, le protocole de niveau 3 IP (Internet Protocol) sait travailler en mode de niveau 4 connecté (TCP) ou non connecté (UDP).

II-A-3 - Fonctions

Les sockets sont manipulés par des fonctions générales :

- socket()
- close() ou closesocket()
- send(), recv(), (mode connecté)
- sendto(), recvfrom() (mode non connecté)
- et des fonctions spécialisées qui dépendent si l'application est un serveur ou un client et si on utilise un mode connecté ou non

II-A-4 - Serveur

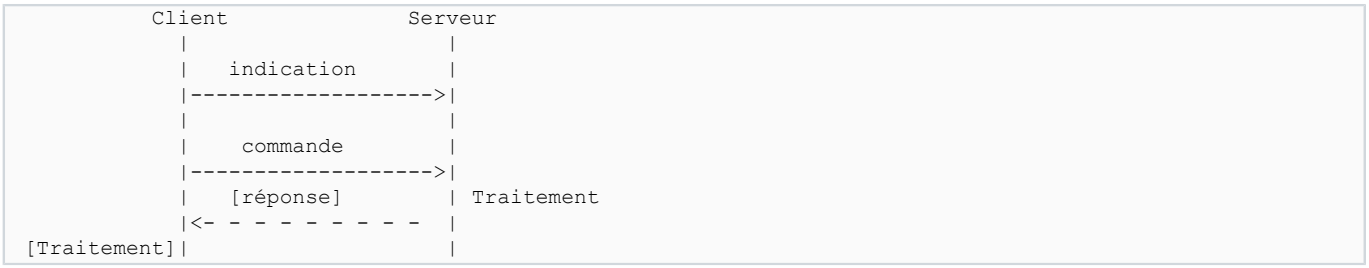
- bind() (TCP)
- listen()
- accept() (mode connecté)

II-A-5 - Client

- connect() (mode connecté)

*Rappel : La documentation complète des fonctions se trouve dans les 'pages man', comme, par exemple, **celles-ci**.*

II-B - Réalisation d'un Client / Serveur UDP/IP



Le mode UDP (*Datagram sockets*) est rustique et ne necessite pas de connexion. Il agit par échange de blocs de données appelés 'Datagrammes'. Un serveur ne peut pas émettre spontanément de données vers un client, car il ne le connait pas. Il ne peut que répondre à une commande en utilisant les données d'adressage reçues avec la commande à traiter.

II-B-1 - Serveur UDP/IP

Les opérations à réaliser sont:

- initialisation
  - ouverture d'un socket en mode datagramme (UDP/IP): socket(AF_INET, SOCK_DGRAM, 0)
  - configurer l'adresse et le port: bind()
- dans une boucle
  - tester la réception : select() [Facultatif]
  - lire le bloc reçu : recvfrom()
  - traitement des données...
  - émission du bloc : sendto()
- fin
  - fermeture du socket : closesocket()

II-B-2 - Client UDP/IP

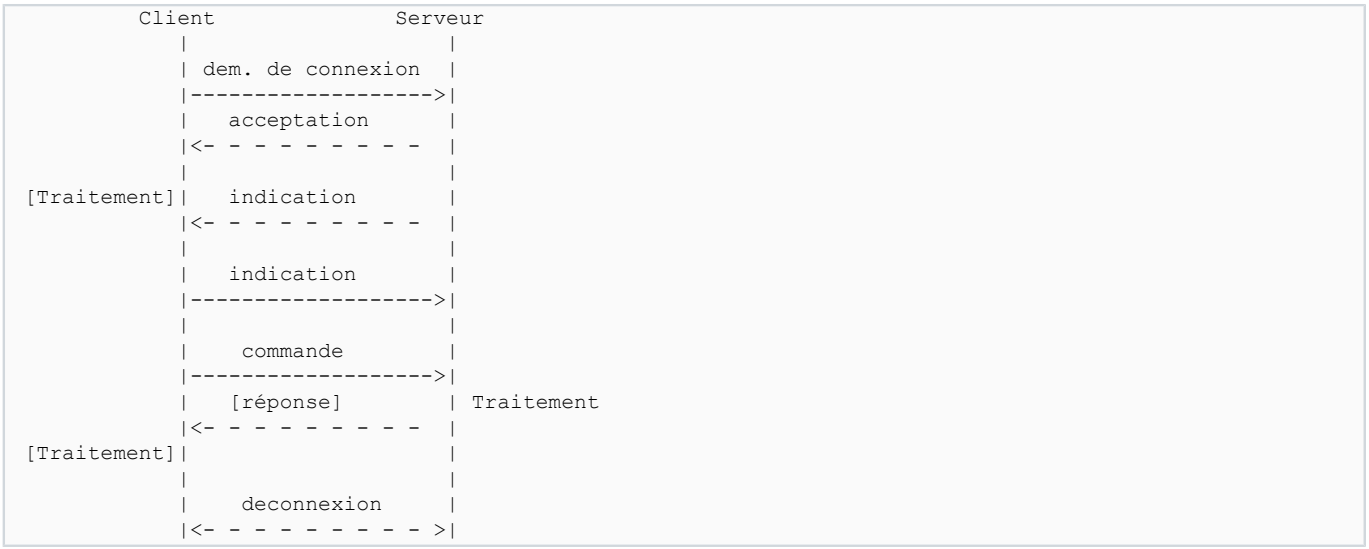
Les opérations à réaliser sont:

- initialisation
  - ouverture d'un socket en mode datagramme (UDP/IP): socket(AF_INET, SOCK_DGRAM, 0)
- selon la demande
  - fabrication des données à émettre...
  - émission du bloc : sendto()
  - tester la réception : select() [Facultatif]
  - lire le bloc reçu : recvfrom()
  - traitement des données reçues...
- fin
  - fermeture du socket : closesocket()

Exemple de client / serveur UDP :

- **serveur**
- **client**

II-C - Réalisation d'un Client / Serveur TCP/IP



Le mode TCP (*Stream sockets*) est solide. Il garanti la transmission des données. Il nécessite l'établissement d'une connexion. Ensuite, les blocs de données peuvent être échangés.

II-C-1 - Serveur TCP/IP

Les opérations à réaliser sont:

- initialisation
  - ouverture d'un socket en mode flux (TCP/IP): socket(AF_INET, SOCK_STREAM, 0)
  - configurer l'adresse et le port: bind()
  - configurer le nombre d'écoutes: listen()
- dans une boucle
  - accepter une connexion: accept()
  - tester la réception : select() [Facultatif]
  - recevoir des données: recv()
  - émettre des données: send()
  - déconnecter : shutdown()
- fin
  - fermeture du socket : closesocket()

II-C-2 - Client TCP/IP

Les opérations à réaliser sont:

- initialisation
  - ouverture d'un socket en mode flux (TCP/IP): socket(AF_INET, SOCK_STREAM, 0)
  - connexion: connect()
- selon la demande
  - émettre des données: send()
  - tester la réception : select() [Facultatif]
  - recevoir des données: recv()
- fin
  - déconnecter : shutdown()
  - fermeture du socket : closesocket()

client minimum (emission)

II-C-3 - Serveur multiclient en TCP/IP

Le principe est de réduire la boucle principale à la surveillance des connexions entrantes (accept()) et de créer un thread comprenant la boucle de traitement avec les données de la connexion client.

thread 'serveur' : dans une boucle :

- attendre une connexion: accept()
- créer un thread avec le socket client

thread 'client' : dans une boucle :

- attendre des données et les lire: recv()
- émettre des données en réponse: send()

Apres la boucle (deconnexion, timeout...)

- déconnecter : shutdown()

III - Réalisation pas à pas d'un serveur TCP/IP

III-A - Introduction

Il s'agit d'une application de mise en oeuvre écrite en langage C. Le détail des fonctions (paramètres, comportement) doit être consulté dans les documents de références habituels (msdn, man etc.).

III-A-1 - Avertissement

Cette réalisation demande une bonne connaissance du langage C. Cependant, je m'efforce de n'utiliser que des concepts simples du langage. Si je juge qu'un point est difficile, je fais une remarque ou je renvoie à une explication détaillée. En cas de difficulté avérée, poser une question dans la rubrique réseau du forum.

III-A-2 - Environnement de développement

Le code est écrit et validé sous Windows avec Dev-C++/Code::Blocks et la blibiothèque -lws2_32. Il compile sous Linux, mais n'est pas validé pour le moment. (Si quelqu'un veut le faire, il est le bienvenu)

III-A-3 - Environnement de test

Il suffit d'une machine supportant le protocole TCP/IP, et munie d'une adresse IP (sinon, utiliser 127.0.0.1). **Si on dispose de 2 machines**, on peut espionner les échanges de trames avec EtherReal.

III-A-4 - Traitement des erreurs

J'ai développé la fonction portable 'psock_perror()' basée sur perror() et WSAGetLastError() selon le système. Un exemple minimum est donné dans le code source suivant nommé "02.c".

III-B - 01 - Initialisation, terminaison

Sous Windows, il est nécessaire d'indiquer au système que le processus courant veut utiliser les sockets. Ce n'est pas nécessaire sous Linux. Sous Windows, winsock2 est compatible avec les sockets BSD. Il est donc indispensable de vérifier qu'on a bien la bonne version de sockets sur la machine cible. De plus, Windows exige de signaler la fin d'utilisation des sockets par le processus.

