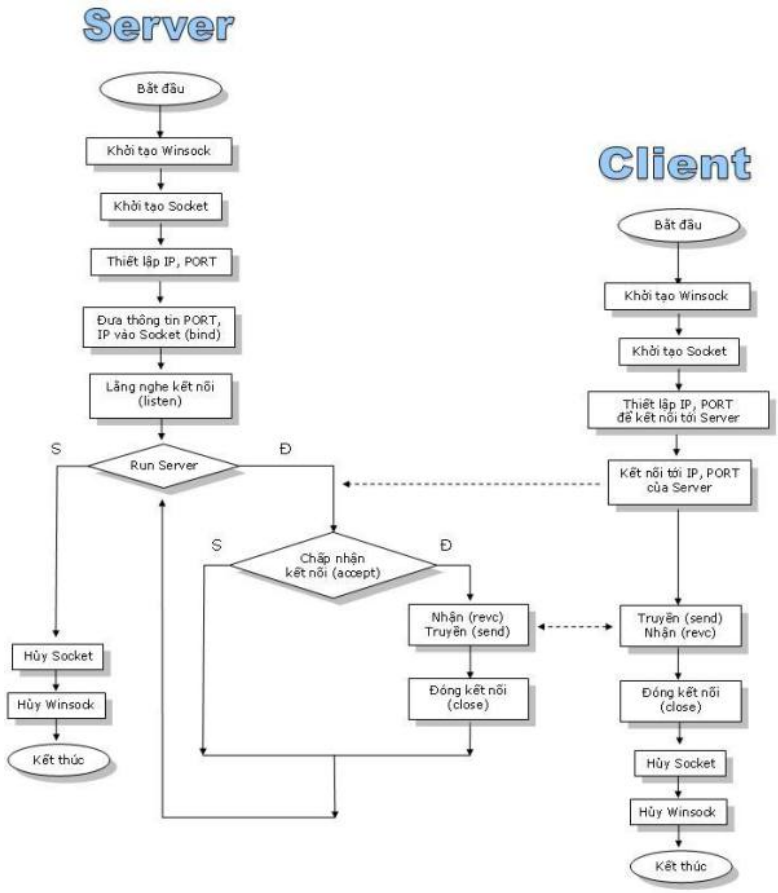
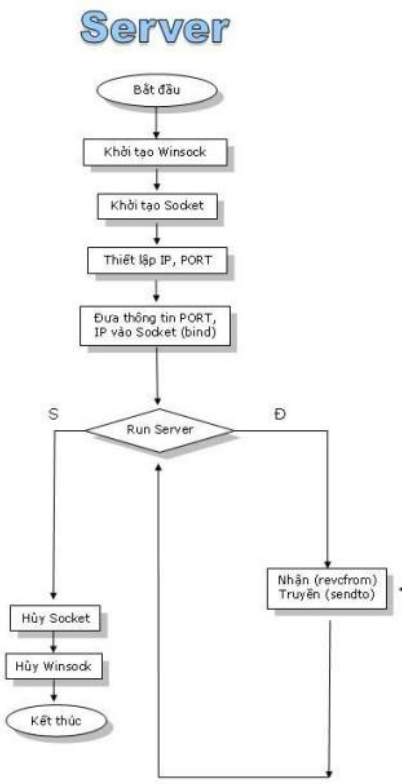
TCP/IP SOCKET PROGRAMMING

1. Introduction



Hoạt động của Client - Server trong giao thức TCP (SOCK_STREAM)



Hoạt động của Client - Server trong giao thức

2. TCP/IP and Socket Programming

Berkeley Sockets provides the foundation for many of the implementations of TCP/IP programs. This includes Windows programs. We will start out with some simple programs that have simple user interfaces. The next several programs that we are going to put together look very similar iin Unix and in Windows. These programs are using the basic Berkeley Socket concepts. Consequently, we are going to start out with a **Console Interface** approach. We will later migrate these programs to more GUI style user interfaces.

2.1. Simple Client TCP/IP Access

The following code can be used to make a simple Client TCP/IP request. To create this program, use the following steps:

Greate a Console Application
Greate a new CPP source file
Gopy the following source and paste it into the new CPP source file
Go to **Project/Settings** and pick the Link tab In the Project Options add : **wsock32.lib**

```
//
// Client.cpp
//
// Extremely simple, stream client example.
// Works in conjunction with Server.cpp.
//
// The program attempts to connect to the server and port
// specified on the command line. The Server program prints
// the needed information when it is started. Once connected,
```

```
// the program sends data to the server, waits for a response
// and then exits.
//
// Compile and link with wsock32.lib
//
// Pass the server name and port number on the command line.
//
// Example: Client MyMachineName 2000
//
#include <stdio.h>
#include <winsock.h>

// Function prototype
void StreamClient(char *szServer, short nPort);

// Helper macro for displaying errors
#define PRINTERROR(s)  \
               fprintf(stderr,"\n%: %d\n", s, WSAGetLastError())

////////////////////////////////////////////////////////////

void main(int argc, char **argv)
{
        WORD wVersionRequested = MAKEWORD(1,1);
        WSADATA wsaData;
        int nRet;
        short nPort;

        //
        // Check for the host and port arguments
        //
        if (argc != 3)
        {
               fprintf(stderr,"\nSyntax: client ServerName PortNumber\n");
               return;
        }

        nPort = atoi(argv[2]);


        //
        // Initialize WinSock and check the version
        //
        nRet = WSAStartup(wVersionRequested, &wsaData);
        if (wsaData.wVersion != wVersionRequested)
        {
               fprintf(stderr,"\n Wrong version\n");
               return;
        }


        //
        // Go do the stuff a stream client does
        //
        StreamClient(argv[1], nPort);


        //
        // Release WinSock
        //
        WSACleanup();
}

////////////////////////////////////////////////////////////

void StreamClient(char *szServer, short nPort)
{
        printf("\nStream Client connecting to server: %s on port: %d",
                                szServer, nPort);

        //
        // Find the server
        //
    LPHOSTENT lpHostEntry;

        lpHostEntry = gethostbyname(szServer);
    if (lpHostEntry == NULL)
    {
        PRINTERROR("gethostbyname()");
        return;
    }

        //
        // Create a TCP/IP stream socket
        //
        SOCKET  theSocket;
```

```
        theSocket = socket(AF_INET,// Address family
                SOCK_STREAM,// Socket type
                IPPROTO_TCP);           // Protocol
        if (theSocket == INVALID_SOCKET)
        {
                PRINTERROR("socket()");
                return;
        }

        //
        // Fill in the address structure
        //
        SOCKADDR_IN saServer;

        saServer.sin_family = AF_INET;

        // Server's address
        saServer.sin_addr = *((LPIN_ADDR)*lpHostEntry->h_addr_list);
        saServer.sin_port = htons(nPort);     // Port number from command line

        //
        // connect to the server
        //
        int nRet;

        nRet = connect(theSocket,       // Socket
                (LPSOCKADDR)&saServer, // Server address
                sizeof(struct sockaddr));// Length of server address structure
        if (nRet == SOCKET_ERROR)
        {
                PRINTERROR("socket()");
                closesocket(theSocket);
                return;
        }


        //
        // Send data to the server
        //
        char szBuf[256];

        strcpy(szBuf, "From the Client");
        nRet = send(theSocket,// Connected socket
                szBuf,  // Data buffer
                strlen(szBuf), // Length of data
                0);     // Flags
        if (nRet == SOCKET_ERROR)
        {
                PRINTERROR("send()");
                closesocket(theSocket);
                return;
        }


        //
        // Wait for a reply
        //
        nRet = recv(theSocket, // Connected socket
                szBuf,  // Receive buffer
                sizeof(szBuf), // Size of receive buffer
                0);     // Flags
        if (nRet == SOCKET_ERROR)
        {
                PRINTERROR("recv()");
                closesocket(theSocket);
                return;
        }


        //
        // Display the received data
        //

        szBuf[nRet]=0;
        printf("\nData received: %s", szBuf);


        closesocket(theSocket);
        return;
}
```

2.2. Simple Server TCP/IP Program

The following code provides the Server end of the Simple TCP/IP program. To create this program, use the following steps:

Greate a Console Application

Greate a new CPP source file

Gopy the following source and paste it into the new CPP source file

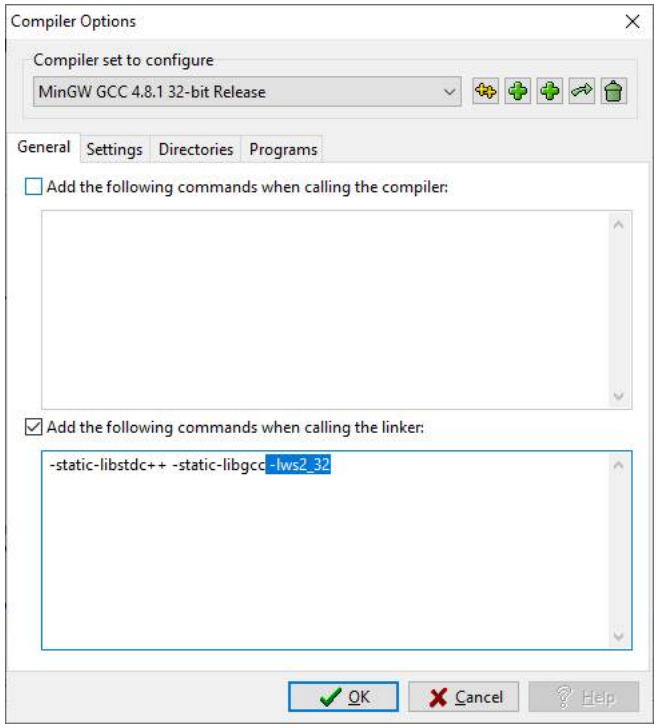
Go to **Project/Settings** and pick the Link tab In the Project Options add : **wsock32.lib**

