Table des matières

Introduction

Dans ce monde où augmente toujours les connectivité par réseaux, beaucoup de programmeurs se retrouvent à écrire des programmes qui communiquent par Internet. Comme pour beaucoup de chose, ce n'est pas d'écrire le code qui est difficile, mais de comprendre le concept qu'il y a derrière. Ce petit cours espère vous donner la théorie et la pratique nécéssaire pour mettre rapidement en selle les programmeurs débutants.

Qu'est-ce qu'une socket? (ou : l'analogie)

Il y a un peu plus de 15 ans, l'ARPA (Advanced Research Projects Agency du département de la défense américaine) a assigné à l'Université de Californie à Berkeley la responsabilité de construire un système d'exploitation qui pourrait être utilisé comme plate-forme standard pour l'ARPAnet, le prédécesseur de l'actuel Internet.

Berkeley, déjà très connu pour son travail sur Unix, a ajouté une nouvelle interface au système d'exploitation pour implémenter les communications réseaux. Cette interface est généralement connue sous le nom de *Berkeley Sockets Interface* et est à l'origine de presque tout ce qui existe comme interface pour TCP/IP, et notamment à l'origine des Windows Sockets (*WinSock*).

Une socket ressemble beaucoup à un téléphone : c'est l'extrémité d'un canal de communication bidirectionnel. En connectant deux sockets ensemble, on peut faire passer des données entre processus, même entre processus s'éxécutant sur des machines différentes, exactement de la même façon qu'on parle à travers le téléphone une fois qu'on s'est connecté chez quelqu'un d'autre en l'appelant.

L'analogie du téléphone est excellente et sera utilisée à maintes reprises pour décrire les fonctions des sockets, même si contrairement au téléphone, il y a une distinction à faire entre le programme qui accepte les connexions entrantes et celui qui demande la connexion. Un *serveur* est un programme qui attend les connexions entrantes et qui propose sûrement un certain service à d'autres programmes. Par contre, Un *client* est un prohramme qui se connecte au serveur, généralement pour lui demander de faire quelque chose. Il est important de se souvenir que ce n'est pas le type d'ordinateur qui distingue ce qu'est un client et ce qu'est un serveur, mais la façon dont le programme utilise la socket. Beaucoup de personnes font la confusion...

En attendant l'agent des télécoms (ou : avant de commencer)

Au début de chaque programme qui utilise des sockets, il faut appeler la fonction WinScock WSAStartup() :

```
WSADATA info;
if (WSAStartup(MAKEWORD(1,1), &info) != 0)
   MessageBox(NULL,  "Impossible  d'initialiser  WinSock!",  "WSAStartup",
MB_OK);
```

Le premier argument est le numéro de version de la librairie WinSock que vous utilisez : la version 1.1 est la plus courante, même si la version 2 commence à devenir utilisable. Etant donné que les nouvelles librairies doivent pouvoir utiliser les applications qui utilisent WinSock 1.1, et que peu de programmeurs ont besoin des nouvelles fonctionalités de WinSock 2.0, spécifier 1.1 vous permettra de travailler avec la plupart des librairies disponibles sur le marché.

Si la fonction d'initialisation échoue, on peut obtenir des informations sur l'erreur qui s'est produite en appelant la fonction WSAGetLastError(), qui retourne le code d'erreur correspondant à la cause de l'echec. Il en va de même pour la plupart des fonctions de WinSock.

De même, il faut utiliser la fonction WSACleanup() avant d'arrêter le programme pour quitter proprement la librairie WinSock. Dans les applications Win32, cette appel n'est pas toujours obligatoire mais c'est indispensable dans des applications Win16.

L'annuaire d'Internet (ou : résolution d'adresses)

Comme pour le téléphone, chaque socket a une adresse unique composée de deux éléments : une adresse IP et un numéro de port.

La première partie est l'*adresse IP*, un nombre généralement écrit comme quatre nombres séparés par des points (comme 192.9.200.10), qui spécifie l'ordinateur à qui vous voulez parler. Tous les ordinateurs d'Internet ont une adresse IP.