Voici notre premier programme 'réseau'. Pour le moment, il ne fait rien sous unixoïde, mais sous Windows, il vérifie la version des sockets.

```
/* 01.c */

#ifdef __cplusplus
#error Be sure you are using a C compiler...
#endif

#if defined (WIN32) || defined (_WIN32)

#include <winsock2.h>

#elif defined (linux) || defined (_POSIX_VERSION) || defined (_POSIX2_C_VERSION)\
 || defined (_XOPEN_VERSION)

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

```
#include <unistd.h> /* close */

#define INVALID_SOCKET -1
#define SOCKET_ERROR -1

#define closesocket(s) close (s)
typedef int SOCKET;
typedef struct sockaddr_in SOCKADDR_IN;
typedef struct sockaddr SOCKADDR;

#else
#error not defined for this platform
#endif

#include <stdio.h>
#include <stdlib.h>

/* macros ============================================================ */
/* constants ========================================================= */
/* types ============================================================= */
/* structures ======================================================== */
/* private data ====================================================== */
/* private functions ================================================= */
/* entry point ======================================================= */

/* ------------------------------------------------------------------------
   ------------------------------------------------------------------------ */
int main (void)
{
   int ret;

#if defined (WIN32) || defined (_WIN32)
   WSADATA wsa_data;
   int err = WSAStartup (MAKEWORD (2, 2), &wsa_data);

   if (!err)
   {
      puts ("WIN: winsock2: OK");
   }
#else
   int err = 0;
   {
#endif
      if (!err)
         /* to be continued ... */

#if defined (WIN32) || defined (_WIN32)
      WSACleanup ();
#else
   }
#endif

   if (err)
   {
      ret = EXIT_FAILURE;
   }
   else
   {
      ret = EXIT_SUCCESS;
   }

   return ret;
}
```

Sortie attendue :

```
WIN: winsock2: OK
```

III-C - 02 - Création, suppression de sockets

Ce programme ne fait rien de visible. Il se contente de créer un socket en mode 'Internet' (IP), puis de le fermer proprement.

Nouvelles fonctions : socket(), closesocket()

Traitement portable des erreurs

```
/* 02.c */

#ifdef __cplusplus
#error Be sure you are using a C compiler...
#endif

#if defined (WIN32) || defined (_WIN32)

#include <winsock2.h>

#elif defined (linux) || defined (_POSIX_VERSION) || defined (_POSIX2_C_VERSION)\
 || defined (_XOPEN_VERSION)

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h> /* close */

#define INVALID_SOCKET -1
#define SOCKET_ERROR -1

#define closesocket(s) close (s)
typedef int SOCKET;
typedef struct sockaddr_in SOCKADDR_IN;
typedef struct sockaddr SOCKADDR;

#else
#error not defined for this platform
#endif

#include <stdio.h>
#include <stdlib.h>

/* macros ======================================================== */

#define NELEM(a) (sizeof(a)/sizeof*(a))

/* constants ===================================================== */
/* types ========================================================= */
/* structures ==================================================== */
/* private data ================================================== */
/* private functions ============================================= */

/* ------------------------------------------------------------------
   ------------------------------------------------------------------ */
static void psock_perror (char const *cmt)
{
#if defined (WIN32) || defined (_WIN32)

   fprintf (stderr, "%s: ", cmt);
   {
      DWORD err = WSAGetLastError ();
      wchar_t msg[1024];
      FormatMessageW (FORMAT_MESSAGE_FROM_SYSTEM, NULL, err, 0,
                      msg, NELEM (msg), NULL);
      fwprintf (stderr, L"%s\n", msg);
   }
```

```
#elif defined (linux) || defined (_POSIX_VERSION) || defined (_POSIX2_C_VERSION)\
 || defined (_XOPEN_VERSION)

   perror (cmt);

#else
#error not defined for this platform
#endif

}

/* ---------------------------------------------------------------------
   --------------------------------------------------------------------- */
static int app (void)
{
   int err = 0;

   /* open a socket in TCP/IP mode. */
   SOCKET sock = socket (AF_INET, SOCK_STREAM, 0);

   if (sock != INVALID_SOCKET)
   {
      printf ("socket %d is now opened in TCP/IP mode\n", sock);

      {
         int sock_err;

         /* to be continued ... */

         printf ("closing socket %d...\n", sock);

         /* close the socket. */
         sock_err = closesocket (sock), sock = INVALID_SOCKET;

         printf ("the socket is now closed\n");

         if (sock_err)
         {
            psock_perror ("socket.close");
            err = 1;
         }
      }
   }
   else
   {
      psock_perror ("socket.open");
      err = 1;
   }

   return err;
}

/* entry point ============================================================ */

/* ---------------------------------------------------------------------
   --------------------------------------------------------------------- */
int main (void)
{
   int ret;
#if defined (WIN32) || defined (_WIN32)
   WSADATA wsa_data;
   int err = WSAStartup (MAKEWORD (2, 2), &wsa_data);

   if (!err)
   {
      puts ("WIN: winsock2: OK");
#else
   int err = 0;
   {
#endif

      err = app ();
```

```
#if defined (WIN32) || defined (_WIN32)
      WSACleanup ();
#else
   }
#endif

   if (err)
   {
      ret = EXIT_FAILURE;
   }
   else
   {
      ret = EXIT_SUCCESS;
   }
   return ret;
}
```

Exemple de sortie attendue

```
WIN: winsock2: OK
socket 44 is now opened in TCP/IP mode
closing socket 44...
the socket is now closed
```

III-D - 03 - Mini serveur : mise en écoute du port 23

Le port 23 est le port Telnet par défaut. La encore, rien de spectaculaire, si ce n'est que si la machine dispose d'un pare-feu (comme Zone Alarm, par exemple), celui-ci signale que le programme veut agir en tant que serveur sur le port SMTP. C'est la conséquence de l'exécution de la fonction 'listen()'. Si le pare-feu demande confirmation, l'exécution du programme est suspendue jusqu'à ce que la confirmation soit validée. Ensuite, comme il n'y a pour le moment rien d'autre à faire, le programme se termine.

Attention, sous unixoïde, le port 23 n'est accessible que si on est en root.

Nouvelles fonctions : bind(), listen(), htons(), htonl().

```
/* 03s.c */
#ifdef __cplusplus
#error Be sure you are using a C compiler...
#endif

#if defined (WIN32) || defined (_WIN32)

#include <winsock2.h>

#elif defined (linux) || defined (_POSIX_VERSION) || defined (_POSIX2_C_VERSION)\
 || defined (_XOPEN_VERSION)

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h> /* close */

#define INVALID_SOCKET -1
#define SOCKET_ERROR -1

#define closesocket(s) close (s)
typedef int SOCKET;
typedef struct sockaddr_in SOCKADDR_IN;
typedef struct sockaddr SOCKADDR;

#else
#error not defined for this platform
```

```
#endif

#include <stdio.h>
#include <stdlib.h>

/* macros ================================================================ */

#define TELNET 23

/* we want to listen to the TELNET port */
#define PORT TELNET

/* constants ============================================================= */
/* types ================================================================= */
/* structures ============================================================ */
/* private data ========================================================== */
/* private functions ===================================================== */

/* ------------------------------------------------------------------------
   ------------------------------------------------------------------------ */
static int app (void)
{
   int err = 0;

   /* open a socket in TCP/IP mode. */
   SOCKET sock = socket (AF_INET, SOCK_STREAM, 0);

   if (sock != INVALID_SOCKET)
   {
      printf ("socket %d is now opened in TCP/IP mode\n", sock);

      {
         int sock_err;
         /* assign the listening port */
         SOCKADDR_IN sin =
         {0};

         /* automatic IP address */
         sin.sin_addr.s_addr = htonl (INADDR_ANY);

         /* protocol family (IP) */
         sin.sin_family = AF_INET;

         /* listening port */
         sin.sin_port = htons (PORT);

         /* bind */
         sock_err = bind (sock, (SOCKADDR *) &sin, sizeof sin);

         if (sock_err != SOCKET_ERROR)
         {
            /* start listening (server mode) */
            sock_err = listen (sock, 5);

            printf ("listening on port %d...\n", PORT);

            if (sock_err != SOCKET_ERROR)
            {
               /* to be continued ... */
            }
            else
            {
               err = 1;
            }
         }
         else
         {
            err = 1;
         }

         printf ("closing socket %d...\n", sock);
```

```
         /* close the socket. */
         sock_err = closesocket (sock), sock = INVALID_SOCKET;

         printf ("the socket is now closed\n");

         if (sock_err)
         {
            perror ("socket.close");
            err = 1;
         }
      }
   }
   else
   {
      perror ("socket.open");
      err = 1;
   }

   return err;
}

/* entry point ================================================================ */

/* ---------------------------------------------------------------------------
   --------------------------------------------------------------------------- */
int main (void)
{
   int ret;
#if defined (WIN32) || defined (_WIN32)
   WSADATA wsa_data;
   int err = WSAStartup (MAKEWORD (2, 2), &wsa_data);

   if (!err)
   {
      puts ("WIN: winsock2: OK");
#else
   int err = 0;
   {
#endif

      err = app ();

#if defined (WIN32) || defined (_WIN32)
      WSACleanup ();
#else
   }
#endif

   if (err)
   {
      ret = EXIT_FAILURE;
   }
   else
   {
      ret = EXIT_SUCCESS;
   }
   return ret;
}
```

Exemple de sortie attendue

```
WIN: winsock2: OK
socket 44 is now opened in TCP/IP mode
listening on port 23...
closing socket 44...
the socket is now closed
```

III-E - 04 - Mini serveur : connexion du client/ Déconnexion du client

Le serveur va maintenant se bloquer en attente de connexion d'un client. On peut alors lancer un client Telnet sur cette machine (ou une autre sur le même réseau) avec le port 23. On constate alors, sur le client, que la connection est immédiatement perdue. Normal, car le serveur a renvoyé une deconnexion immédiate du socket client.

Nouvelles fonctions : accept(), shutdown(), inet_ntoa().

