

Network Programming Using the Winsock API

This guide is going to provide an introduction into network programming using Microsoft's Winsock API for the Microsoft Windows operating system. Winsock is short for Windows Sockets, it is a specification that defines how Windows network applications should access network services.

History of Winsock

I am going to provide a brief history about Winsock. Winsock is based on **BSD Sockets**, but it provides additional functionality to allow the API to comply with the standard Windows programming model. The Winsock API covered almost all the features found in the **BSD Sockets** API, but there were unavoidable problems which mostly arose from fundamental differences between Windows and Unix.

One of the design goals for Winsock was that it should be relatively easy for developers to port socket-based applications from a Unix platform to a Microsoft Windows platform. The first specification for Winsock was released in June 1992 as Winsock version 1.0. Winsock has been updated in the latest Microsoft Windows operating systems in the Service Packs released.

Using Winsock to Create a Client Connection

First of all I am going to outline the process in which is used to create a client socket that can connect to a network service. Before we start using the Winsock API we need to initialise the Winsock DLL for the current application. The function *WSAStartup(...)* is used for this.

Parameter	Description
WORD wVersionRequested	Highest version of Winsock the caller can use
LPWSADATA lpWSADATA	WSADATA structure that receives the details of the Winsock implementation

So first of all we need to declare a WSADATA structure, then then call *WSAStartup(...)*. Below is an example implementation in code section 1.0.

Code Section 1.0

```
#include <windows.h>
#include <winsock.h>
#include <stdio.h>
#include <string.h>

#pragma comment(lib, "ws2_32.lib")

int main(int argc, char** argv)
{
    WSADATA WSAData;

    WSAStartup(0x202, &WSAData);

    return 0;
}
```

In this example I have used *0x202* as the Winsock version. This means that it will allow Winsock version 2.2 as the highest version the application can use. Next the host name or IP address of the server we wish to connect to is required. The function *gethostbyname(...)* is used to retrieve host information for the corresponding host name from a host database. We are going to hard-code the host name **www.google.com** into the application. A pointer to a *hostent* structure is need to store the information returned by the *gethostbyname(...)* function. Below is the implementation of the *gethostbyname(...)* function into the previous code, shown in code section 2.0.

Code Section 2.0

```

#include <windows.h>
#include <winsock.h>
#include <stdio.h>
#include <string.h>

#pragma comment(lib, "wsock32.lib")

int main(int argc, char** argv)
{
    WSADATA WSAData;
    struct hostent* HostEnt;

    WSASStartup(0x202, &WSAData);

    HostEnt = gethostbyname("www.google.com");

    if (HostEnt == NULL)
    {
        // Handle the error...
        WSACleanup();

        return -1;
    }

    WSACleanup();

    return 0;
}

```

You may see I've used the function *WSACleanup(...)* in the above code, this cleans up Winsock after we have finished using it. We check to see if *HostEnt* is **NULL** after we call the *gethostbyname(...)* function. If the *HostEnt* pointer is **NULL** we know that getting the host information has failed, and we cannot continue.

Next we need to create the socket which we will use to connect to the server, and eventually send and receive data over. A socket variable is declared using the data type *SOCKET*. Then the function *socket(...)* is used to create the socket. Then the *SOCKET* variable is test against *INVALID_SOCKET* to see if the socket creation failed. Below is the socket creation code added, shown in code section 3.0.

Code Section 3.0

```

// Winsock initialisation code...

SOCKET Socket;

Socket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

if (Socket == INVALID_SOCKET)
{
    // Handle the error...
    WSACleanup();

    return -1;
}

```

This creates the socket ready for use. Next we need to fill in a data structure containing the server information. This structure is called *struct sockaddr_in*. It has three members that we need to fill in: The 'family', the server port, and the server IP address.

The family is usually always set to *AF_INET*. The port should be converted to a network-byte address using the *htons(...)* function. Then finally we need to copy the value in the member *h_addr* of the pointer to the *struct hostent* into *sin_addr* of the *struct sockaddr_in* structure. This is shown below in code section 4.0.

Code Section 4.0

```
// Winsock initialisation code...
// Host name information code...
// Socket creation code...

struct sockaddr_in ServerInfo;

memset(&ServerInfo, 0, sizeof(ServerInfo));

memcpy(&ServerInfo.sin_addr, HostEnt->h_addr, HostEnt->h_length);

ServerInfo.sin_family = HostEnt->h_addrtype; // Always AF_INET
ServerInfo.sin_port = htons(80);
```

As you can see we set the port to port 80. This is because the application will try to connect to a web server. The next thing we need to do is to try and connect to the server using the *struct sockaddr_in* structure we declared and initialised. We use the function *connect(...)* to connect to the server. Below in code section 5.0 is the code for connecting to the server.