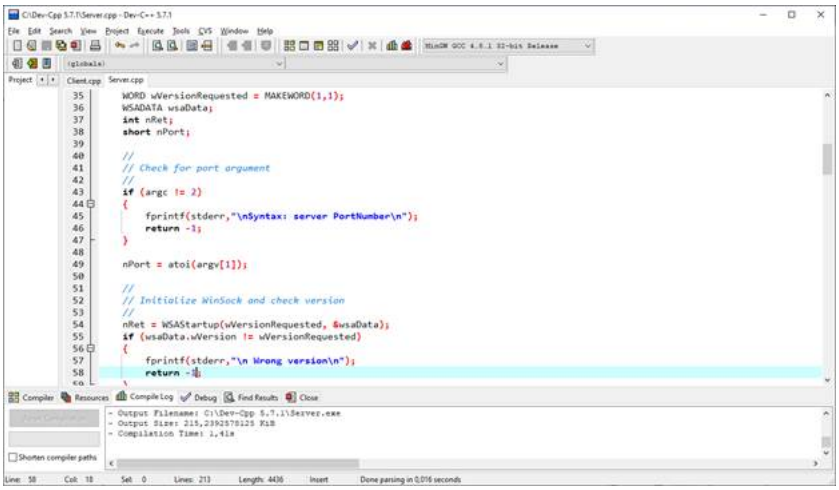
```
//
// Server.cpp
//
// Extremely simple, stream server example.
// Works in conjunction with Client.cpp.
//
// The program sets itself up as a server using the TCP
// protoocl. It waits for data from a client, displays
// the incoming data, sends a message back to the client
// and then exits.
//
// Compile and link with wsock32.lib
//
// Pass the port number that the server should bind() to
// on the command line. Any port number not already in use
// can be specified.
//
// Example: Server 2000
//

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

// Function prototype
void StreamServer(short nPort);

// Helper macro for displaying errors
#define PRINTERROR(s)   \
               fprintf(stderr,"\n%: %d\n", s, WSAGetLastError())

////////////////////////////////////////////////////////////

void main(int argc, char **argv)
{
        WORD wVersionRequested = MAKEWORD(1,1);
        WSADATA wsaData;
        int nRet;
        short nPort;

        //
        // Check for port argument
        //
        if (argc != 2)
        {
               fprintf(stderr,"\nSyntax: server PortNumber\n");
               return;
        }

        nPort = atoi(argv[1]);

        //
        // Initialize WinSock and check version
        //
        nRet = WSAStartup(wVersionRequested, &wsaData);
        if (wsaData.wVersion != wVersionRequested)
        {
               fprintf(stderr,"\n Wrong version\n");
               return;
        }


        //
        // Do the stuff a stream server does
        //
        StreamServer(nPort);


        //
        // Release WinSock
        //
        WSACleanup();
}

////////////////////////////////////////////////////////////

void StreamServer(short nPort)
{
        //
```

```
// Create a TCP/IP stream socket to "listen" with
//
SOCKET  listenSocket;

listenSocket = socket(AF_INET,                  // Address family
                                        SOCK_STREAM,          // Socket type
                                        IPPROTO_TCP);         // Protocol
if (listenSocket == INVALID_SOCKET)
{
       PRINTERROR("socket()");
       return;
}


//
// Fill in the address structure
//
SOCKADDR_IN saServer;

saServer.sin_family = AF_INET;
saServer.sin_addr.s_addr = INADDR_ANY;         // Let WinSock supply address
saServer.sin_port = htons(nPort);              // Use port from command line

//
// bind the name to the socket
//
int nRet;

nRet = bind(listenSocket,      // Socket
       (LPSOCKADDR)&saServer, // Our address
       sizeof(struct sockaddr));// Size of address structure
if (nRet == SOCKET_ERROR)
{
       PRINTERROR("bind()");
       closesocket(listenSocket);
       return;
}

//
// This isn't normally done or required, but in this
// example we're printing out where the server is waiting
// so that you can connect the example client.
//
int nLen;
nLen = sizeof(SOCKADDR);
char szBuf[256];

nRet = gethostname(szBuf, sizeof(szBuf));
if (nRet == SOCKET_ERROR)
{
       PRINTERROR("gethostname()");
       closesocket(listenSocket);
       return;
}

//
// Show the server name and port number
//
printf("\nServer named %s waiting on port %d\n",
               szBuf, nPort);

//
// Set the socket to listen
//

printf("\nlisten()");
nRet = listen(listenSocket,    // Bound socket
       SOMAXCONN);    // Number of connection request queue
if (nRet == SOCKET_ERROR)
{
       PRINTERROR("listen()");
       closesocket(listenSocket);
       return;
}

//
// Wait for an incoming request
//
SOCKET  remoteSocket;

printf("\nBlocking at accept()");
remoteSocket = accept(listenSocket,  // Listening socket
       NULL,   // Optional client address
       NULL);
if (remoteSocket == INVALID_SOCKET)
{
       PRINTERROR("accept()");
```

```
            closesocket(listenSocket);
            return;
        }

        //
        // We're connected to a client
        // New socket descriptor returned already
        // has clients address

        //
        // Receive data from the client
        //
        memset(szBuf, 0, sizeof(szBuf));
        nRet = recv(remoteSocket,                                   // Connected client
                            szBuf,                                          // Receive buffer
                            sizeof(szBuf),                          // Length of buffer
                            0);                                             // Flags
        if (nRet == INVALID_SOCKET)
        {
            PRINTERROR("recv()");
            closesocket(listenSocket);
            closesocket(remoteSocket);
            return;
        }

        //
        // Display received data
        //
        szBuf[nRet]=0;
        printf("\nData received: %s", szBuf);

        //
        // Send data back to the client
        //
        strcpy(szBuf, "From the Server");
        nRet = send(remoteSocket,                           // Connected socket
                            szBuf,                                  // Data buffer
                            strlen(szBuf),                          // Lenght of data
                            0);                                             // Flags

        //
        // Close BOTH sockets before exiting
        //
        closesocket(remoteSocket);
    closesocket(listenSocket);
        return;
}
```

Run DevC to compile sources.





3. Datagram Socket Programming

I have some simple single file console programs that demonstrate datagram Client, datagram Server, and a special program that most of you probably are aware of known as ping. All of these programs need **`wsock32.lib`** to be added to the project settings link tab properties. Without much commentary, here are the programs:

3.1. Datagram Client

```
//
// DClient.cpp
//
// Extremely simple, totally useless datagram client example.
// Works in conjunction with DServer.cpp.
//
// The program attempts to connect to the server and port
// specified on the command line. The DServer program prints
// the needed information when it is started. Once connected,
// the program sends data to the server, waits for a response
// and then exits.
//
// Compile and link with wsock32.lib
//
// Pass the server name and port number on the command line.
//
// Example: DClient MyMachineName 2000
//

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

// Function prototype
void DatagramClient(char *szServer, short nPort);

// Helper macro for displaying errors
#define PRINTERROR(s)   \
                fprintf(stderr,"\n%: %d\n", s, WSAGetLastError())

////////////////////////////////////////////////////////////

void main(int argc, char **argv)
{
        WORD wVersionRequested = MAKEWORD(1,1);
        WSADATA wsaData;
        int nRet;
        short nPort;

        //
        // Check for the host and port arguments
        //
        if (argc != 3)
        {
                fprintf(stderr,"\nSyntax: dclient ServerName PortNumber\n");
                return;
        }

        nPort = atoi(argv[2]);

        //
        // Initialize WinSock and check the version
```

```
        //
        nRet = WSAStartup(wVersionRequested, &wsaData);
        if (wsaData.wVersion != wVersionRequested)
        {
               fprintf(stderr,"\n Wrong version\n");
               return;
        }

        //
        // Go do all the stuff a datagram client does
        //
        DatagramClient(argv[1], nPort);

        //
        // Release WinSock
        //
        WSACleanup();
}

//////////////////////////////////////////////////////////

void DatagramClient(char *szServer, short nPort)
{

        printf("\nDatagram Client sending to server: %s on port: %d",
                              szServer, nPort);

        //
        // Find the server
        //
    LPHOSTENT lpHostEntry;

        lpHostEntry = gethostbyname(szServer);
    if (lpHostEntry == NULL)
    {
        PRINTERROR("gethostbyname()");
        return;
    }


        //
        // Create a UDP/IP datagram socket
        //
        SOCKET  theSocket;

        theSocket = socket(AF_INET,                    // Address family
                              SOCK_DGRAM,              // Socket type
                              IPPROTO_UDP);  // Protocol
        if (theSocket == INVALID_SOCKET)
        {
               PRINTERROR("socket()");
               return;
        }

        //
        // Fill in the address structure for the server
        //
        SOCKADDR_IN saServer;

        saServer.sin_family = AF_INET;
        saServer.sin_addr = *((LPIN_ADDR)*lpHostEntry->h_addr_list);
                                                                          // ^ Server's address
        saServer.sin_port = htons(nPort);     // Port number from command line

        //
        // Send data to the server
        //
        char szBuf[256];
        int nRet;

        strcpy(szBuf, "From the Client");
        nRet = sendto(theSocket, // Socket
                               szBuf,                                     // Data buffer
                              strlen(szBuf), // Length of data
                               0,                                         // Flags
        (LPSOCKADDR)&saServer, // Server address
        sizeof(struct sockaddr)); // Length of address
        if (nRet == SOCKET_ERROR)
        {
               PRINTERROR("sendto()");
               closesocket(theSocket);
               return;
        }

        //
        // Wait for the reply
        //
```

```
        memset(szBuf, 0, sizeof(szBuf));
        int nFromLen;

        nFromLen = sizeof(struct sockaddr);
        recvfrom(theSocket,                                     // Socket
                        szBuf,                                          // Receive buffer
                        sizeof(szBuf),                          // Length of receive buffer
                        0,                                              // Flags
                        (LPSOCKADDR)&saServer,                  // Buffer to receive sender's address
                        &nFromLen);                             // Length of address buffer
        if (nRet == SOCKET_ERROR)
        {
                PRINTERROR("recvfrom()");
                closesocket(theSocket);
                return;
        }

        //
        // Display the data that was received
        //
        printf("\nData received: %s", szBuf);

        closesocket(theSocket);
        return;
}

//------
```

3.2. Datagram Server

```
//-------

//
// DServer.cpp
//
// Extremely simple, totally useless datagram server example.
// Works in conjunction with DClient.cpp.
//
// The program sets itself up as a server using the UDP
// protoocl. It waits for data from a client, displays
// the incoming data, sends a message back to the client
// and then exits.
//
// Compile and link with wsock32.lib
//
// Pass the port number that the server should bind() to
// on the command line. Any port number not already in use
// can be specified.
//
// Example: DServer 2000
//

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

// Function prototype
void DatagramServer(short nPort);

// Helper macro for displaying errors
#define PRINTERROR(s)   \
                fprintf(stderr,"\n%: %d\n", s, WSAGetLastError())

//////////////////////////////////////////////////////////

void main(int argc, char **argv)
{
        WORD wVersionRequested = MAKEWORD(1,1);
        WSADATA wsaData;
        int nRet;
        short nPort;

        //
        // Check for port argument
        //
        if (argc != 2)
        {
                fprintf(stderr,"\nSyntax: dserver PortNumber\n");
                return;
        }

        nPort = atoi(argv[1]);

        //
        // Initialize WinSock and check version
        //
```

```
        nRet = WSAStartup(wVersionRequested, &wsaData);
        if (wsaData.wVersion != wVersionRequested)
        {
                fprintf(stderr,"\n Wrong version\n");
                return;
        }

        //
        // Do all the stuff a datagram server does
        //
        DatagramServer(nPort);

        //
        // Release WinSock
        //
        WSACleanup();
}

////////////////////////////////////////////////////////////

void DatagramServer(short nPort)
{
        //
        // Create a UDP/IP datagram socket
        //
        SOCKET theSocket;

        theSocket = socket(AF_INET,           // Address family
                                        SOCK_DGRAM,  // Socket type
                                        IPPROTO_UDP);// Protocol
        if (theSocket == INVALID_SOCKET)
        {
                PRINTERROR("socket()");
                return;
        }


        //
        // Fill in the address structure
        //
        SOCKADDR_IN saServer;

        saServer.sin_family = AF_INET;
        saServer.sin_addr.s_addr = INADDR_ANY; // Let WinSock assign address
        saServer.sin_port = htons(nPort);        // Use port passed from user


        //
        // bind the name to the socket
        //
        int nRet;

        nRet = bind(theSocket,                          // Socket descriptor
                                (LPSOCKADDR)&saServer,  // Address to bind to
                                sizeof(struct sockaddr)         // Size of address
                                );
        if (nRet == SOCKET_ERROR)
        {
                PRINTERROR("bind()");
                closesocket(theSocket);
                return;
        }


        //
        // This isn't normally done or required, but in this
        // example we're printing out where the server is waiting
        // so that you can connect the example client.
        //
        int nLen;
        nLen = sizeof(SOCKADDR);
        char szBuf[256];

        nRet = gethostname(szBuf, sizeof(szBuf));
        if (nRet == SOCKET_ERROR)
        {
                PRINTERROR("gethostname()");
                closesocket(theSocket);
                return;
        }

        //
        // Show the server name and port number
        //
        printf("\nServer named %s waiting on port %d\n",
                        szBuf, nPort);
```

```
        //
        // Wait for data from the client
        //
        SOCKADDR_IN saClient;

        memset(szBuf, 0, sizeof(szBuf));
        nRet = recvfrom(theSocket,                          // Bound socket
                                szBuf,                              // Receive buffer
                                sizeof(szBuf),                  // Size of buffer in bytes
                                0,                                          // Flags
                                (LPSOCKADDR)&saClient, // Buffer to receive client address
                                &nLen);                             // Length of client address buffer

        //
        // Show that we've received some data
        //
        printf("\nData received: %s", szBuf);


        //
        // Send data back to the client
        //
        strcpy(szBuf, "From the Server");
        sendto(theSocket,                                           // Bound socket
                  szBuf,                                                    // Send buffer
                  strlen(szBuf),                                    // Length of data to be sent
                  0,                                                        // Flags
                  (LPSOCKADDR)&saClient,                    // Address to send data to
                  nLen);                                                    // Length of address

        //
        // Normally a server continues to run so that
        // it is available to other clients. In this
        // example, we exit as soon as one transaction
        // has taken place.
        //
        closesocket(theSocket);
        return;
}

//------
```