```
/* 04s.c */
#ifdef __cplusplus
#error Be sure you are using a C compiler...
#endif

#if defined (WIN32) || defined (_WIN32)

#include <winsock2.h>

#elif defined (linux) || defined (_POSIX_VERSION) || defined (_POSIX2_C_VERSION)\
 || defined (_XOPEN_VERSION)

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h> /* close */

#define INVALID_SOCKET -1
#define SOCKET_ERROR -1

#define closesocket(s) close (s)
typedef int SOCKET;
typedef struct sockaddr_in SOCKADDR_IN;
typedef struct sockaddr SOCKADDR;

#else
#error not defined for this platform
#endif

#include <stdio.h>
#include <stdlib.h>

/* macros ================================================================ */

/* we want to listen to the TELNET port */
#define TELNET 23

#define PORT TELNET

/* constants ============================================================= */
/* types ================================================================= */
/* structures ============================================================ */
/* private data ========================================================== */
/* private functions ===================================================== */

/* ------------------------------------------------------------------------
   ------------------------------------------------------------------------ */
static int app (void)
{
   int err = 0;

   /* open a socket in TCP/IP mode. */
   SOCKET sock = socket (AF_INET, SOCK_STREAM, 0);

   if (sock != INVALID_SOCKET)
   {
      printf ("socket %d is now opened in TCP/IP mode\n", sock);

      {
         int sock_err;
```

```
        /* assign the listening port */
        SOCKADDR_IN sin =
        {0};

        /* automatic IP address */
        sin.sin_addr.s_addr = htonl (INADDR_ANY);

        /* protocol family (IP) */
        sin.sin_family = AF_INET;

        /* listening port */
        sin.sin_port = htons (PORT);

        /* bind */
        sock_err = bind (sock, (SOCKADDR *) &sin, sizeof sin);

        if (sock_err != SOCKET_ERROR)
        {
           /* start listening (server mode) */
           sock_err = listen (sock, 5);

           printf ("listening on port %d...\n", PORT);

           if (sock_err != SOCKET_ERROR)
           {
              /* wait for a client connection */
              printf ("waiting for a client connection on port %d...\n", PORT);
              {
                 SOCKADDR_IN csin =
                 {0};
                 int recsize = (int) sizeof csin;
                 SOCKET csock = accept (sock, (SOCKADDR *) & csin, &recsize);

                 if (csock != INVALID_SOCKET)
                 {
                    printf ("client connected with socket %d from %s:%d\n"
                            ,csock
                            ,inet_ntoa (csin.sin_addr)
                            ,htons (csin.sin_port));

                    /* to be continued ... */

                    shutdown (csock, 2);
                    printf ("closing client socket %d...\n", csock);
                    closesocket (csock), csock = INVALID_SOCKET;

                 }
                 else
                 {
                    perror ("socket.accept");
                    err = 1;
                 }
              }
           }
           else
           {
              perror ("socket.listen");
              err = 1;
           }
        }
        else
        {
           perror ("socket.bind");
           err = 1;
        }

        printf ("closing socket %d...\n", sock);

        /* close the socket. */
        sock_err = closesocket (sock), sock = INVALID_SOCKET;

        printf ("the socket is now closed\n");
```

```
         if (sock_err)
         {
            perror ("socket.close");
            err = 1;
         }
      }
   }
   else
   {
      perror ("socket.open");
      err = 1;
   }

   return err;
}

/* entry point ================================================================ */

/* ---------------------------------------------------------------------------
   --------------------------------------------------------------------------- */
int main (void)
{
   int ret;
#if defined (WIN32) || defined (_WIN32)
   WSADATA wsa_data;
   int err = WSAStartup (MAKEWORD (2, 2), &wsa_data);

   if (!err)
   {
      puts ("WIN: winsock2: OK");
#else
   int err = 0;
   {
#endif

      err = app ();

#if defined (WIN32) || defined (_WIN32)
      WSACleanup ();
#else
   }
#endif

   if (err)
   {
      ret = EXIT_FAILURE;
   }
   else
   {
      ret = EXIT_SUCCESS;
   }
   return ret;
}
```

Exemple de sortie attendue

```
WIN: winsock2: OK
socket 44 is now opened in TCP/IP mode
listening on port 23...
waiting for a client connection on port 23...
client connected with socket 48 from 192.168.0.17:1290
closing client socket 48...
closing socket 44...
the socket is now closed
```

III-F - 05 - Réception d'un caractère puis fermeture

Une fois que le client Telnet est lance, le serveur se bloque sur l'attente d'un bloc de donné. Le client telnet envoyant en temps réel toutes la frappes du clavier, un seul caractère suffit à débloquer le serveur qui envoi alors une commande de deconnexion.

Nouvelles fonctions : recv().

```
/* 05s.c */
#ifdef __cplusplus
#error Be sure you are using a C compiler...
#endif

#if defined (WIN32) || defined (_WIN32)

#include <winsock2.h>

#elif defined (linux) || defined (_POSIX_VERSION) || defined (_POSIX2_C_VERSION)\
 || defined (_XOPEN_VERSION)

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h> /* close */

#define INVALID_SOCKET -1
#define SOCKET_ERROR -1

#define closesocket(s) close (s)
typedef int SOCKET;
typedef struct sockaddr_in SOCKADDR_IN;
typedef struct sockaddr SOCKADDR;

#else
#error not defined for this platform
#endif

#include <stdio.h>
#include <stdlib.h>

/* macros ================================================================ */

/* we want to listen to the TELNET port */
#define TELNET 23

#define PORT TELNET


/* constants ============================================================= */
/* types ================================================================= */
/* structures ============================================================ */
/* private data ========================================================== */
/* private functions ===================================================== */

/* ------------------------------------------------------------------------
   ------------------------------------------------------------------------ */
static int app (void)
{
   int err = 0;

   /* open a socket in TCP/IP mode. */
   SOCKET sock = socket (AF_INET, SOCK_STREAM, 0);

   if (sock != INVALID_SOCKET)
   {
      printf ("socket %d is now opened in TCP/IP mode\n", sock);

      {
```

```
int sock_err;
/* assign the listening port */
SOCKADDR_IN sin =
{0};

/* automatic IP address */
sin.sin_addr.s_addr = htonl (INADDR_ANY);

/* protocol family (IP) */
sin.sin_family = AF_INET;

/* listening port */
sin.sin_port = htons (PORT);

/* bind */
sock_err = bind (sock, (SOCKADDR *) &sin, sizeof sin);

if (sock_err != SOCKET_ERROR)
{
   /* start listening (server mode) */
   sock_err = listen (sock, 5);

   printf ("listening on port %d...\n", PORT);

   if (sock_err != SOCKET_ERROR)
   {
      /* wait for a client connection */
      printf ("waiting for a client connection on port %d...\n", PORT);

      {
         SOCKADDR_IN csin =
         {0};
         int recsize = (int) sizeof csin;
         SOCKET csock = accept (sock, (SOCKADDR *) & csin, &recsize);

         if (csock != INVALID_SOCKET)
         {
            printf ("client connected with socket %d from %s:%d\n"
                    ,csock
                    ,inet_ntoa (csin.sin_addr)
                    ,htons (csin.sin_port));

            {
               /* wait for the receive of a data block (string expected, hence + 1) */
               unsigned char data[128 + 1];
               sock_err = recv (csock, data, (int) sizeof data - 1, 0);

               if (sock_err != SOCKET_ERROR)
               {
                  size_t nb_rec = (size_t) sock_err;
                  /* convert to string */
                  data[nb_rec] = 0;
                  printf ("%u byte%s received:\n%s\n"
                          ,(unsigned) nb_rec
                          ,nb_rec > 1 ? "s" : ""
                          ,data);
                  fflush (stdout);

                  /* to be continued ... */
               }
               else
               {
                  perror ("socket.recv");
                  err = 1;
               }
            }
            shutdown (csock, 2);
            printf ("closing client socket %d...\n", csock);
            closesocket (csock), csock = INVALID_SOCKET;

         }
         else
```

```
                  {
                     perror ("socket.accept");
                     err = 1;
                  }
               }
            }
            else
            {
               perror ("socket.listen");
               err = 1;
            }
         }
         else
         {
            perror ("socket.bind");
            err = 1;
         }

         printf ("closing socket %d...\n", sock);

         /* close the socket. */
         sock_err = closesocket (sock), sock = INVALID_SOCKET;

         printf ("the socket is now closed\n");

         if (sock_err)
         {
            perror ("socket.close");
            err = 1;
         }
      }
   }
   else
   {
      perror ("socket.open");
      err = 1;
   }

   return err;
}

/* entry point ================================================================ */

/* ---------------------------------------------------------------------------
   --------------------------------------------------------------------------- */
int main (void)
{
   int ret;
#if defined (WIN32) || defined (_WIN32)
   WSADATA wsa_data;
   int err = WSAStartup (MAKEWORD (2, 2), &wsa_data);

   if (!err)
   {
      puts ("WIN: winsock2: OK");
#else
   int err = 0;
   {
#endif

      err = app ();

