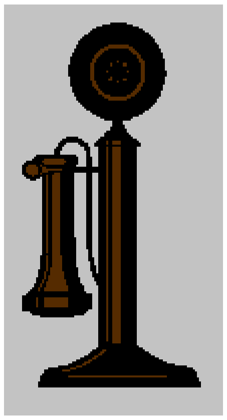


Windows Sockets: A Quick And Dirty Primer

by Jim Frost
Last modified December 31, 1999

Contents

Introduction

In this world of ever-increasing network connectivity more programmers are finding themselves writing software that communicates over the net. As with GUI programming the most difficult part of the task isn't writing the code, it's understanding the concepts behind it. This short primer is intended to provide both the theory and practice necessary to get the novice network programmer up and running fast.

What is a socket, anyway? (or: The Analogy)

A little over fifteen years ago ARPA, the Advanced Research Projects Agency of the Department of Defense, assigned the University of California at Berkeley the responsibility of building an operating system that could be used as the standard platform for supporting the ARPAnet, which was the predecessor of today's Internet.

Berkeley, which was already well-known for their work on UNIX, added a new interface to the operating system to support network communication. This interface is generally known as the Berkeley Sockets Interface and is the basis for almost all existing TCP/IP network protocol interfaces, including Windows Sockets (commonly referred to as WinSock).

A socket is very much like a telephone - it's the endpoint of a two-way communications channel. By connecting two sockets together you can pass data between processes, even processes running on different computers, just as you can talk over the telephone once you've made a phone call connecting your telephone with someone else's.

The telephone analogy is a very good one, and will be used repeatedly to describe socket behavior, but unlike the telephone there is a distinction in the terminology used for a programs which accept incoming connections and those which make connections. A server is a program which waits for incoming connections and presumably provides some service to other programs. In contrast a client is a program which connects to a server, usually to ask it to do something. It is important to remember that it is not the computer that distinguishes what's a client or a server, but the way that the program uses the socket. Most MIS managers believe that "server" means "mainframe" and "client" means "PC." This is not necessarily the case, and has resulted in a lot of confusion as desktop computers often work in both client and server roles simultaneously.

Waiting For The Phone Guy (or: Before you begin)

At the start of every program that uses sockets you must call the WinSock function `WSAStartup()`:

```
WSADATA info;
if (WSAStartup(MAKEWORD(1,1), &info) != 0)
  MessageBox(NULL, "Cannot initialize WinSock!", "WSAStartup", MB_OK);
```

The first argument is the version number of the WinSock library that you expect to be using; version 1.1 is the most popular, although 2.0 is increasingly becoming available. Since newer libraries must support applications which request WinSock 1.1, and very few programmers require the additional features in WinSock 2.0, specifying 1.1 will allow you to work with most of the WinSock libraries on the market.

If the initialization function (or most of the other WinSock functions) fails you can get additional error information by calling `WSAGetLastError()`, which returns an error code indicating the cause of the failure.

Similarly you must call the `WSACleanup()` function before your program exits to properly shut down the library. In Win32 applications this good form although not strictly necessary; it is critical in Win16 or Win32s applications.

The Internet Phone Book (or: Address resolution)

As with the telephone, every socket has a unique address made up of a couple of different parts.

The first part is the IP address, a four-digit number usually written as four decimal numbers separated by periods (e.g. 192.9.200.10), which specifies which computer you want to talk to. Every computer on the Internet has at least one IP address.

The second is the port number, allowing more than one conversation to take place per computer - much like extension numbers in an office. An application may either pick a port number to use (some are, of course, reserved) or request a random port when assigning an address to a socket.

Unfortunately numerics are difficult to remember, particularly when you have to deal with a lot of them on a regular basis. As with the telephone a lookup service exists so that you can remember a simple name (e.g. world.std.com) rather than a set of digits (192.74.137.5). The most commonly used interface to this service is the `gethostbyname()` function, which takes the name of a computer and returns its IP address information. Similarly it's possible to find the name of a computer if you have its number using the `gethostbyaddr()` function.