---

4. Ping program

Ping (2 files: Ping.cpp and Ping.h)

Ping.h file:

```
//---------

//
// Ping.h
//

#pragma pack(1)

#define ICMP_ECHOREPLY 0
#define ICMP_ECHOREQ   8

// IP Header -- RFC 791
typedef struct tagIPHDR
{
        u_char  VIHL;                   // Version and IHL
        u_char  TOS;                    // Type Of Service
        short   TotLen;                 // Total Length
        short   ID;                             // Identification
        short   FlagOff;                // Flags and Fragment Offset
        u_char  TTL;                    // Time To Live
        u_char  Protocol;               // Protocol
        u_short Checksum;               // Checksum
        struct  in_addr iaSrc; // Internet Address - Source
        struct  in_addr iaDst; // Internet Address - Destination
}IPHDR, *PIPHDR;


// ICMP Header - RFC 792
typedef struct tagICMPHDR
{
        u_char  Type;                   // Type
        u_char  Code;                   // Code
        u_short Checksum;               // Checksum
        u_short ID;                             // Identification
        u_short Seq;                    // Sequence
        char    Data;                   // Data
}ICMPHDR, *PICMPHDR;
```

```
#define REQ_DATASIZE 32                // Echo Request Data size

// ICMP Echo Request
typedef struct tagECHOREQUEST
{
        ICMPHDR icmpHdr;
        DWORD   dwTime;
        char    cData[REQ_DATASIZE];
}ECHOREQUEST, *PECHOREQUEST;


// ICMP Echo Reply
typedef struct tagECHOREPLY
{
        IPHDR   ipHdr;
        ECHOREQUEST    echoRequest;
        char    cFiller[256];
}ECHOREPLY, *PECHOREPLY;


#pragma pack()


//---------
```