Code Section 5.0

```
// Winsock initialisation code...
// Host name information code...
// Socket creation code...
// struct sockaddr_in initialisation...

if (connect(Socket, (struct sockaddr*) &ServerInfo, sizeof(ServerInfo)) == -1)
{
    // Handle the error...
    WSACleanup();

    return -1;
}

// Hooray connected to the server!
```

If the call to the *connect(...)* function doesn't return -1, it means that we successfully connected to the server we specified. In the next part of this guide I will show you how you can send and receive data over this socket connection. Below is the full source code for the application we developed to connect to **www.google.com** on port **80**, shown in code section 6.0.

Code Section 6.0

```
#include <windows.h>
#include <winsock.h>
#include <stdio.h>
#include <string.h>

#pragma comment(lib, "wsock32.lib")

int main(int argc, char** argv)
{
    WSADATA WSAData;
    struct hostent* HostEnt;
    SOCKET Socket;
    struct sockaddr_in ServerInfo;

    WSASStartup(0x202, &WSAData);

    HostEnt = gethostbyname("www.google.com");

    if (HostEnt == NULL)
    {
        printf("Unable to get the host information.\n");
    }
}
```

```

    WSACleanup();

    return -1;
}

Socket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

if (Socket == INVALID_SOCKET)
{
    printf("Unable to create the socket.\n");

    WSACleanup();

    return -1;
}

memset(&ServerInfo, 0, sizeof(ServerInfo));

memcpy(&ServerInfo.sin_addr, HostEnt->h_addr, HostEnt->h_length);

ServerInfo.sin_family = HostEnt->h_addrtype; // Always AF_INET
ServerInfo.sin_port = htons(80);

if (connect(Socket, (struct sockaddr*) &ServerInfo, sizeof(ServerInfo)) == -1)
{
    printf("Unable to connect to the server.\n");

    WSACleanup();

    return -1;
}

printf("Successfully connected to the server!\n");

closesocket(Socket);

WSACleanup();

return 0;
}

```

If you compile the above source code and run the application, it should try to connect to **www.google.com** on port **80** and display the *successful* message; if there are any problems appropriate error messages will be displayed. This completes the section on using a socket to make a client connection to a server. Next is how to send and receive data over the socket making the client connection.

Sending data over the socket connection is very simple; we call the *send(...)* function. The parameters are shown in the table below.

Parameter	Description
SOCKET s	This is the socket you wish to send the data over.
const char* buffer	This is a buffer containing the data to be sent.
int length	This is the length of the buffer passed as the second parameter.
int flags	This is a set of flags that specify the way in which the call is made.

The *send(...)* function will either return the amount of bytes sent, or *SOCKET_ERROR* if it fails. An example of sending a HTTP GET request to **www.google.com** using our existing application is shown in code section 7.0.

Code Section 7.0

```

// previous application code...

int iRetVal = 0;

```

```

char szSendBuffer[1024];

ZeroMemory(szSendBuffer, sizeof(szSendBuffer));

Strncpy(szSendBuffer, "GET / HTTP/1.1\r\n\r\n", sizeof(szSendBuffer));

iRetVal = send(Socket, szSendBuffer, sizeof(szSendBuffer));

if (iRetVal != SOCKET_ERROR)
{
    // Successfully sent the data here...
}
else
{
    // Handle error here...
}

```

The above code sends "GET / HTTP/1.1\r\n\r\n" to the server we are connected to. If we were to receive some data back from the server after this call we would get a listing of the HTML from the server. In the next example I will show you how to receive data from the server using the function *recv(...)*. The parameters for this function are shown in the table below.

Parameter	Description
SOCKET s	The socket that the incoming data is received over.
char* buffer	This is a buffer that will be filled with the incoming data.
int length	This is the length of the buffer.
int flags	This is a set of flags that specify the way in which the call is made.

The *recv(...)* functions works in a similar way to how we send data, except the buffer will be filled with the data we receive. Below is an example of receiving data over a socket we're connected with.

Code Section 8.0

```

// previous code...

int iBytesRecv = 0;
char szRecvBuffer[1024];

ZeroMemory(szRecvBuffer, sizeof(szRecvBuffer));

iBytesRecv = recv(Socket, szRecvBuffer, sizeof(szRecvBuffer), 0);

if (iBytesRecv <= 0)
{
    // Nothing received, handle accordingly...
}
else
{
    // received data, handle accordingly...
}

```

This is just a single call to the *recv(...)* function and checks to see if anything was received. This concludes the client-side of the Winsock API. Next I'm going to outline how to use the server-side of the API to show you how to make a simple server that will send "Hello world!" to the clients that connect to it.