Back in the early days of the ARPAnet when there were only a few hundred computers on the entire net everyone kept around a list of all the computers, and these functions would simply search the file for the requested name. As the net grew into the tens of thousands of computers this rapidly became unworkable; changes happened too fast to keep a master file up-to-date, and so many people needed the file that the computer that stored it was always overloaded.

The solution to this problem was the Domain Name Service, or DNS. As with a postal address a DNS host name is broken up into several parts which have increasingly finer resolution, starting at the right with the top-level domain (e.g. .com, .edu) and working leftward to the organizational domain (e.g. std, harvard), then on into organizational sub-domains if they exist, and finally to the computer name.

The idea here is that nobody can remember all of the addresses on the Internet, but everyone can remember their own local addresses and a few top-level addresses. When you don't know the address of a computer you're interested in, you can ask the top-level domain and it'll forward the request on to the organizational domain and so on down the chain until someone knows the answer. This process is very similar to using the telephone company's 555-1212 service; dial 555-1212 in the area code you're interested in and provide the name and address information of the person you're looking for and they give you their telephone number.

The result is a huge distributed database which has proven able to support millions of different computers without long delays in looking up the name.

As well as needing to know the address of the computer you want to talk to, you must also know the address of your own computer. Unfortunately there is no way to say "give me my address," in part because it's possible for a single computer to have more than one address. Instead you can use the `gethostname()` function to ask "what's my name," and then use `gethostbyname()` to look up your own address. This process will be illustrated shortly.

Installing Your New Phone (or: How to listen for socket connections)

In order to receive telephone calls, you must first have a telephone installed. Likewise you must create a socket to listen for connections, a process which involves several steps.

First, you must create a socket - which is much like getting a telephone line installed from the phone company. The `socket()` function is used to do this.

Since sockets can have several types, you must specify what type of socket you want when you create one. One option that you have is the address family of a socket. Just as the mail service uses a different scheme to deliver mail than the telephone company uses to complete calls, so can sockets differ. The most common address family (and the only one supported by WinSock 1.1) is the internet format, which is specified by the name `AF_INET`.

Another option which you must supply when creating a socket is the type of socket. The two most common types are `SOCK_STREAM` and `SOCK_DGRAM`. `SOCK_STREAM` indicates that data will come across the socket as a stream of characters, while `SOCK_DGRAM` indicates that data will come in bunches (called datagrams). We will be dealing with `SOCK_STREAM` sockets, which are the most common and easiest to use.

After creating a socket, we must give the socket an address to listen to, just as you get a telephone number so that you can receive calls. The `bind()` function is used to do this (it binds a socket to an address, hence the name). An internet socket address is specified using the `sockaddr_in` structure, which contains fields that specify the address family, address, and port number for a socket. A pointer to this structure is passed to functions like `bind()` which need an address. Because sockets are intended to support more than one address family it's necessary to cast the `sockaddr_in` pointer to a `sockaddr` structure pointer to avoid compiler warnings.

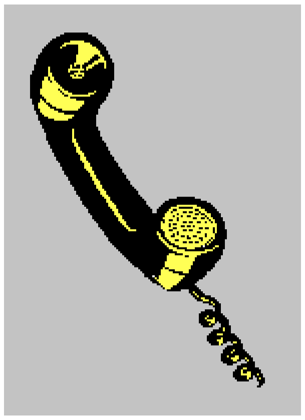
`SOCK_STREAM` type sockets have the ability to queue incoming connection requests, which is a lot like having "call waiting" for your telephone. If you are busy handling a connection, the connection request will wait until you can deal with it. The `listen()` function is used to set the maximum number of requests (up to a maximum of five, usually) that will be queued before requests start being denied.

Figure 1 shows how to use the `socket()`, `gethostname()`, `gethostbyname()`, `bind()`, and `listen()` functions to establish a socket which can accept incoming connections.

After you create a socket to get calls, you must wait for calls to that socket. The `accept()` function is used to do this. Calling `accept()` is analogous to picking up the telephone if it's ringing. `Accept()` returns a new socket which is connected to the caller.

A program which listens for incoming socket connections usually calls `accept()` in a loop and handles each incoming connection. A skeleton framework for a server program is shown in figure 2.