Ping.cpp file:

```
//-------

//
// PING.C -- Ping program using ICMP and RAW Sockets
//

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

#include "ping.h"

// Internal Functions
void Ping(LPCSTR pstrHost);
void ReportError(LPCSTR pstrFrom);
int  WaitForEchoReply(SOCKET s);
u_short in_cksum(u_short *addr, int len);

// ICMP Echo Request/Reply functions
int            SendEchoRequest(SOCKET, LPSOCKADDR_IN);
DWORD   RecvEchoReply(SOCKET, LPSOCKADDR_IN, u_char *);


// main()
void main(int argc, char **argv)
{
    WSADATA wsaData;
    WORD wVersionRequested = MAKEWORD(1,1);
    int nRet;

        // Check arguments
    if (argc != 2)
    {
            fprintf(stderr,"\nUsage: ping hostname\n");
            return;
    }

        // Init WinSock
    nRet = WSAStartup(wVersionRequested, &wsaData);
    if (nRet)
    {
            fprintf(stderr,"\nError initializing WinSock\n");
            return;
    }

        // Check version
        if (wsaData.wVersion != wVersionRequested)
        {
            fprintf(stderr,"\nWinSock version not supported\n");
            return;
        }

        // Go do the ping
        Ping(argv[1]);

        // Free WinSock
    WSACleanup();
}


// Ping()
// Calls SendEchoRequest() and
// RecvEchoReply() and prints results
```

```
void Ping(LPCSTR pstrHost)
{
        SOCKET    rawSocket;
        LPHOSTENT lpHost;
        struct    sockaddr_in saDest;
        struct    sockaddr_in saSrc;
        DWORD     dwTimeSent;
        DWORD     dwElapsed;
        u_char    cTTL;
        int       nLoop;
        int       nRet;

        // Create a Raw socket
        rawSocket = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
        if (rawSocket == SOCKET_ERROR)
        {
               ReportError("socket()");
               return;
        }

        // Lookup host
        lpHost = gethostbyname(pstrHost);
        if (lpHost == NULL)
        {
               fprintf(stderr,"\nHost not found: %s\n", pstrHost);
               return;
        }

        // Setup destination socket address
        saDest.sin_addr.s_addr = *((u_long FAR *) (lpHost->h_addr));
        saDest.sin_family = AF_INET;
        saDest.sin_port = 0;

        // Tell the user what we're doing
        printf("\nPinging %s [%s] with %d bytes of data:\n",
                               pstrHost,
                               inet_ntoa(saDest.sin_addr),
                               REQ_DATASIZE);

        // Ping multiple times
        for (nLoop = 0; nLoop < 4; nLoop++)
        {
               // Send ICMP echo request
               SendEchoRequest(rawSocket, &saDest);

               // Use select() to wait for data to be received
               nRet = WaitForEchoReply(rawSocket);
               if (nRet == SOCKET_ERROR)
               {
                       ReportError("select()");
                       break;
               }
               if (!nRet)
               {
                       printf("\nTimeOut");
                       break;
               }

               // Receive reply
               dwTimeSent = RecvEchoReply(rawSocket, &saSrc, &cTTL);

               // Calculate elapsed time
               dwElapsed = GetTickCount() - dwTimeSent;
               printf("\nReply from: %s: bytes=%d time=%ldms TTL=%d",
               inet_ntoa(saSrc.sin_addr),
                         REQ_DATASIZE,
               dwElapsed,
               cTTL);
        }
        printf("\n");
        nRet = closesocket(rawSocket);
        if (nRet == SOCKET_ERROR)
               ReportError("closesocket()");
}


// SendEchoRequest()
// Fill in echo request header
// and send to destination
int SendEchoRequest(SOCKET s,LPSOCKADDR_IN lpstToAddr)
{
        static ECHOREQUEST echoReq;
        static nId = 1;
        static nSeq = 1;
        int nRet;

        // Fill in echo request
        echoReq.icmpHdr.Type           = ICMP_ECHOREQ;
        echoReq.icmpHdr.Code           = 0;
        echoReq.icmpHdr.Checksum       = 0;
        echoReq.icmpHdr.ID                     = nId++;
        echoReq.icmpHdr.Seq                    = nSeq++;
```

```
        // Fill in some data to send
        for (nRet = 0; nRet < REQ_DATASIZE; nRet++)
               echoReq.cData[nRet] = ' '+nRet;

        // Save tick count when sent
        echoReq.dwTime                        = GetTickCount();

        // Put data in packet and compute checksum
        echoReq.icmpHdr.Checksum = in_cksum((u_short *)&echoReq, sizeof(ECHOREQUEST));

        // Send the echo request
        nRet = sendto(s,                                            /* socket */
                              (LPSTR)&echoReq,                      /* buffer */
                              sizeof(ECHOREQUEST),
                              0,                                                  /* flags */
                              (LPSOCKADDR)lpstToAddr, /* destination */
                              sizeof(SOCKADDR_IN));   /* address length */

        if (nRet == SOCKET_ERROR)
               ReportError("sendto()");
        return (nRet);
}


// RecvEchoReply()
// Receive incoming data
// and parse out fields
DWORD RecvEchoReply(SOCKET s, LPSOCKADDR_IN lpsaFrom, u_char *pTTL)
{
        ECHOREPLY echoReply;
        int nRet;
        int nAddrLen = sizeof(struct sockaddr_in);

        // Receive the echo reply
        nRet = recvfrom(s,                                    // socket
                                      (LPSTR)&echoReply,      // buffer
                                      sizeof(ECHOREPLY),      // size of buffer
                                      0,                                  // flags
                                      (LPSOCKADDR)lpsaFrom,   // From address
                                      &nAddrLen);                         // pointer to address len

        // Check return value
        if (nRet == SOCKET_ERROR)
               ReportError("recvfrom()");

        // return time sent and IP TTL
        *pTTL = echoReply.ipHdr.TTL;
        return(echoReply.echoRequest.dwTime);
}

// What happened?
void ReportError(LPCSTR pWhere)
{
        fprintf(stderr,"\n%s error: %d\n",
               WSAGetLastError());
}


// WaitForEchoReply()
// Use select() to determine when
// data is waiting to be read
int WaitForEchoReply(SOCKET s)
{
        struct timeval Timeout;
        fd_set readfds;

        readfds.fd_count = 1;
        readfds.fd_array[0] = s;
        Timeout.tv_sec = 5;
    Timeout.tv_usec = 0;

        return(select(1, &readfds, NULL, NULL, &Timeout));
}


//
// Mike Muuss' in_cksum() function
// and his comments from the original
// ping program
//
// * Author -
// *    Mike Muuss
// *    U. S. Army Ballistic Research Laboratory
// *    December, 1983

/*
 *                      I N _ C K S U M
 *
 * Checksum routine for Internet Protocol family headers (C Version)
 *
 */
u_short in_cksum(u_short *addr, int len)
{
```

```
        register int nleft = len;
        register u_short *w = addr;
        register u_short answer;
        register int sum = 0;

        /*
         *  Our algorithm is simple, using a 32 bit accumulator (sum),
         *  we add sequential 16 bit words to it, and at the end, fold
         *  back all the carry bits from the top 16 bits into the lower
         *  16 bits.
         */
        while( nleft > 1 )  {
               sum += *w++;
               nleft -= 2;
        }

        /* mop up an odd byte, if necessary */
        if( nleft == 1 ) {
               u_short u = 0;

               *(u_char *)(&u) = *(u_char *)w ;
               sum += u;
        }

        /*
         * add back carry outs from top 16 bits to low 16 bits
         */
        sum = (sum >> 16) + (sum & 0xffff);   /* add hi 16 to low 16 */
        sum += (sum >> 16);                   /* add carry */
        answer = ~sum;                        /* truncate to 16 bits */
        return (answer);
}

//------
```

5. TCP client-server program example

Abilities:

- Able to understand the advanced networking of the TCP/IP.
- Able to understand Winsock implementation and operations through the APIs and program examples.
- Able to gather, understand and use the Winsock functions, structures and macros in your programs.
- Able to build programs that use Microsoft C/Standard C programming language and Winsock APIs.

addrinfo Structure

Item	Description
Structure	`addrinfo`
Info	Used by the getaddrinfo() function to hold host address information.

Definition	`typedef struct addrinfo {`
	`  int ai_flags;`
	`  int ai_family;`
	`  int ai_socktype;`
	`  int ai_protocol;`
	`  size_t ai_addrlen;`
	`  char* ai_canonname;`
	`  struct sockaddr* ai_addr;`
	`  struct addrinfo* ai_next;`
	`} addrinfo;`

Members	ai_flags - Flags that indicate options used in the getaddrinfo() function. See AI_PASSIVE, AI_CANONNAME, and AI_NUMERICHOST. ai_family - Protocol family, such as PF_INET. ai_socktype - Socket type, such as SOCK_RAW, SOCK_STREAM, or SOCK_DGRAM. ai_protocol - Protocol, such as IPPROTO_TCP or IPPROTO_UDP. For protocols other than IPv4 and IPv6, set ai_protocol to zero. ai_addrlen - Length of the ai_addr member, in bytes. ai_canonname - Canonical name for the host. ai_addr - Pointer to a sockaddr structure. ai_next - Pointer to the next structure in a linked list. This parameter is set to NULL in the last addrinfo structure of a linked list.
Header file	<winsock2.h>, <ws2tcpip.h>.
Remark	Declared in ws2tcpip.h; include wspiapi.h for Windows 95/98/Me, Windows 2000 and Windows NT.