La seconde partie est le *numéro de port*, qui autorise plusieurs conversations simultanées sur chaque ordinateur. Une application peut soit prendre un numéro de port réservé pour son type d'application, soit en demander un au hasard lorsqu'il lie une adresse à sa socket.

Malheureusement, les nombres sont difficiles à retenir, surtout quand vous devez travailler avec beaucoup de nombres différents. Comme pour le téléphone, un service de recherche

existe pour se souvenir un nom simple (comme guill.net) plutôt que plusieurs nombres (192.74.137.5). L'interface la plus utilisée pour retrouver une adresse est la fonction gethostbyname(), qui prend le nom d'un ordinateur et vous renvoie son adresse IP. De même, il est possible de retrouver le nom d'un ordinateur quand on a son adresse IP en utilisant la fonction gethostbyaddr().

Retournons dans les début d'ARPAnet, quand il y avait seulement quelques centaines d'ordinateurs sur l'ensemble du réseau. Quelques ordinateurs gardaient une liste de tous les ordinateurs, et ces fonctions recherchaient simplement dans un fichier le nom recherché. Maintenant que le réseau s'est agrandit à des dizaines de milliers d'ordinateurs, cette solution ne pouvait plus marcher : les changements permanents demanderaient des mises-à-jour trop fréquentes, et tellement de personnes auraient besoin de ces informations que l'ordinateur qui les garderait serait toujours en surcharge.

La solution à ce problème a été le DNS, *Domain Name Service*. Comme un code postal, le nom d'un hôte DNS est composé de plusieurs parties, en commençant par le domaine le plus haut (comme .com, .fr, .net, .org) et allant de droite à gauche vers l'entité la plus petite, soit le nom de domaine, puis le nom de sous-domaine (si il y en a un), et enfin le nom de l'ordinateur.

L'idée est que si personne ne peut se souvenir de toutes les adresses d'Internet, chacun est capable de retenir sa propre adresse local ainsi que les quelques domaines dans lesquels il évolue. Quand on ne connait pas l'adresse d'un ordinateur, on demande directement à son domaine qui emet une requête jusqu'à ce que quelqu'un lui donne la réponse.

Le résultat est une base de données norme qui est capable de répondre à des millions de demandes différentes sans chercher trop longtemps.

En plus de l'adresse de celui à qui vous voulez parler, vous devez savoir quelle est votre propre adresse. Malheureusement, il n'y a pas moyen de dire "Donne-moi mon adresse", principalement parce qu'il est possible d'avoir plusieurs adresses IP (adresse unicast et multicast). Donc, vous pouvez utiliser la fonction gethostname() pour demander "Quel est mon nom?" et utiliser ensuite la fonction gethostbyname() pour obtenir votre propre adresse IP. The procédé sera rapidement illustré.

Installer son nouveau téléphone (ou : Comment écouter les connexions par socket)

Pour recevoir des appels téléphoniques, il faut d’abord installer votre téléphone. Pour ce faire, il faut créer une socket pour écouter les connections, un procédé qui se passe en plusieurs étapes.

D’abord, il faut créer une socket, ce qui ressemble à se faire installer une ligne de téléphone par la compagnie des télécoms. La fonction socket() est utilisée pour ça.

Comme il y a plusieurs types de sockets, il faut spécifier le type de socket que vous voulez quand vous la créez. Un des arguments est la famille d’adresse utilisée par la socket. Comme le service des postes utilise plusieurs techniques pour délivrer le courrier ou que la compagnie des télécoms utilise plusieurs types de numéros, les sockets peuvent être différentes. La famille d’adresse la plus courante (et la seule disponible avec Winsock 1.1) est le format Internet, spécifié par le nom AF_INET.

Une autre argument qu'il faut remplir est le type de socket. Les deux plus connus sont SOCK_STREAM et SOCK_DGRAM. SOCK_STREAM indique que les données seront transportées comme une chaîne de caractères (TCP), alors que SOCK_DGRAM indique que les donées seront transportées en mode datagramme (UDP).

Nous ne nous intéresserons ici qu'aux sockets SOCK_STREAM, qui sont les plus courantes et les plus faciles à utiliser.