Dialing (or: How to call a socket)



You now know how to create a socket that will accept incoming calls. So how do you call it? As with the telephone, you must first have the phone before using it to call. You use the `socket()` function to do this, exactly as you establish a socket to receive connections.

After creating a socket to make the call with, and giving it an address, you use the connect() function to try to connect to a listening socket. Figure 3 illustrates a function that creates the socket, sets it up, and calls a particular port number on a particular computer, returning a connected socket through which data can flow.

Conversation (or: How to talk between sockets)

Now that you have a connection between sockets you want to send data between them. The `send()` and `recv()` functions are used to do this.

Unlike when you're reading and writing a file, the network can only send or receive a limited amount of data at a time. As a result you can ask for many characters but you'll often get back fewer than you asked for. One way to handle this is to loop until you have received the number of characters that you want. A simple function to read a given number of characters into a buffer is:

```
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;
}
```

A very similar function should be used to send data; we leave that function as an exercise to the reader.

Hanging Up (or: What to do when you're done with a socket)

Just as you hang up when you're through speaking to someone over the telephone, so must you close a connection between sockets. The `closesocket()` function is used to close each end of a socket connection. If one end of a socket is closed and you try to `send()` to the other end, `send()` will return an error. A `recv()` which is waiting when the other end of a socket connection is closed will return zero bytes.

Speaking The Language (or: Byte order is important)

Now that you can talk between computers, you have to be careful what you say. Some computers use differing dialects, such as ASCII versus (yech) EBCDIC, although this has become increasingly unusual. More commonly there are byte-order problems; unless you always pass text, you'll run up against the byte-order problem eventually. Luckily people have already figured out what to do about it. Once upon a time in the dark ages someone decided which byte order was "right." Now there exist functions that convert one to the other if necessary. Some of these functions are:

- `htons()` (host to network short integer)
- `ntohs()` (network to host short integer)
- `htonl()` (host to network long integer)
- `ntohl()` (network to host long integer)

For these functions, a "short integer" is a 16-bit entity, and a "long integer" is a 32-bit entity. Before sending an integer through a socket, you should first massage it with the `htonl()` function:

```
i = htonl(i);
write_data(s, &i, sizeof(i));
```

and after reading data you should convert it back with ntohl():

```
read_data(s, &i, sizeof(i));
i = ntohl(i);
```

If you keep in the habit of using these functions you'll be less likely to goof it up in those circumstances where it is necessary.

Making Simple Stuff Hard (or: Sockets in a windowed application)

While the code we have seen so far is fairly simple and easily understood, it is written in a synchronous model which is inappropriate for most Windows applications, since they must watch for user interaction as well as waiting for something to happen on the network.

As you probably expect, it's possible to get a message delivered to your program whenever there's data waiting on a socket. The function `WSAAsyncSelect()` is used to make Windows send a message to a window whenever a socket changes its status. Its use is illustrated by the application framework seen in figure 4, which sets up the socket to send a message to its main window whenever data is available to be read. Additional flags to `WSAAsyncSelect()` cause Windows to notify you of many other conditions other than "data has arrived," including the status of the `connect()` call.

In addition, there are several asynchronous versions of socket functions which normally must wait for something to happen, including `WSAAsyncGetHostByName()`, a function which usually goes out across the network to find information about the indicated host. You should consult the online documentation for more information about these functions.

The Future Is In Your Hands (or: What to do now)

Using just what's been discussed here, you should be able to build your own programs that communicate with sockets. As with all new things, however, it would be a good idea to look at what's already been done. Luckily there exists a large body of WinSock examples and documentation; simply search for the word socket on the MSDN documentation disk and you'll find more than you'll ever read, including information on creating and using datagram sockets and the multiprotocol enhancements found in WinSock 2.0.

Beware that the examples given here leave out a lot of error checking which should be used in a real application. You should check the documentation for each of the functions discussed here for further information.

Figure 1, the `establish()` function.

```
#include

/* 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, a skeleton server.

```
#include

#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, the `call_socket()` function.

```
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, a skeleton client application using asynchronous sockets.

```
#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;
  }
}
```