#if defined (WIN32) || defined (_WIN32)
      WSACleanup ();
#else
   }
#endif

   if (err)
   {
      ret = EXIT_FAILURE;
   }
```

```
    else
    {
        ret = EXIT_SUCCESS;
    }
    return ret;
}
```

Exemple de sortie attendue

L'utilisateur tape 'a' sur la console du client:

```
WIN: winsock2: OK
socket 44 is now opened in TCP/IP mode
listening on port 23...
waiting for a client connection on port 23...
client connected with socket 48 from 192.168.0.17:1299
1 byte received:
a
closing client socket 48...
closing socket 44...
the socket is now closed
```

III-G - 06 - Mini serveur : réception d'un bloc de données puis fermeture sur ESC

L'ajout d'une boucle sur recv() permet de recevoir plusieurs caractères. Si on reçoit ESC (27), la boucle est rompue et une demande de deconnexion est émise vers le client. Le serveur s'arrête.

III-G-1 - Pseudo-code

```
BEGIN
    sock := socket()
    bind(sock, ALL_IP, 23)
    listen(sock, 5)
    csock := accept(sock, csin)
    REPEAT
        recv(csock, data)
        send(csock, "OK")
    UNTIL data[0] = ESC
    closesocket(csock)
    closesocket(sock)
END
```

Ce fonctionnement est simple, mais il n'accepte qu'une seule connexion. Dès qu'elle se termine, le serveur s'arrête. Les fonctions accept() et recv() sont bloquantes.

III-G-2 - Code

```
/* 06s.c */
#ifdef __cplusplus
#error Be sure you are using a C compiler...
#endif

#if defined (WIN32) || defined (_WIN32)

#include <winsock2.h>

#elif defined (linux) || defined (_POSIX_VERSION) || defined (_POSIX2_C_VERSION)\
 || defined (_XOPEN_VERSION)

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

```
#include <unistd.h>             /* close */

#define INVALID_SOCKET -1
#define SOCKET_ERROR -1

#define closesocket(s) close (s)
typedef int SOCKET;
typedef struct sockaddr_in SOCKADDR_IN;
typedef struct sockaddr SOCKADDR;


#else
#error not defined for this platform
#endif

#include <stdio.h>
#include <stdlib.h>

/* macros ================================================================ */

#define TELNET 23

/* we want to listen to the TELNET port */
#define PORT TELNET

#define ESC 27

/* constants ============================================================= */
/* types ================================================================= */
/* structures ============================================================ */
/* private data ========================================================== */
/* private functions ===================================================== */

/* ------------------------------------------------------------------------
   ------------------------------------------------------------------------ */
static int app (void)
{
   int err = 0;

   /* open a socket in TCP/IP mode. */
   SOCKET sock = socket (AF_INET, SOCK_STREAM, 0);

   if (sock != INVALID_SOCKET)
   {
      printf ("socket %d is now opened in TCP/IP mode\n", sock);

      {
         int sock_err;
         /* assign the listening port */
         SOCKADDR_IN sin =
         {0};

         /* automatic IP address */
         sin.sin_addr.s_addr = htonl (INADDR_ANY);

         /* protocol family (IP) */
         sin.sin_family = AF_INET;

         /* listening port */
         sin.sin_port = htons (PORT);

         /* bind */
         sock_err = bind (sock, (SOCKADDR *) &sin, sizeof sin);

         if (sock_err != SOCKET_ERROR)
         {
            /* start listening (server mode) */
            sock_err = listen (sock, 5);

            printf ("listening on port %d...\n", PORT);

            if (sock_err != SOCKET_ERROR)
```

```
{
   /* wait for a client connection */
   printf ("waiting for a client connection on port %d...\n", PORT);

   {
      SOCKADDR_IN csin =
      {0};
      int recsize = (int) sizeof csin;
      SOCKET csock = accept (sock, (SOCKADDR *) &csin, &recsize);

      if (csock != INVALID_SOCKET)
      {
         printf ("client connected with socket %d from %s:%d\n"
                 ,csock
                 ,inet_ntoa (csin.sin_addr)
                 ,htons (csin.sin_port));
         {
            int end = 0;
            do
            {
               /* wait for the receive of a data block */
               unsigned char data[128];
               sock_err = recv (csock, data, (sizeof data - 1), 0);

               if (sock_err != SOCKET_ERROR)
               {
                  size_t nb_rec = sock_err;
                  if (nb_rec > 0)
                  {
                     /* convert to string */
                     data[nb_rec] = 0;
                     printf ("%u byte%s received:\n'%s'\n"
                             ,(unsigned) nb_rec
                             ,nb_rec > 1 ? "s" : ""
                             ,data);
                     fflush (stdout);

                     if (data[0] == ESC)
                     {
                        end = 1;
                     }
                     else
                     {
                        /* send some YES-TO-ALL answer */
                        char const response[] = "OK\n";
                        send (csock, response, strlen (response), 0);
                     }
                  }
                  else
                  {
                     puts("client is disconnected");
                     end = 1;
                  }
               }
               else
               {
                  perror ("socket.recv");
                  err = 1;
                  end = 1;
               }
            }
            while (!end);
         }
         shutdown (csock, 2);
         printf ("closing client socket %d...\n", csock);
         closesocket (csock), csock = INVALID_SOCKET;

      }
      else
      {
         perror ("socket.accept");
         err = 1;
```

```
                  }
               }
            }
            else
            {
               perror ("socket.listen");
               err = 1;
            }
         }
         else
         {
            perror ("socket.bind");
            err = 1;
         }

         printf ("closing socket %d...\n", sock);

         /* close the socket. */
         sock_err = closesocket (sock), sock = INVALID_SOCKET;

         printf ("the socket is now closed\n");

         if (sock_err)
         {
            perror ("socket.close");
            err = 1;
         }
      }
   }
   else
   {
      perror ("socket.open");
      err = 1;
   }

   return err;
}

/* entry point ================================================================ */

/* ---------------------------------------------------------------------------
   --------------------------------------------------------------------------- */
int main (void)
{
   int ret;
#if defined (WIN32) || defined (_WIN32)
   WSADATA wsa_data;
   int err = WSAStartup (MAKEWORD (2, 2), &wsa_data);

   if (!err)
   {
      puts ("WIN: winsock2: OK");
#else
   int err = 0;
   {
#endif

      err = app ();

#if defined (WIN32) || defined (_WIN32)
      WSACleanup ();
#else
   }
#endif

   if (err)
   {
      ret = EXIT_FAILURE;
   }
   else
   {
      ret = EXIT_SUCCESS;
```

```
    }

    return ret;
}
```

III-H - 07 - Mini serveur : réception d'un bloc de données puis deconnexion sur ESC

L'ajout d'une boucle sur accept() permet de recevoir plusieurs demandes de connexions à la suite. Le serveur ne s'arrête jamais.

III-H-1 - Pseudo-code

```
BEGIN
   sock := socket()
   bind(sock, ALL_IP, 23)
   listen(sock, 5)
   DO
      csock := accept(sock, csin)
      closesocket(csock)
   FOREVER
   closesocket(sock)
END
```

Afin de simplifier l'organisaton du code, les boucles sont implémentées dans les fonctions clients() et client().

```
FUNCTION client(csock:SOCKET)
BEGIN
   REPEAT
      recv(csock, data)
      send(csock, "OK")
   UNTIL data[0] = ESC
   closesocket(csock)
END

FUNCTION clients(sock:SOCKET)
BEGIN
   DO
      csock := accept(sock, csin)
      client(csock)
   FOREVER
END

BEGIN
   sock := socket()
   bind(sock, ALL_IP, 23)
   listen(sock, 5)
   clients(sock)
   closesocket(sock)
END
```

Ce fonctionnement reste simple, mais il n'accepte qu'une seule connexion à la fois. Par contre, dès qu'elle se termine, le serveur est près à accepter une nouvelle connexion. Les fonctions accept() et recv() sont bloquantes. Si une demande de connexion se fait pendant que le serveur traite un client, la connexion est établie, mais le serveur ne traite pas les données tant que la connexion courante n'est pas terminée. listen(..., 5) permet de traiter jusqu'à 5 demandes simultanées.