Après avoir créé une socket, il faut lui donner une adresse à écouter, de la même façon qu'on prend un numéro de téléphone pour pouvoir recevoir des appels. La fonction bind() est utilisée pour ça (to bind veut dire « lier », il s'agit ici de lier l'objet socket à une adresse). Une adresse de socket Internet est spécifié en utilisant la structure sockaddr_in, qui contient les champs qui spécifient la famille d'adresse, soit l'adresse IP et le numéro de port pour la socket. Un pointeur vers cette structure est passé en argument aux fonctions qui, comme la fonction bind(), ont besoin d'une adresse. Comme les sockets prétendent gérer plus qu'une famille d'adresse, il faut mettre le pointeur vers la structure sockaddr_in dans un pointeur de structure sockaddr pour éviter les warnings à la compilation.

Les sockets de type SOCK_STREAM ont la capacité de mettre les requêtes de connexion en files d'attentes, ce qui ressemble au téléphone qui sonne en attendant que l'on réponde. Si c'est occupé, la connexion va attendre que vous libériez la ligne. La fonction listen() est utilisée pour donner le nombre maximum de requêtes en attente (généralement jusqu'à 5 maximum) avant de refuser les connexions.

La figure 1 montre comment utiliser les fonctions socket(), gethostname(), gethostbyname(), bind(), et listen() pour mettre en place une socket qui peut accepter les demandes de connexion entrantes.

Après avoir créé une socket pour recevoir des appels, il faut accepter les appels vers cette socket. La fonction accept() est utilisée pour se faire. Appeler la fonction accept() est équivalent à prendre le combiné lorsque le téléphone sonne. Accept() renvoie une nouvelle socket qui est connecté à celui qui appelle.

Un programme qui attend les connexion par socket accept() généralement en boucle et gère toutes les connexions qui arrivent (serveur). Le squelette d'un serveur est donné en figure 2.

Composer un numéro (ou : comment appeler une socket)

Vous savez maintenant comment créer des sockets qui acceptent les appels entrants. Alors, comment on l'appelle? Comme pour le téléphone, il faut d'abord avoir un téléphone avant de l'utiliser. On utilise la fonction socket() exactement de la même façon que pour recevoir les appels.

Après avoir créé une socket, et après lui avoir donné une adresse, il faut utiliser la fonction connect() pour essayer de se connecter à une socket qui attend les appels. La figure 3 illustre une fonction qui créée une socket, qui la prépare et qui appelle un port particulier sur un ordinateur particulier., renvoyant une socket connectée à travers

laquelle les données peuvent passer.

Conversation (ou : comment parler à travers une socket)

Maintenant que vous avez une connexion entre deux sockets, vous voulez envoyer des données entre elles. Les fonctions send() and recv() sont là pour ça.

Contrairement à quand vous lisez ou écrivez un fichier, le réseau peut seulement envoyer ou recevoir un certain volume de données à chaque fois. C'est pourquoi, même si vous demandez beaucoup de caractères d'un coup, vous n'en obtiendrez souvent moins que ce que vous avez demandé. Une façon d'éviter ça est de faire une boucle jusqu'à ce que vous ayez reçu le nombre de caractères que vous vouliez. Une fonction simple permettant de lire un nombre donné de caractères dans un buffer est :

```
int read_data(SOCKET s,  /* connected socket */
              char *buf, /* pointer to the buffer */
              int n      /* number of characters (bytes) we want */
             )
{ int bcount; /* counts bytes read */
  int br;     /* bytes read this pass */

  bcount = 0;
  br = 0;
  while (bcount < n) {             /* loop until full buffer */
    if ((br = recv(s, buf, n - bcount)) > 0) {
      bcount += br;                /* increment byte counter */
      buf += br;                   /* move buffer ptr for next read */
    }
    else if (br < 0)               /* signal an error to the caller */
      return -1;
  }
  return bcount;
}
```

Une fonction très ressemblante devrait envoyer les données : voilà un bon exercice pour le lecteur.

