Getting Started with Winsock

The following is a step-by-step guide to getting started with Windows Sockets programming. It is designed to provide an understanding of basic Winsock functions and data structures, and how they work together.

About Servers and Clients

There are two distinct types of socket network applications: Server and Client. Servers and Clients have different behaviors; therefore, the process of creating them is different. What follows is the general model for creating a streaming TCP/IP Server and Client.

Server

1. Initialize WSA.
2. Create a socket.
3. Bind the socket.
4. Listen on the socket.
5. Accept a connection.
6. Send and receive data.
7. Disconnect.

Client

1. Initialize WSA.
2. Create a socket.
3. Connect to the server.
4. Send and receive data.
5. Disconnect.

Note Some of the steps are the same. These steps are implemented almost exactly alike. The steps in this guide will be specific to the type of application being created.

First Step: Creating a Basic Winsock Application

Creating a Basic Winsock Application

To create a basic Winsock application

1. Create a new empty project.
2. Add an empty C++ source file to the project.
3. Ensure that the build environment refers to the Include, Lib, and Src directories of the Microsoft Platform SDK.
4. Ensure that the build environment links to the Winsock Library file WS2_32.lib.
5. Begin programming the Winsock application. Use the Winsock API by including the Winsock 2 header file.

   Note Stdio.h is used for standard input and output, specifically the **printf()** function.

```
#include <stdio.h>
#include "winsock2.h"

void main() {
  return;
}
```

Next Step: Initializing Winsock

Initializing Winsock

All Winsock applications must be initialized to ensure that Windows sockets are supported on the system.

To initialize Winsock

1. Create a **WSADATA** object called wsaData.

```
WSADATA wsaData;
```

2. Call **WSAStartup** and return its value as an integer and check for errors.

```
int iResult = WSAStartup( MAKEWORD(2,2), &wsaData );
if ( iResult != NO_ERROR )
    printf("Error at WSAStartup()\n");
```

The **WSAStartup** function is called to initiate use of WS2_32.lib.

The **WSADATA** structure contains information about the Windows Sockets implementation. The MAKEWORD (2,2) parameter of **WSAStartup** makes a request for the version of Winsock on the system, and sets the passed version as the highest version of Windows Sockets support that the caller can use.

Next Step: Creating a Socket

Creating a Socket

After initialization, a **SOCKET** object must be instantiated.

To create a socket

1. Create a **SOCKET** object called m_socket.

```
SOCKET m_socket;
```

2. Call the **socket** function and return its value to the m_socket variable. For this application, use the Internet address family, streaming sockets, and the TCP/IP protocol.

```
m_socket = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP );
```

3. Check for errors to ensure that the socket is a valid socket.

```
if ( m_socket == INVALID_SOCKET ) {
    printf( "Error at socket(): %ld\n", WSAGetLastError() );
    WSACleanup();
    return;
}
```

The parameters passed to the **socket** function can be changed for different implementations.

Error detection is a key part of successful networking code. If the **socket** call fails, it returns INVALID_SOCKET. The **if** statement in the previous code is used to catch any errors that may have occurred while creating the socket. **WSAGetLastError** returns an error number associated with the last error that occurred.

Note More extensive error checking may be necessary depending on the application.

WSACleanup is used to terminate the use of the WS2_32 DLL.

Server Next Step: Binding a Socket

Client Next Step: Connecting to a Socket

Binding a Socket

For a server to accept client connections, it must be bound to a network address within the system. The following code demonstrates how to bind a socket that has already been created to an IP address and port. Client applications use the IP address and port to connect to the host network.

The **sockaddr** structure holds information regarding the address family, IP address, and port number. **sockaddr_in** is a subset of **sockaddr** and is used for IP version 4 applications.

To bind a socket

1. Create a **sockaddr_in** object and set its values.

```
sockaddr_in service;

service.sin_family = AF_INET;
service.sin_addr.s_addr = inet_addr( "127.0.0.1" );
service.sin_port = htons( 27015 );
```

2. Call the **bind** function, passing the created **socket** and the **sockaddr_in** structure as parameters. Check for general errors.

```
if ( bind( m_socket, (SOCKADDR*) &service, sizeof(service) ) == SOCKET_ERROR ) {
    printf( "bind() failed.\n" );
    closesocket(m_socket);
    return;
}
```

The three lines following the declaration of **sockaddr_in** *service* are used to set up the **sockaddr** structure:

- AF_INET is the Internet address family.
- "127.0.0.1" is the local IP address to which the socket will be bound.
- 27015 is the port number to which the socket will be bound.