III-H-2 - Code

```
/* 7s.c */
#ifdef __cplusplus
#error Be sure you are using a C compiler...
#endif
```

```
#if defined (WIN32) || defined (_WIN32)

#include

#elif defined (linux) || defined (_POSIX_VERSION) || defined (_POSIX2_C_VERSION)\
 || defined (_XOPEN_VERSION)

#include
#include
#include
#include
#include              /* close */

#define INVALID_SOCKET -1
#define SOCKET_ERROR -1

#define closesocket(s) close (s)
typedef int SOCKET;
typedef struct sockaddr_in SOCKADDR_IN;
typedef struct sockaddr SOCKADDR;

#else
#error not defined for this platform
#endif

#include
#include

/* macros ================================================================ */

#define TELNET 23

/* we want to listen to the TELNET port */
#define PORT TELNET

#define ESC 27

/* constants ============================================================= */
/* types ================================================================= */
/* structures ============================================================ */

struct cli
{
   SOCKADDR_IN sin;
   int recsize;
   SOCKET sock;
   int err;
};

/* private data ========================================================== */
/* private functions ===================================================== */

/* thread-like client function */
static void *client (void *p_data)
{
   struct cli *p_cli = p_data;
   if (p_cli != NULL)
   {
      int end = 0;
      do
      {
         /* wait for the receive of a data block */
         unsigned char data[128];
         int sock_err = recv (p_cli->sock, data, (sizeof data - 1), 0);

         if (sock_err != SOCKET_ERROR)
         {
            size_t nb_rec = sock_err;
            if (nb_rec > 0)
            {
               /* convert to string */
               data[nb_rec] = 0;
```

```
                printf ("%u byte%s received:\n'%s'\n",
                        (unsigned) nb_rec, nb_rec > 1 ? "s" : "", data);
                fflush (stdout);

                if (data[0] == ESC)
                {
                   end = 1;
                }
                else
                {
                   /* send some YES-TO-ALL answer */
                   char const response[] = "OK\n";
                   send (p_cli->sock, response, strlen (response), 0);
                }
             }
             else
             {
                puts ("client is disconnected");
                end = 1;
             }
          }
          else
          {
             perror ("socket.recv");
             p_cli->err = 1;
             end = 1;
          }
       }
       while (!end);
       shutdown (p_cli->sock, 2);
       printf ("closing client socket %d...\n", p_cli->sock);
       closesocket (p_cli->sock), p_cli->sock = INVALID_SOCKET;
    }

    return NULL;
}

static int clients (SOCKET sock)
{
    int err = 0;
    int end = 0;
    do
    {
       /* wait for a client connection */
       printf ("waiting for a client connection on port %d...\n", PORT);

       {
          /* create a new client context */
          struct cli *p_cli = malloc (sizeof *p_cli);
          if (p_cli != NULL)
          {
             p_cli->recsize = (int) sizeof p_cli->sin;
             p_cli->sock =
                accept (sock, (SOCKADDR *) & p_cli->sin, &p_cli->recsize);

             if (p_cli->sock != INVALID_SOCKET)
             {
                printf
                   ("client connected with socket %d from %s:%d\n",
                    p_cli->sock, inet_ntoa (p_cli->sin.sin_addr),
                    htons (p_cli->sin.sin_port));

                client (p_cli);

             }
             else
             {
                perror ("socket.accept");
                err = 1;
             }

             /* free the client context */
```

```
            free (p_cli), p_cli = NULL;
         }
         else
         {
            printf ("client creation failed : memory error\n");
         }
      }
   }
   while (!end);
   return err;
}

/* ------------------------------------------------------------------------
   ------------------------------------------------------------------------ */
static int app (void)
{
   int err = 0;

   /* open a socket in TCP/IP mode. */
   SOCKET sock = socket (AF_INET, SOCK_STREAM, 0);

   if (sock != INVALID_SOCKET)
   {
      printf ("socket %d is now opened in TCP/IP mode\n", sock);

      /* we want to listen on the TELNET port */
      {
         int sock_err;
         /* assign the listening port */
         SOCKADDR_IN sin = { 0 };

         /* automatic IP address */
         sin.sin_addr.s_addr = htonl (INADDR_ANY);

         /* protocol family (IP) */
         sin.sin_family = AF_INET;

         /* listening port */
         sin.sin_port = htons (PORT);

         /* bind */
         sock_err = bind (sock, (SOCKADDR *) & sin, sizeof sin);

         if (sock_err != SOCKET_ERROR)
         {
            /* start listening (server mode) */
            sock_err = listen (sock, 5);

            printf ("listening on port %d...\n", PORT);

            if (sock_err != SOCKET_ERROR)
            {
               err = clients (sock);
            }
            else
            {
               perror ("socket.listen");
               err = 1;
            }
         }
         else
         {
            perror ("socket.bind");
            err = 1;
         }

         printf ("closing socket %d...\n", sock);

         /* close the socket. */
         sock_err = closesocket (sock), sock = INVALID_SOCKET;

         printf ("the socket is now closed\n");
```

```
         if (sock_err)
         {
            perror ("socket.close");
            err = 1;
         }
      }
   }
   else
   {
      perror ("socket.open");
      err = 1;
   }

   return err;
}

/* entry point ================================================================ */

/* ---------------------------------------------------------------------------
   --------------------------------------------------------------------------- */
int main (void)
{
   int ret;
#if defined (WIN32) || defined (_WIN32)
   WSADATA wsa_data;
   int err = WSAStartup (MAKEWORD (2, 2), &wsa_data);

   puts ("WIN: winsock2: OK");
#else
   int err = 0;
   {
#endif

      if (!err)
      {
         err = app ();

#if defined (WIN32) || defined (_WIN32)
         WSACleanup ();
#else
      }
#endif
   }
   if (err)
   {
      ret = EXIT_FAILURE;
   }
   else
   {
      ret = EXIT_SUCCESS;
   }

   return ret;
}
```

Pour réaliser une serveur plus conséquent, on peut créer un thread (ou un processus) par client connecté.

III-I - 08 - Serveur-multi-clients : réception d'un bloc de données puis deconnexion sur ESC

En rendant autonome la fonction 'client()' et en la transformant en **thread**, le serveur devient multiclient. En effet, après connexion d'un client, au lieu de rester bloqué sur la réception, le serveur se bloque à nouveau sur l'attente de connexion (accept()). Ca permet de recevoir et de traiter plusieurs demandes de connexions en même temps. Le serveur ne s'arrête jamais.

III-I-1 - Pseudo-code

```
STRUCTURE CLIENT
BEGIN
   thread:PTHREAD
   sock:SOCKET
END

THREAD client(data:GENERIC):GENERIC
BEGIN
   cli := CLIENT(data)
   WITH cli
   BEGIN
      REPEAT
         recv(sock, data)
         send(sock, "OK")
      UNTIL data[0] = ESC
      closesocket(sock)
   END
   client := NIL
END

FUNCTION clients(sock:SOCKET)
BEGIN
   DO
      csock := accept(sock, csin)
      pthread_create(cli.thread, client, cli.csock)
   FOREVER
END

BEGIN
   sock := socket()
   bind(sock, ALL_IP, 23)
   listen(sock, 5)
   clients(sock)
   closesocket(sock)
END
```

Ce fonctionnement reste simple, et il accepte plusieurs connexions à la fois. Les fonctions accept() et recv() sont bloquantes, mais elles sont maintenant dans des threads différents. Dès qu'un client est crée, la fonction 'clients' revient en attente de connexion.

III-I-2 - Code

```
/* 08s.c */
#ifdef __cplusplus
#error Be sure you are using a C compiler...
#endif

#if defined (WIN32) || defined (_WIN32)

#include <winsock2.h>

#elif defined (linux) || defined (_POSIX_VERSION) || defined (_POSIX2_C_VERSION)\
 || defined (_XOPEN_VERSION)

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>             /* close */

#define INVALID_SOCKET -1
#define SOCKET_ERROR -1

#define closesocket(s) close (s)
typedef int SOCKET;
typedef struct sockaddr_in SOCKADDR_IN;
```

```
typedef struct sockaddr SOCKADDR;

#else
#error not defined for this platform
#endif

#include <stdio.h>
#include <stdlib.h>

#include <pthread.h>

/* macros ============================================================ */

#define TELNET 23

/* we want to listen to the TELNET port */
#define PORT TELNET

#define ESC 27

/* constants ========================================================= */
/* types ============================================================= */
/* structures ======================================================== */

struct cli
{
   pthread_t thread;
   SOCKADDR_IN sin;
   int recsize;
   SOCKET sock;
   int err;
};

/* private data ====================================================== */
/* private functions ================================================= */

/* thread client function */
static void *client (void *p_data)
{
   struct cli *p_cli = p_data;
   if (p_cli != NULL)
   {
      int end = 0;
      do
      {
         /* wait for the receive of a data block */
         unsigned char data[128];
         int sock_err = recv (p_cli->sock, data, (sizeof data - 1), 0);

         if (sock_err != SOCKET_ERROR)
         {
            size_t nb_rec = sock_err;
            if (nb_rec > 0)
            {
               /* convert to string */
               data[nb_rec] = 0;
               printf ("%u byte%s received:\n'%s'\n",
                       (unsigned) nb_rec, nb_rec > 1 ? "s" : "", data);
               fflush (stdout);

               if (data[0] == ESC)
               {
                  end = 1;
               }
               else
               {
                  /* send some YES-TO-ALL answer */
                  char const response[] = "OK\n";
                  send (p_cli->sock, response, strlen (response), 0);
               }
            }
            else
```

```
            {
               puts ("client is disconnected");
               end = 1;
            }
         }
         else
         {
            perror ("socket.recv");
            p_cli->err = 1;
            end = 1;
         }
      }
      while (!end);
      shutdown (p_cli->sock, 2);
      printf ("closing client socket %d...\n", p_cli->sock);
      closesocket (p_cli->sock), p_cli->sock = INVALID_SOCKET;

      /* the memory is now under the control of the thread */
      free (p_cli), p_cli = NULL;

   }

   return NULL;
}

static int clients (SOCKET sock)
{
   int err = 0;
   int end = 0;
   do
   {
      /* wait for a client connection */
      printf ("waiting for a client connection on port %d...\n", PORT);

      {
         /* create a new client context */
         struct cli *p_cli = malloc (sizeof *p_cli);
         if (p_cli != NULL)
         {
            p_cli->recsize = (int) sizeof p_cli->sin;
            p_cli->sock =
               accept (sock, (SOCKADDR *) &p_cli->sin, &p_cli->recsize);

            if (p_cli->sock != INVALID_SOCKET)
            {
               printf
                  ("client connected with socket %d from %s:%d\n",
                   p_cli->sock, inet_ntoa (p_cli->sin.sin_addr),
                   htons (p_cli->sin.sin_port));

               /* send ...*/
               pthread_create (&p_cli->thread, NULL, client, p_cli);
               /* ... and forget */
               p_cli = NULL;

            }
            else
            {
               perror ("socket.accept");
               err = 1;
            }
         }
         else
         {
            fprintf (stderr, "client creation failed : memory error\n");
         }
      }
   }
   while (!end);
   return err;
}
```

```
/* ---------------------------------------------------------------------
   --------------------------------------------------------------------- */
static int app (void)
{
   int err = 0;

   /* open a socket in TCP/IP mode. */
   SOCKET sock = socket (AF_INET, SOCK_STREAM, 0);

   if (sock != INVALID_SOCKET)
   {
      printf ("socket %d is now opened in TCP/IP mode\n", sock);

      /* we want to listen on the TELNET port */
      {
         int sock_err;
         /* assign the listening port */
         SOCKADDR_IN sin = { 0 };

         /* automatic IP address */
         sin.sin_addr.s_addr = htonl (INADDR_ANY);

         /* protocol family (IP) */
         sin.sin_family = AF_INET;

         /* listening port */
         sin.sin_port = htons (PORT);

         /* bind */
         sock_err = bind (sock, (SOCKADDR *) &sin, sizeof sin);

         if (sock_err != SOCKET_ERROR)
         {
            /* start listening (server mode) */
            sock_err = listen (sock, 5);

            printf ("listening on port %d...\n", PORT);

            if (sock_err != SOCKET_ERROR)
            {
               err = clients (sock);
            }
            else
            {
               perror ("socket.listen");
               err = 1;
            }
         }
         else
         {
            perror ("socket.bind");
            err = 1;
         }

         printf ("closing socket %d...\n", sock);

         /* close the socket. */
         sock_err = closesocket (sock), sock = INVALID_SOCKET;

         printf ("the socket is now closed\n");

         if (sock_err)
         {
            perror ("socket.close");
            err = 1;
         }
      }
   }
   else
   {
      perror ("socket.open");
      err = 1;
```

```
   }

   return err;
}

/* entry point ================================================================ */

/* ---------------------------------------------------------------------------
   --------------------------------------------------------------------------- */
int main (void)
{
   int ret;
#if defined (WIN32) || defined (_WIN32)
   WSADATA wsa_data;
   int err = WSAStartup (MAKEWORD (2, 2), &wsa_data);

   if (!err)
   {
      puts ("WIN: winsock2: OK");
#else
   int err = 0;
   {
#endif

      err = app ();

#if defined (WIN32) || defined (_WIN32)
      WSACleanup ();
#else
   }
#endif

   if (err)
   {
      ret = EXIT_FAILURE;
   }
   else
   {
      ret = EXIT_SUCCESS;
   }

   system ("pause");

   return ret;
}
```