Raccrocher (ou : que faire quand on a fini)

De la même façon que vous raccrichez après avoir eu quelqu'un au téléphone, il faut fermer la connexion entre les deux sockets. La fonction closesocket() est utilisée pour fermer chaque extrémité de la connexion. Si une des extrémité est fermée et que vous essayez d'utiliser la fonction send() à l'autre, la fonction send() renverra une erreur. Un recv() qui attend quand la connection à l'autre extrémité est fermé ne retournera aucun octet.

Parler la langue (ou : l'ordre des octets est important)

Maintenant que vous poucez parler entre ordinateurs, il faut faire attention à ce qu'on dit. Les ordinateurs peuvent utiliser des dialectes différents, comme l'ASCII ou l'EBCDIC, même si c'est devenu assez rare. Plus courament, il y a un problème d'ordre des octets : à moins de ne toujours passer que du texte, vous pouvez avoir des problèmes d'ordre. Heureusement, certains ont déjà trouvé la solution à ce problème. Il était une fois, dans des âges plutôt

sombres, quelqu'un qui decida quel ordre était bon pour les octets... Maintenant, il existe des fonctions qui convertissent de l'une à l'autre si besoin est. Certaines de ces fonctions sont :

- htons() (du type host au type network short integer)
- ntohs() (du type network au type host short integer)
- htonl() (du type host au type network long integer)
- ntohl() (du type network au type host long integer)

Pour ces fonctions, un "short integer" est une entité de 16 bits, un "long integer" est une entité de 32 bits. Avant d'envoyer un entier par une socket, il faut d'abord le passer par la fonction htonl() :

```
i = htonl(i);write_data(s, &i, sizeof(i));
et après avoir lu des données, il faut les reconvertir avec la fonction
ntohl() :
read_data(s, &i, sizeof(i));
i = ntohl(i);
```

Si vous prenez l'habitude d'utiliser ces fonctions, vous aurez assurement moins de problèmes...

Faire compliqué avec quelque chose de simple (ou : les sockets dans les applications)

Si le code que nous avons vu est plutôt simple et facilement compréhensible, il est écrit en mode synchrone, ce qui ne convient pas à la majorité des applications Windows car il faut surveiller l'intéraction de l'utilisateur et attendre que quelque chose arrive sur le réseau.

Comme vous l'expérez sans doute, il est possible d'avoir un message délivré au programme dès que des données attendent dans la socket. La fonction WSAAsyncSelect() est utilisée pour faire en sorte que Windows envoie un message à une fenêtre dès qu'une socket change d'état. L'utilisation de cette fonction est illustrée dans la figure 4, qui fait en sorte que la socket envoie un message à la fenêtre principale dès qu'une donnée est arrivée. Des flags (options) de WSAAsyncSelect() font en sorte que Windows vous prévienne pour autre chose que quand les données sont arrivées, comme le changement d'état de la fonction connect().

En plus, il y a plusieurs versions asynchrones des fonctions des sockets qui attendent normalement que quelque chose se passe, comme WSAAsyncGetHostByName(), une fonction qui va généralement à travers le réseau pour trouver le bon hôte. Consultez la documentation de votre compilateur pour plus d'informations sur ces fonctions.

Le futur est entre vos mains (ou : que faire maintenant)

En utilisant uniquement ce dont on a parlé jusque là, vous devriez pouvoir faire vos propres programmes qui communiquent par sockets. Comme pour tout ce que est nouveau, il serait bon de regarder ce qui a déjà été fait. Heureusement, il existe de nombreux exemples sur WinSock. Recherchez simplement *socket* dans la documentation MSDN et vous trouverez plus d'informations que vous n'en avez jamais lu ,comme par exemple comment créer et utiliser des sockets de type datagramme...