Next Step: Listening on a Socket

Listening on a Socket

After the socket is bound to an IP address and port on the system, the server must then listen on that IP address and port for incoming connection requests.

To listen on a socket

- Call the **listen** function, passing the created socket and the maximum number of allowed connections to accept as parameters. Check for general errors.

```
if ( listen( m_socket, 1 ) == SOCKET_ERROR )
    printf( "Error listening on socket.\n");
```

Next Step: Accepting a Connection

Accepting a Connection

Once the socket is listening for a connection, the program must handle connection requests on that socket.

To accept a connection on a socket

1. Create a temporary **SOCKET** object called AcceptSocket for accepting connections.

```
SOCKET AcceptSocket;
```

2. Create a continuous loop that checks for connections requests. If a connection request occurs, call the **accept** function to handle the request.

```
printf( "Waiting for a client to connect...\n" );
while (1) {
    AcceptSocket = SOCKET_ERROR;
    while ( AcceptSocket == SOCKET_ERROR ) {
        AcceptSocket = accept( m_socket, NULL, NULL );
    }
```

3. When the client connection has been accepted, transfer control from the temporary socket to the original socket and stop checking for new connections.

```
    printf( "Client Connected.\n");
    m_socket = AcceptSocket;
    break;
}
```

Next Step: Connecting to a Socket

Connecting to a Socket

For a client to communicate on a network, it must connect to a server.

To connect to a socket

1. Create a **sockaddr_in** object clientService and set its values.

```
sockaddr_in clientService;

clientService.sin_family = AF_INET;
clientService.sin_addr.s_addr = inet_addr( "127.0.0.1" );
clientService.sin_port = htons( 27015 );
```

2. Call the **connect** function, passing the created socket and the **sockaddr_in** structure as parameters. Check for general errors.

```
if ( connect( m_socket, (SOCKADDR*) &clientService, sizeof(clientService) ) == SOCKET_E
    printf( "Failed to connect.\n" );
    WSACleanup();
    return;
}
```

The three lines following the declaration of **sockaddr_in** *clientService* are used to set up the **sockaddr** structure:

- AF_INET is the Internet address family.
- "127.0.0.1" is the remote IP address of the server that the client will connect to.
- 27015 is the port number associated with the server that the client will connect to.

Next Step: Sending and Receiving Data

Sending and Receiving Data

The following code demonstrates the **send** and **recv** functions.

Server

```
int bytesSent;
int bytesRecv = SOCKET_ERROR;
char sendbuf[32] = "Server: Sending Data.";
char recvbuf[32] = "";

bytesRecv = recv( m_socket, recvbuf, 32, 0 );
printf( "Bytes Recv: %ld\n", bytesRecv );

bytesSent = send( m_socket, sendbuf, strlen(sendbuf), 0 );
printf( "Bytes Sent: %ld\n", bytesSent );
```

Client

```
int bytesSent;
int bytesRecv = SOCKET_ERROR;
char sendbuf[32] = "Client: Sending data.";
char recvbuf[32] = "";

bytesSent = send( m_socket, sendbuf, strlen(sendbuf), 0 );
printf( "Bytes Sent: %ld\n", bytesSent );

while( bytesRecv == SOCKET_ERROR ) {
    bytesRecv = recv( m_socket, recvbuf, 32, 0 );
    if ( bytesRecv == 0 || bytesRecv == WSAECONNRESET ) {
        printf( "Connection Closed.\n");
        break;
    }
    if (bytesRecv < 0)
        return;
    printf( "Bytes Recv: %ld\n", bytesRecv );
}
```

In this code, two integers are used to keep track of the number of bytes that are sent and received. The **send** and **recv** functions both return an integer value of the number of bytes sent or received, respectively, or an error. Each function also takes the same parameters: the active socket, a **char** buffer, the number of bytes to send or receive, and any flags to use.