III-J - Conclusion

Les bases de la réalisation d'un mini serveur Telnet multi-clients sont jetées. Il suffit maintenant de réconstituer les lignes de commandes, de les transmettre à un interpréteur de commande etc.

Exemple de serveur multiclient simple avec client qui demande l'heure courante (now) ou quitte (quit) :

- **serveur**
- **client**

IV - Transmettre du texte par sockets

Un socket permet de transmettre des données 'brutes' sous la forme de blocs d'octets caractérisés dans le code par une adresse et une longueur (les paramètres de send(), par exemple).

Des informations de type texte peuvent évidemment se ramener à une structure adresse, longueur. Mais il convient de le faire proprement et en respectant certaines conventions telles que

- Le jeu de caractères est ASCII
- Pas de 0 en ligne
- Les chaines sont terminées par une marque de fin de ligne telle que CR, LF, CRLF ou autre, définie par protocole utilisé

Le respect de ces conventions permet d'élaborer des fonctions d'émission et reception de texte correctes et efficaces.

la technique utilisée dans ce qui suit n'est valide que sous Windows (winsock2).

IV-A - Emission de petits blocs (cas simple)

Par 'petit blocs', on entend des chaines dont la longueur est inférieure à une trame (quelques centaines d'octets). Le principe est donc de passer l'adresse du premier élément du tableau de char qui contient la chaine, et sa longueur mesurée ici avec strlen().

La fonction send() retourne le nombre d'octets effectivement transmis

```
char *text = "Hello world\n";
int n = send (sock, text, strlen (text), 0);
if (n >= 0)
{
   /* debug */
   printf ("%d octet%s sent\n", n, n != 1 ? "s" : "");
}
```

IV-B - Réception de petits blocs

La réception doit tenir compte du fait qu'en C, les chaines sont terminées par un 0. Il faut donc, non seulement placer ce 0 "à la main" (puisqu'il n'est pas transmis), mais en plus prévoir de la place pour lui dans la chaine de reception.

La fonction recv() retourne le nombre d'octets effectivement reçus. Cette valeur est <= à la taille passée en paramètre. Elle sert à positionner le 0. Une valeur < 0 signale une erreur. 0 signifie 'rien reçu' dans le cas d'un socket non bloquant. Dans le cas d'un socket bloquant, en mode TCP, il signifie 'déconnectée'. Cette valeur ne devrait pas exister en mode UDP bloquant.

```
char text[128]; /* taille arbitraire */
int n = recv (sock, text, sizeof text - 1, 0);
if (n > 0)
{
   text[n] = 0;

   /* debug */
   printf ("received : '%s'\n", text);
}
```

IV-C - Emission de gros blocs (cas général)

Pour émettre un gros bloc, il faut utiliser une technique de découpage si on ne veut pas perdre de données. Etant donné qu'en général on ne sait pas quelle est la valeur maximale d'une trame sur la machine, le mieux est de concevoir du code auto-adaptatif. Pour cela, on utilise une information capitale qui est la valeur retournée par send(). En effet, la valeur retournée est le nombre d'octets effectivement émis.

Par exemple, si la trame maximale fait 1024 octets et qu'on demande à en émettre 2000, send() va retourner 1024. A nous de créer un algorithme qui tient compte de ce résultat et envoie la suite des données.

```
/* non teste */
size_t const len = strlen (text);
size_t sent = 0;

while (sent != len)
{
   int n = send (socket, text + sent, len - sent, 0);

   if (n >=0)
   {
      sent += n;
   }
   else
   {
      /* erreur de transmission */
   }
}
```

Cette technique est générale et couvre évidemment le cas des petits blocs. Il serait intéressant de placer ce code dans une fonction.

IV-D - Réception de gros blocs

La réception assez simple. Il suffit de stocker les blocs reçus et de les passer à une fonction de traitement des données.

Etant donné que l'émission peut avoir découpé les informations en trames plus petites que les informations à transmettre, il faut veiller à ce que les données soient 'réassemblées' sous la forme de lignes cohérentes, c'est à dire terminées par un '\n' si on cherche à interpéter les données reçues.

Les données reçues après un '\n' ne doivent pas être éliminées, mais sont le début de la ligne suivante.

Un mécanisme de réconstitution des lignes doit donc être mis en place en se basant sur la marque de fin de ligne. Si le programme ne fait que router les donnée (par exemple, un serveur de connexion client/client ou une transmission de fichier texte), il pourra se contenter de retransmettre les données inchangées.

IV-E - Etude de cas : transmettre un fichier texte

IV-E-1 - Spécifications

Soit à réaliser un logiciel qui transmet un fichier nommé "fichier.txt" contenant un nombre non spécifié de lignes de textes correctement formées (CR, LF, CRLF ou LFCR).

L'accent est mis sur le protocole et l'integrité des données, et non sur les fonctionnalités qui sont réduites au strict nécessaire.

IV-E-2 - Conception

Le logiciel est composé de deux programmes.

- L'un tourne sur un serveur. Il reçoit la commande de transmission de la part d'un utilisateur et il fourni le fichier à transmettre. C'est le serveur
- L'autre tourne sur une machine utilisateur. Il émet la demande de transmission et reçoit le fichier qu'il stocke localement : c'est le client

IV-E-3 - Protocole

Afin de garantir l'intégrité du transfert, le protocole suivant est établi

- Un message est une ligne de texte. La fin de ligne peut être CR, LF, CRLF ou LFCR
- Le messages peuvent être des commandes ou des réponses
- Le client a l'initiative du transert. C'est lui qui étalit la connexion avec le serveur et qui transmet la commande de récupération de fichier
- Le serveur répond alors "OK" ou "KO", Si la réponse est OK, le transfert peut commencer dès que le client passe une commande d'exécution. Le transfert se termine lorsque la dernière ligne a été transmise. La connexion est alors fermée par le serveur

IV-E-4 - Format des messages

```
message ::= <texte><EOL>
texte ::= sequence of printable characters (ASCII)
EOL ::= CR | LF | CRLF | LFCR
```

IV-E-5 - Commandes

Le client peut passer les commandes suivantes

- get <fichier>
- go

IV-E-6 - Réponses

Le serveur peut émettre les réponses suivantes

- ok
- ko

IV-E-7 - Exemple de dialogue

```
Client                                      Serveur
                ------ connexion ----------->
                ------ "get fichier.txt" --->
                <----- "ok" -----------------
fopen(..., "w")
                ------ "go" ---------------->
                                          fopen(..., "r")
                                          fgets() send()
                <----- donnees --------------
recv()
fputs()
                                          fopen(..., "r")
```

```
                                          fgets() send()
                <----- donnees --------------
recv()
fputs()
                <----- ... ------------------
                                          fclose()
                <----- deconnexion ----------
fclose()
```

Ce protocole basique devrait fonctionner, mais le fait de compter sur la déconnexion pour fermer le fichier en écriture est un peu risqué. Il y a plusieurs façons d'améliorer le protocole :

- Transmettre le nombre de lignes attendues et fermer quand la dernière ligne a été reçue
- Acquitter chaque ligne par un "OK". La dernière est acquittée par un "EOF"

V - Transmettre des données binaires par sockets

La transmission de données binaire doit être 'transparente', c'est à dire qu'on ne doit en aucun cas utiliser les données transmises comme délimiteur. Il n'y a donc pas de marqueur de fin. Il faut donc soit transmettre des données de taille connue (entêtes, par exemple) soit transmettre la taille (dans un en-tête, par exemple), puis les données.