Using Winsock to Create a Simple Server

I'm just going to outline the different functions that are needed to create the simple server, and give the code listing at the end. First of all when creating a server we need to create a socket, then we need to *bind* our socket to a specific listening IP address and port, then we need to set our socket to listen for incoming connections.

The first function we need to call is the *socket(...)* function which has already been explained in the client-side bit. The next function we need to make a call to is *bind(...)*. The parameters for this function are explained in the table below.

Parameter	Description
SOCKET s	This is the socket we are binding to the IP and port.
const struct sockaddr_in* name	Address to assign to the socket from the <i>sockaddr</i> structure.
int namelen	The length of the value in the name parameter.

Similarly to the client-side example, we need to fill in a *sockaddr_in** data structure with the IP address and port we wish to listen on. Below in code section 9.0 is the example for filling out the structure to listen on port 8080.

Code Section 9.0

```
// previous code...

struct sockaddr_in ServerInfo;

ZeroMemory(&ServerInfo, sizeof(ServerInfo));

ServerInfo.sin_family = AF_INET;
ServerInfo.sin_addr.s_addr = INADDR_ANY;
ServerInfo.sin_port = htons(8080);

int iRetVal = 0;

iRetVal = bind(Socket, (struct sockaddr*) &ServerInfo, sizeof(struct sockaddr_in));

if (iRetVal != SOCKET_ERROR)
{
    // bound successfully...
}
else
{
    // Handle the error...
}
```

After we have bound the socket to an IP address and port to listen to we have to actually make it listen for incoming connections. To do this we use the function *listen(...)*. Below in the table are the parameters for the function.

Parameter	Description
SOCKET s	This is the socket that is going to be listening.
int backlog	This is the maximum length of the queue of pending connections.

Below in code section 10.0 is the code to set the socket to listen for incoming connections.

Code Section 10.0

```
// previous code...

iRetVal = listen(Socket, 10);
```

```

if (iRetVal != SOCKET_ERROR)
{
    // Listening for incoming connections...
}
else
{
    // Handle the error...
}

```

Once we are finally listening for incoming connections we want to accept the incoming connections and send them our data, "Hello world!" Below in code section 11.0 is the final code listening for the simple server.

Code Section 11.0

```

#include <string.h>
#include <winsock2.h>
#include <windows.h>

#pragma comment(lib, "w32_2.lib");

int main(int argc, char** argv)
{
    WSADATA WsaData;
    SOCKET ServerSocket;
    SOCKET ClientSocket;
    struct sockaddr_in* ServerInfo;
    int iRetVal = 0;
    char* pszSendData = "Hello world!";

    WSASStartup(0x202, &WsaData);

    ServerSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

    if (ServerSocket == INVALID_SOCKET)
    {
        printf("error: unable to create the listening socket...\n");
    }
    else
    {
        ServerInfo->sin_family = AF_INET;
        ServerInfo->sin_addr.s_addr = INADDR_ANY;
        ServerInfo->sin_port = htons(8080);

        iRetVal = bind(ServerSocket, (struct sockaddr*) &ServerInfo, sizeof(struct sockaddr));

        if (iRetVal == SOCKET_ERROR)
        {
            printf("error: unable to bind listening socket...\n");
        }
        else
        {
            iRetVal = listen(ServerSocket, 10);

            if (iRetVal == SOCKET_ERROR)
            {
                printf("error: unable to listen on listening socket...\n");
            }
            else
            {
                while (true)
                {
                    ClientSocket = accept(ServerSocket, NULL, NULL);

                    printf("Incoming connection accepted!\n");
                    send(ClientSocket, pszSendData, strlen(pszData), 0);

                    closesocket(ClientSocket);
                }
            }
        }
    }
}

```

```
closesocket (ServerSocket);  
  
WSACleanup();  
  
return 0;  
}
```

In the final example, you run the simple server and then client connections can be accepted. To test you can telnet to the server and it should send "Hello world!" back to you. Use the following command from a command prompt, *telnet localhost 8080*. Hopefully it should send the data and also on the server window should display the message "Incoming client connection accepted".

This concludes the guide for using the Winsock API. If you would like to find out more information you can visit <http://msdn2.microsoft.com/en-us/library/ms740673.aspx> which describes the Winsock API fully and gives a few examples.