Complete Source Code

- Complete Server Code
- Complete Client Code

Complete Server Code

The following is the complete source code for the TCP/IP Server application.

Server Source Code

```
#include <stdio.h>
#include "winsock2.h"

void main() {

    // Initialize Winsock.
    WSADATA wsaData;
    int iResult = WSAStartup( MAKEWORD(2,2), &wsaData );
    if ( iResult != NO_ERROR )
        printf("Error at WSAStartup()\n");

    // Create a socket.
    SOCKET m_socket;
    m_socket = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP );

    if ( m_socket == INVALID_SOCKET ) {
        printf( "Error at socket(): %ld\n", WSAGetLastError() );
        WSACleanup();
        return;
    }

    // Bind the socket.
    sockaddr_in service;

    service.sin_family = AF_INET;
    service.sin_addr.s_addr = inet_addr( "127.0.0.1" );
    service.sin_port = htons( 27015 );

    if ( bind( m_socket, (SOCKADDR*) &service, sizeof(service) ) == SOCKET_ERROR ) {
        printf( "bind() failed.\n" );
        closesocket(m_socket);
        return;
    }

    // Listen on the socket.
    if ( listen( m_socket, 1 ) == SOCKET_ERROR )
        printf( "Error listening on socket.\n");

    // Accept connections.
    SOCKET AcceptSocket;

    printf( "Waiting for a client to connect...\n" );
    while (1) {
        AcceptSocket = SOCKET_ERROR;
        while ( AcceptSocket == SOCKET_ERROR ) {
            AcceptSocket = accept( m_socket, NULL, NULL );
        }
        printf( "Client Connected.\n");
        m_socket = AcceptSocket;
        break;
    }

    // Send and receive data.
    int bytesSent;
    int bytesRecv = SOCKET_ERROR;
    char sendbuf[32] = "Server: Sending Data.";
    char recvbuf[32] = "";

    bytesRecv = recv( m_socket, recvbuf, 32, 0 );
    printf( "Bytes Recv: %ld\n", bytesRecv );
```

```
    bytesSent = send( m_socket, sendbuf, strlen(sendbuf), 0 );
    printf( "Bytes Sent: %ld\n", bytesSent );

    return;
}
```

Complete Client Code

The following is the complete source code for the TCP/IP Client Application.

Client Source Code

```
#include <stdio.h>
#include "winsock2.h"

void main() {

    // Initialize Winsock.
    WSADATA wsaData;
    int iResult = WSAStartup( MAKEWORD(2,2), &wsaData );
    if ( iResult != NO_ERROR )
        printf("Error at WSAStartup()\n");

    // Create a socket.
    SOCKET m_socket;
    m_socket = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP );

    if ( m_socket == INVALID_SOCKET ) {
        printf( "Error at socket(): %ld\n", WSAGetLastError() );
        WSACleanup();
        return;
    }

    // Connect to a server.
    sockaddr_in clientService;

    clientService.sin_family = AF_INET;
    clientService.sin_addr.s_addr = inet_addr( "127.0.0.1" );
    clientService.sin_port = htons( 27015 );

    if ( connect( m_socket, (SOCKADDR*) &clientService, sizeof(clientService) ) == SOCKET_ERR
        printf( "Failed to connect.\n" );
        WSACleanup();
        return;
    }

    // Send and receive data.
    int bytesSent;
    int bytesRecv = SOCKET_ERROR;
    char sendbuf[32] = "Client: Sending data.";
    char recvbuf[32] = "";

    bytesSent = send( m_socket, sendbuf, strlen(sendbuf), 0 );
    printf( "Bytes Sent: %ld\n", bytesSent );

    while( bytesRecv == SOCKET_ERROR ) {
        bytesRecv = recv( m_socket, recvbuf, 32, 0 );
        if ( bytesRecv == 0 || bytesRecv == WSAECONNRESET ) {
            printf( "Connection Closed.\n");
            break;
        }
        if (bytesRecv < 0)
            return;
        printf( "Bytes Recv: %ld\n", bytesRecv );
    }

    return;
}
```