V-A - En-tête

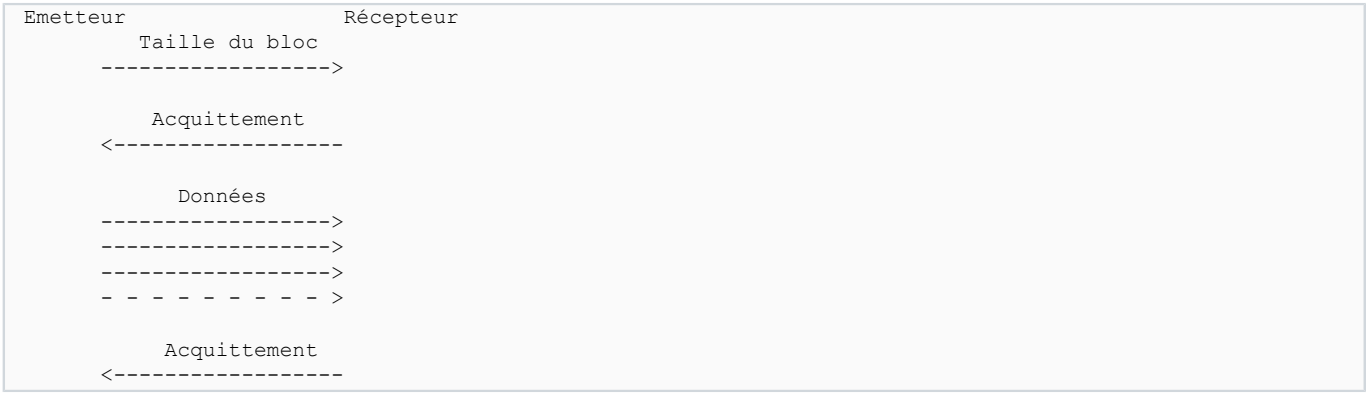
L'en-tête minimum contient la taille du bloc a transmettre. C'est une donnée de taille fixe. Pour la taille, un entier de 32-bit est adapté, codé selon la norma réseau, c'est à dire MSB en tête.

V-B - Protocole

Il est prudent de mettre en place un protocole minimum qui acquitte la transmission de l'entête.

- Taille de l'entête : 4 octets
- Acquittement : 1 octet : 00 = OK 01 = ERREUR

On peut aussi utiliser l'acquittement pour finaliser la réception du bloc de données :



V-C - Mise en oeuvre

Soit à transmettre une bloc de données de 256 octets comprenant les valeurs 0-255 dans l'ordre (ce sera plus simple pour vérifier).

V-C-1 - Transmission de l'entête

Il faut tout d'abord créer l'entête, qui est un entier non signé de 32-bit valant 256. Pour cela, on décorelle la représentation interne de la représentation réseau de manière portable (indépendante de la représentation interne dans la machine) :

```
{
   /* taille a transmettre */
   unsigned long size = 256;

   /* donnees 'taille' effectivement transmises */
   unsigned char data[4];

   data[0] = (size >> (3 * 8)) & 0xFF; /* MSB (3) en tete (0) */
   data[1] = (size >> (2 * 8)) & 0xFF;
   data[2] = (size >> (1 * 8)) & 0xFF;
   data[3] = (size >> (0 * 8)) & 0xFF;
```

```
   /* transmission du bloc : */
   int n = send (sock, data, sizeof data, 0);

   if (n > 0)
   {
      /* transmission reussie */
   }
   else
   {
      /* echec de la transmission */
   }
```

V-C-2 - Réception de l'entête

C'est l'opération inverse. On reçoit un bloc de 4 octets (MSB en tête), et on en déduit la taille :

```
{
   /* donnees 'taille' effectivement reçues */
   unsigned char data[4];

   /* reception du bloc : */
   int n = recv (sock, data, sizeof data, 0);

   if (n == sizeof data)
   {
      /* reception reussie */
      /* taille a recevoir */
      unsigned long size = 0;

      size |= (data[0] << (3 * 8)) ; /* MSB (3) en tete (0) */
      size |= (data[1] << (2 * 8)) ;
      size |= (data[2] << (1 * 8)) ;
      size |= (data[3] << (0 * 8)) ;

      printf ("%lu bytes attendus\n", size);

   }
   else
   {
      /* echec de la reception */
   }
```

V-C-3 - Transmission et réception de l'accquittement

C'est exactement le même principe, sauf pour une taille de 1, on peut simplifier le code ainsi :

```
enum
{
   REP_ACK = 0,
   REP_NAK = 1,
   dummy
};


/* emission */
{
   /* reponse a transmettre */
   int rep = REP_ACK;

   /* donnees effectivement transmises */
   unsigned char data[1];

   data[0] = rep;

   /* transmission du bloc : */
   int n = send (sock, data, sizeof data, 0);
```

```
    if (n > 0)
    {
       /* transmission reussie */
    }
    else
    {
       /* echec de la transmission */
    }
}

/* reception */
{
    /* donnees effectivement reçues */
    unsigned char data[1];

    /* reception du bloc : */
    int n = recv (sock, data, sizeof data, 0);

    if (n == sizeof data)
    {
       /* reception reussie */

       /* reponse a recevoir */
       int rep = data[0];

       switch (rep)
       {
       case REP_ACK:
          puts("ACK");

          /* affichage nombre de bytes ou donnees*/
          break;
       case REP_NAK:
          puts("NAK");
          break;
       default:
          puts("?");
       }
    }
    else
    {
       /* echec de la reception */
    }
```

V-C-4 - Transmission des données

Il faut tout d'abord créer le bloc de données :

```
{
    /* exemple de bloc de donnees */
    unsigned char data[256];

    int i;

    for (i = 0; i < sizeof data; i++)
    {
       data[i] = i;
    }
```

ensuite, on émet le bloc de la manière courante :

```
{
    /* transmission du bloc : */
    int n = send (sock, data, sizeof data, 0);

    if (n > 0)
    {
```

```
      /* transmission reussie */
   }
   else
   {
      /* echec de la transmission */
   }
```

V-C-5 - Réception des données

En s'appuyant sur l'information 'taille' (size) reçue précédemment, on alloue un bloc d'une taille suffisante, puis on réceptionne les données selon la méthode classique :

```
{
   /* donnees effectivement reçues */
   unsigned char *data = malloc (size * sizeof *data);

   if (data != NULL)
   {
      /* reception du bloc : */
      int n = recv (sock, data, size, 0);

      if (n == size)
      {
         /* reception reussie */

         /* affichage de controle */
         int i;
         for (i = 0; i < n; i++)
         {
            printf ("%4u", data[i]);
            if ((i + 1) % 8 == 0)
            {
               printf ("\n");
            }
         }
         printf ("\n");
      }
      else
      {
         /* echec de la reception */
      }
      free (data);
   }
```

Une simulation opérationelle se trouve : **ici**.

```
WIN: winsock2: OK
SIM: Appuyer sur ENTER pour lancer le client
SRV: La socket 88 est maintenant ouverte en mode TCP/IP
SRV: ecoute du port 23...
SRV: Patientez pendant que le client se connecte sur le port 23...

CLI: Connexion a 127.0.0.1 sur le port 23
SRV: Un client se connecte avec la socket 120 de 127.0.0.1:50295
CLI: 256 bytes attendus
SRV: ACK
   0   1   2   3   4   5   6   7
   8   9  10  11  12  13  14  15
  16  17  18  19  20  21  22  23
  24  25  26  27  28  29  30  31
  32  33  34  35  36  37  38  39
  40  41  42  43  44  45  46  47
  48  49  50  51  52  53  54  55
  56  57  58  59  60  61  62  63
  64  65  66  67  68  69  70  71
  72  73  74  75  76  77  78  79
  80  81  82  83  84  85  86  87
  88  89  90  91  92  93  94  95
```

```
  96  97  98  99 100 101 102 103
 104 105 106 107 108 109 110 111
 112 113 114 115 116 117 118 119
 120 121 122 123 124 125 126 127
 128 129 130 131 132 133 134 135
 136 137 138 139 140 141 142 143
 144 145 146 147 148 149 150 151
 152 153 154 155 156 157 158 159
 160 161 162 163 164 165 166 167
 168 169 170 171 172 173 174 175
 176 177 178 179 180 181 182 183
 184 185 186 187 188 189 190 191
 192 193 194 195 196 197 198 199
 200 201 202 203 204 205 206 207
 208 209 210 211 212 213 214 215
 216 217 218 219 220 221 222 223
 224 225 226 227 228 229 230 231
 232 233 234 235 236 237 238 239
 240 241 242 243 244 245 246 247
 248 249 250 251 252 253 254 255

SRV: ACK
SRV: Fermeture du socket client

WIN: winsock2: closed

Process returned 0 (0x0)   execution time : 5.338 s
Press any key to continue.
```

VI - Du bon usage de select()

VI-A - Introduction

La fonction **select()** suspend l'exécution de la tache courante. Celle-ci reprend si au moins une des conditions suivantes est vérifiée :

- Réception sur un flux (sauf Windows) ou un socket
- Fin d'émission sur un flux (sauf Windows) ou un socket
- Réception d'un message d'erreur sur un flux (sauf Windows) ou un socket
- Echéance de temps

L'analyse de la valeur retournée permet en partie d'identifier l'évènement. (-1 = erreur, 0 = echéance, autre = évènement flux)

VI-B - Programmation

Chacune des conditions est programmable individuellement. On utilise pour cela les paramètres de la fonction, qui, il faut le reconnaitre, paraissent un peu étranges au début. En fait tout cela est extrêmement simple et logique :

Le premier paramètre est le plus grand numéro de flux surveillé augmenté de 1.