Figure 1, la fonction establish()

```
#include <winsock.h>

/* code to establish a socket
 */
SOCKET establish(unsigned short portnum)
{ char   myname[256];
  SOCKET s;
  struct sockaddr_in sa;
  struct hostent *hp;

  memset(&sa, 0, sizeof(struct sockaddr_in)); /* clear our address */
  gethostname(myname, sizeof(myname));        /* who are we? */
  hp = gethostbyname(myname);                 /* get our address info */
  if (hp == NULL)                             /* we don't exist !? */
    return(INVALID_SOCKET);
   sa.sin_family = hp->h_addrtype;             /* this is our host address
*/
  sa.sin_port = htons(portnum);               /* this is our port number */
  s = socket(AF_INET, SOCK_STREAM, 0);        /* create the socket */
  if (s == INVALID_SOCKET)
    return INVALID_SOCKET;

  /* bind the socket to the internet address */
  if (bind(s, (struct sockaddr *)&sa, sizeof(struct sockaddr_in)) ==
      SOCKET_ERROR) {
    closesocket(s);
    return(INVALID_SOCKET);
  }
   listen(s, 3);                               /* max # of queued connects
*/
  return(s);
}
```

Figure 2, un quelette de serveur

```
#include <winsock.h>

#define PORTNUM 50000 /* random port number, we need something */

void do_something(SOCKET);

main()
{ SOCKET s;

  if ((s = establish(PORTNUM)) == INVALID_SOCKET) { /* plug in the phone */
    perror("establish");
    exit(1);
  }

  for (;;) {                            /* loop for phone calls */
    SOCKET new_sock = accept(s, NULL, NULL);
    if (s == INVALID_SOCKET) {
      fprintf(stderr, "Error waiting for new connection!\n");
      exit(1);
    }
    do_something(new_sock);
    closesocket(new_sock);
```

```
  }
}

/* this is the function that plays with the socket.  it will be called
 * after getting a connection.
 */
void do_something(SOCKET s)
{
  /* do your thing with the socket here
      :
      :
   */
}
```

Figure 3, la fonction call_socket()

```
SOCKET call_socket(const char *hostname, unsigned short portnum)
{ struct sockaddr_in sa;
  struct hostent     *hp;
  SOCKET s;

  hp = gethostbyname(hostname);
  if (hp == NULL) /* we don't know who this host is */
    return INVALID_SOCKET;

  memset(&sa,0,sizeof(sa));
  memcpy((char *)&sa.sin_addr, hp->h_addr, hp->h_length);   /* set address
*/
  sa.sin_family = hp->h_addrtype;
  sa.sin_port = htons((u_short)portnum);

  s = socket(hp->h_addrtype, SOCK_STREAM, 0);
  if (s == INVALID_SOCKET)
    return INVALID_SOCKET;

  /* try to connect to the specified socket */
  if (connect(s, (struct sockaddr *)&sa, sizeof sa) == SOCKET_ERROR) {
    closesocket(s);
    return INVALID_SOCKET;
  }
  return s;
}
```

Figure 4, un squelette de client utilisant des sockets asynchrones

```
#define SOCKET_READY 0x40000 /* special message indicating a socket is
ready */

int WINAPI WinMain(HINSTANCE instance, HINSTANCE prev, LPSTR args, int
show)
{
  HWND app_window;
  WSADATA info;
  SOCKET s;
  MSG msg;

  /* create application's main window */
```

```
  app_window = CreateApplicationWindow(instance, args, show);

  /* initialize the socket library */
  if (WSAStartup(MAKELONG(1, 1), &info) == SOCKET_ERROR) {
    MessageBox(app_window, "Could not initialize socket library.",
               "Startup", MB_OK);
    return 1;
  }

  /* connect to the server */
  s = call_socket("world.std.com", 50000);
  if (s == INVALID_SOCKET) {
    MessageBox(NULL, "Could not connect.", "Connect", MB_OK);
    return 1;
  }

  /* make the socket asynchronous so we get a message whenever there's
   * data waiting on the socket */
  WSAAsyncSelect(s, app_window,  SOCKET_READY, FD_READ);

  /* normal message loop */
  while (GetMessage(&msg, NULL, 0, 0) == TRUE) {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }
  WSACleanup();
}

long WINAPI
MainWindowEventHandler(HWND window,
                       UINT message,
                       WPARAM wparam,
                       LPARAM lparam)
{
  switch (message) {
    /* ... */
  case SOCKET_READY:
  {
    char buf[1024];
    int bytes_read;

    bytes_read = recv(SocketConnection, buf, sizeof(buf));
    if (bytes_read >= 0)
      DoSomethingWithData(buf, bytes_read);
    return 0;
  }
}
```