Table 1

freeaddrinfo()

Item	Description
Function	`freeaddrinfo()`
Use	Frees address information that the getaddrinfo() function dynamically allocates in its addrinfo structures.
Prototype	`void freeaddrinfo(struct addrinfo* ai);`

Parameters	ai - [in] Pointer to the addrinfo structure or linked list of addrinfo structures to be freed. All dynamic storage pointed to within the addrinfo structure(s) is also freed.
Return value	This function has no return values.
Include file	<winsock2.h>, <ws2tcpip.h>
Library	ws2_32.lib
Remark	Declared in ws2tcpip.h; include wspiapi.h for Windows 95/98/Me, Windows 2000 and Windows NT. The freeaddrinfo() function frees the initial addrinfo structure pointed to in its ai parameter, including any buffers to which its members point, then continues freeing any addrinfo structures linked by its ai_next member. The freeaddrinfo() function continues freeing linked structures until a NULL ai_next member is encountered. Macros in the Winsock header file define the mixed-case function name FreeAddrInfo() to freeaddrinfo(); either spelling is acceptable.

Table 2

.

A Complete client-server program example

The following codes are server and client program examples that used the previous discussed Winsock functions and structures. In this case, to make the client-server communication possible you have to make the server settings (through the arguments- protocol, port number etc.) must match with the client settings. For example if you choose to run the UDP server, the client also must be UDP client. Please run the server program first and you can run the client from different computers.

```
/* Server program example for IPv4 */
#define WIN32_LEAN_AND_MEAN
#include <winsock2.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>


#define DEFAULT_PORT 2007
// default TCP socket type
#define DEFAULT_PROTO SOCK_STREAM

void Usage(char *progname)
{
    fprintf(stderr,"Usage: %s -p [protocol] -e [port_num] -i [ip_address]\n", progname);
    fprintf(stderr,"Where:\n\t- protocol is one of TCP or UDP\n");
    fprintf(stderr,"\t- port_num is the port to listen on\n");
    fprintf(stderr,"\t- ip_address is the ip address (in dotted\n");
    fprintf(stderr,"\t  decimal notation) to bind to. But it is not useful here...\n");
    fprintf(stderr,"\t- Hit Ctrl-C to terminate server program...\n");
    fprintf(stderr,"\t- The defaults are TCP, 2007 and INADDR_ANY.\n");
    WSACleanup();
    exit(1);
}

int main(int argc, char **argv)
{
    char Buffer[128];
    char *ip_address= NULL;
    unsigned short port=DEFAULT_PORT;
    int retval;
    int fromlen;
    int i;
    int socket_type = DEFAULT_PROTO;
    struct sockaddr_in local, from;
    WSADATA wsaData;
    SOCKET listen_socket, msgsock;

    /* Parse arguments, if there are arguments supplied */
    if (argc > 1)
       {
        for(i=1; i<argc; i++)
              {
                     // switches or options...
            if ((argv[i][0] == '-') || (argv[i][0] == '/'))
                     {
                            // Change to lower...if any
                           switch(tolower(argv[i][1]))
                           {
                     // if -p or /p
                    case 'p':
                        if (!stricmp(argv[i+1], "TCP"))
                            socket_type = SOCK_STREAM;
                        else if (!stricmp(argv[i+1], "UDP"))
                            socket_type = SOCK_DGRAM;
                        else
                            Usage(argv[0]);
                        i++;
                        break;
```

```
                    // if -i or /i, for server it is not so useful...
                   case 'i':
                       ip_address = argv[++i];
                       break;
                   // if -e or /e
                   case 'e':
                       port = atoi(argv[++i]);
                       break;
                    // No match...
                   default:
                       Usage(argv[0]);
                       break;
               }
           }
           else
               Usage(argv[0]);
       }
   }

   // Request Winsock version 2.2
   if ((retval = WSAStartup(0x202, &wsaData)) != 0)
      {
       fprintf(stderr,"Server: WSAStartup() failed with error %d\n", retval);
       WSACleanup();
       return -1;
   }
   else
      printf("Server: WSAStartup() is OK.\n");

   if (port == 0)
      {
       Usage(argv[0]);
   }

   local.sin_family = AF_INET;
   local.sin_addr.s_addr = (!ip_address) ? INADDR_ANY:inet_addr(ip_address);

   /* Port MUST be in Network Byte Order */
   local.sin_port = htons(port);
   // TCP socket
   listen_socket = socket(AF_INET, socket_type,0);

   if (listen_socket == INVALID_SOCKET){
       fprintf(stderr,"Server: socket() failed with error %d\n", WSAGetLastError());
       WSACleanup();
       return -1;
   }
   else
      printf("Server: socket() is OK.\n");

   // bind() associates a local address and port combination with the socket just created.
   // This is most useful when the application is a
   // server that has a well-known port that clients know about in advance.
   if (bind(listen_socket, (struct sockaddr*)&local, sizeof(local)) == SOCKET_ERROR)
      {
       fprintf(stderr,"Server: bind() failed with error %d\n", WSAGetLastError());
       WSACleanup();
       return -1;
   }
   else
             printf("Server: bind() is OK.\n");

    // So far, everything we did was applicable to TCP as well as UDP.
    // However, there are certain steps that do not work when the server is
    // using UDP. We cannot listen() on a UDP socket.
   if (socket_type != SOCK_DGRAM)
      {
       if (listen(listen_socket,5) == SOCKET_ERROR)
             {
           fprintf(stderr,"Server: listen() failed with error %d\n", WSAGetLastError());
           WSACleanup();
           return -1;
       }
      else
             printf("Server: listen() is OK.\n");
   }
   printf("Server: %s: I'm listening and waiting connection\non port %d, protocol %s\n", argv[0], port,
(socket_type == SOCK_STREAM)?"TCP":"UDP");

      while(1)
      {
       fromlen =sizeof(from);
       // accept() doesn't make sense on UDP, since we do not listen()
       if (socket_type != SOCK_DGRAM)
             {
           msgsock = accept(listen_socket, (struct sockaddr*)&from, &fromlen);
           if (msgsock == INVALID_SOCKET)
```

```
            {
                fprintf(stderr,"Server: accept() error %d\n", WSAGetLastError());
                WSACleanup();
                return -1;
             }
            else
               printf("Server: accept() is OK.\n");
               printf("Server: accepted connection from %s, port %d\n", inet_ntoa(from.sin_addr),
htons(from.sin_port)) ;

         }
         else
             msgsock = listen_socket;

         // In the case of SOCK_STREAM, the server can do recv() and send() on
         // the accepted socket and then close it.
         // However, for SOCK_DGRAM (UDP), the server will do recvfrom() and sendto()  in a loop.
         if (socket_type != SOCK_DGRAM)
             retval = recv(msgsock, Buffer, sizeof(Buffer), 0);

        else
        {
             retval = recvfrom(msgsock,Buffer, sizeof(Buffer), 0, (struct sockaddr *)&from, &fromlen);
             printf("Server: Received datagram from %s\n", inet_ntoa(from.sin_addr));
         }

         if (retval == SOCKET_ERROR)
               {
             fprintf(stderr,"Server: recv() failed: error %d\n", WSAGetLastError());
             closesocket(msgsock);
             continue;
         }
        else
             printf("Server: recv() is OK.\n");

         if (retval == 0)
               {
             printf("Server: Client closed connection.\n");
             closesocket(msgsock);
             continue;
         }
         printf("Server: Received %d bytes, data \"%s\" from client\n", retval, Buffer);

         printf("Server: Echoing the same data back to client...\n");
         if (socket_type != SOCK_DGRAM)
             retval = send(msgsock, Buffer, sizeof(Buffer), 0);
         else
             retval = sendto(msgsock, Buffer, sizeof(Buffer), 0, (struct sockaddr *)&from, fromlen);

               if (retval == SOCKET_ERROR)
               {
                      fprintf(stderr,"Server: send() failed: error %d\n", WSAGetLastError());
                }
               else
                      printf("Server: send() is OK.\n");

         if (socket_type != SOCK_DGRAM)
        {
             printf("Server: I'm waiting more connection, try running the client\n");
             printf("Server: program from the same computer or other computer...\n");
             closesocket(msgsock);
         }
         else
             printf("Server: UDP server looping back for more requests\n");
         continue;
     }
        return 0;
}
```