Le deuxième paramètre est un pointeur sur un objet de type fd_set qui contient la liste des flux entrants surveillés (réception de données)

Le troisième paramètre est un pointeur sur un objet de type fd_set qui contient la liste des flux sortants surveillés (émission de données)

Le quatrième paramètre est un pointeur sur un objet de type fd_set qui contient la liste des flux entrants surveillés (réception d'un message hors bandes : supervision)

Le cinquième parametre est l'adresse d'un objet de type struct timeval qui contient la durée max de la suspension (timeout).

VI-C - Exemples d'utilisation

Ces exemples, un peu scolaires, montrent une possiblité à la fois. Lorsqu'un paramètre n'est pas utilisé, on lui donne la valeur 0 ou NULL.

VI-C-1 - Suspension temporisée

Il suffit de regler la valeur de la temporisation. Attention, pour être portable, celle-ci doit être mise à jour à chaque fois, car il est possible que les valeurs de la structure soient altérées par select().

```
for (;;)
{
   /* set the timer to 1.5 second */
   struct timeval timeout;

   timeout.tv_sec = 1; /* 1 s */
   timeout.tv_usec = 5 * 100 * 1000; /* 500 ms */

   int err = select (0, NULL, NULL, NULL, &timeout);
```

```
      switch (err)
      {
      case 0:
         /* timeout */
         puts ("timeout");
         break;

      case -1:
         /* error */
         puts ("error");
         break;

      default:
         /* stream event */

         /* - (client) connexion has been accepted or data has been received */

         /* - data has been sent */

         /* - an error has been received */
         ;
      }
   }
```

VI-C-2 - Surveillance d'un flux en réception

Cet exemple est purement scolaire, car il n'a pas d'intérêt fonctionnel, étant donné que les fonctions recv() et recvfrom() sont blocantes par défaut. Mais il permet de montrer la syntaxe sur un cas simple. Les macros FD_ZERO() et FD_SET() facilitent la manipulation de l'objet fd_set.

```
   for (;;)
   {
      fd_set readfs;

      FD_ZERO (&readfs);       /* clears readfs */
      FD_SET (sock, &readfs); /* adds a stream */

      int err = select (sock + 1, &readfs, NULL, NULL, NULL);

      switch (err)
      {
      case 0:
         /* timeout */
         break;

      case -1:
         /* error */
         puts ("error");
         break;

      default:
         /* stream event */

         /* - (server) a new client is connected or data has been received */

         if (FD_ISSET (sock, &readfs))
         {
            char data[128];
            int n = recv (sock, data, sizeof data, 0);

            if (n > 0)
            {
               /* process the received data */
            }
         }

         /* - data has been sent */

         /* - an error has been received */
```

```
    }
}
```

VI-C-3 - Surveillance de deux flux en réception

Cet exemple est réel. Il permet de suspendre l'exécution en un endroit précis et unique du logiciel et donc de surveiller la réception de n flux (ici, 2). FD_SETSIZE retourne une valeur correcte mais non optimisée pour le premier paramètre de select().

```
for (;;)
{
   fd_set readfs;

   FD_ZERO (&readfs);      /* clears readfs */
   FD_SET (sock_a, &readfs); /* adds a stream */
   FD_SET (sock_b, &readfs); /* adds another stream */

   int err = select (FD_SETSIZE, &readfs, NULL, NULL, NULL);

   switch (err)
   {
   case 0:
      /* timeout */
      break;

   case -1:
      /* error */
      puts ("error");
      break;

   default:
      /* stream event */

      /* - (server) a new client is connected or data has been received */

      if (FD_ISSET (sock_a, &readfs))
      {
         char data[128];
         int n = recv (sock_a, data, sizeof data, 0);

         if (n > 0)
         {
            /* process the received data */
         }
      }

      if (FD_ISSET (sock_b, &readfs))
      {
         char data[128];
         int n = recv (sock_b, data, sizeof data, 0);

         if (n > 0)
         {
            /* process the received data */
         }
      }

      /* - data has been sent */

      /* - an error has been received */

   }
}
```

L'usage de FD_SETSIZE pour un serveur surveillant quelques sockets peut s'avérer pénalisant. Dans ce cas, il est préférable de s'en tenir à la définition, c'est à dire la valeur max des sockets surveillés augmentée de 1 :

```
int max_sock = MAX (sock_a, sock_b);

select (max_sock + 1, /* ... */);
```

VI-C-4 - Conclusion

Les bases sont posées. Il ne reste plus maintenant qu'à faire jouer son imagination pour combiner les possibilités en fonction des besoins. Au cours de la conception, ne pas oublier les possibilités offertes par les threads.

VII - Recupérer l'adresse IP du serveur

A chaque serveur est attribué une ou plusieurs IP publiques sur le réseau mondial Internet. Pour communiquer avec tel ou tel serveur, l'usage des sockets implique de connaitre une de ces IP publiques. On utilise pour ça la fonction gethostbyname() et les éléments de la structure récupérée (h_*).

```
/* 10.c */

#ifdef __cplusplus
#error Be sure you are using a C compiler...
#endif

#if defined (WIN32) || defined (_WIN32)

#include <winsock2.h>

#elif defined (linux) || defined (_POSIX_VERSION) || defined (_POSIX2_C_VERSION)\
 || defined (_XOPEN_VERSION)

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>             /* close() */
#include <netdb.h>              /* gethostbyname() */

#define INVALID_SOCKET -1
#define SOCKET_ERROR -1

#define closesocket(s) close (s)
typedef int SOCKET;
typedef struct sockaddr_in SOCKADDR_IN;
typedef struct sockaddr SOCKADDR;

#else
#error not defined for this platform
#endif

#include <stdio.h>
#include <stdlib.h>

/* macros ================================================================ */
/* constants ============================================================= */
/* types ================================================================= */
/* structures ============================================================ */
/* private data ========================================================== */
/* private functions ===================================================== */
/* entry point =========================================================== */

/* ------------------------------------------------------------------------
   ------------------------------------------------------------------------ */
int main (void)
{
   int ret;

#if defined (WIN32) || defined (_WIN32)
   WSADATA wsa_data;
   int err = WSAStartup (MAKEWORD (2, 2), &wsa_data);

   if (!err)
   {
      puts ("WIN: winsock2: OK");
 #else
    int err;
 #endif

    {
       struct hostent *p = gethostbyname ("www.google.com");
```

```
      if (p != NULL)
      {
         struct hostent host = *p;
 /* on est tranquille, on a maintenant une copie stable de l'info. */

         printf ("IP de %s\n", host.h_name);

         {
            int i = 0;
            char *ip;

            while ((ip = host.h_addr_list[i]) != NULL)
            {
               int j;
               for (j = 0; j < host.h_length; j++)
               {
                  if (j > 0)
                  {
                     putchar ('.');
                  }
                  printf ("%u", (unsigned) (unsigned char) ip[j]);
               }
               putchar ('\n');

               i++;
            }
         }

      }
    }

#if defined (WIN32) || defined (_WIN32)
      WSACleanup ();
#else
   }
#endif

 if (err)
 {
    ret = EXIT_FAILURE;
 }
 else
 {
    ret = EXIT_SUCCESS;
 }

 return ret;
}
```

Ce qui donne :

```
WIN: winsock2: OK
IP de www.l.google.com
216.239.59.103
216.239.59.104
216.239.59.147
216.239.59.99

Press ENTER to continue.
```

On constate que le véritable nom de serveur de google est "www.l.google.com" et que, vu de chez moi, il possède 4 IP publiques.

VIII - Ebauche d'une bibliothèque sockets portable (psock)

VIII-A - Interfaces communes

psock.h psleep.h

VIII-B - Sockets UNix (sun)

sun.h sun_err.itm sun.c

VIII-C - Sockets IP (Internet Protocol)

inet.h inet_err.itm inet.c

IX - Ressources

Ouvrage de références : **'Unix network programming'** (tome 1) de W. Richard Stevens

- le site d'**IBM**
- le tutoriel rigolo mais bien fourni de **Brian "Beej" Hall**
- l'excellent tutoriel **Les sockets en C** de Benjamin Roux de Developpez.com
- l'excellent logiciel libre multi-plateformes d'analyse de réseau **WireShark** (ex-EtherReal)
- l'excellent site de ressources pour protocoles divers **Ip-Relax**