Sample output:

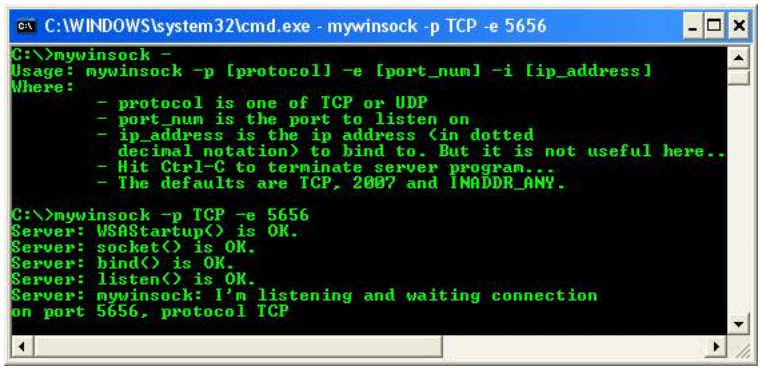


Figure 1

The following is the client program.

```
// Client program example
#define WIN32_LEAN_AND_MEAN
#include <winsock2.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define DEFAULT_PORT 2007
// TCP socket type
#define DEFAULT_PROTO SOCK_STREAM

void Usage(char *progname)
{
    fprintf(stderr,"Usage: %s -p [protocol] -n [server name/IP] -e [port_num] -l [iterations]\n",
progname);
    fprintf(stderr,"Where:\n\tprotocol is one of TCP or UDP\n");
    fprintf(stderr,"\t- server is the IP address or name of server\n");
    fprintf(stderr,"\t- port_num is the port to listen on\n");
    fprintf(stderr,"\t- iterations is the number of loops to execute.\n");
    fprintf(stderr,"\t- (-l by itself makes client run in an infinite loop,\n");
    fprintf(stderr,"\t- Hit Ctrl-C to terminate it)\n");
    fprintf(stderr,"\t- The defaults are TCP , localhost and 2007\n");
    WSACleanup();
    exit(1);
}

int main(int argc, char **argv)
{
    char Buffer[128];
    // default to localhost
    char *server_name= "localhost";
    unsigned short port = DEFAULT_PORT;
    int retval, loopflag = 0;
    int i, loopcount, maxloop=-1;
    unsigned int addr;
    int socket_type = DEFAULT_PROTO;
    struct sockaddr_in server;
    struct hostent *hp;
    WSADATA wsaData;
    SOCKET  conn_socket;

    if (argc >1)
       {
        for(i=1; i<argc; i++)
        {
            if ((argv[i][0] == '-') || (argv[i][0] == '/'))
           {
                switch(tolower(argv[i][1]))
                 {
                    case 'p':
                        if (!stricmp(argv[i+1], "TCP"))
                            socket_type = SOCK_STREAM;
                        else if (!stricmp(argv[i+1], "UDP"))
                            socket_type = SOCK_DGRAM;
                        else
                            Usage(argv[0]);
                        i++;
                        break;
                    case 'n':
                        server_name = argv[++i];
                        break;
                    case 'e':
                        port = atoi(argv[++i]);
                        break;
                    case 'l':
                        loopflag =1;
                        if (argv[i+1]) {
                            if (argv[i+1][0] != '-')
                                maxloop = atoi(argv[i+1]);
                        }
                        else
                            maxloop = -1;
                        i++;
                        break;
                    default:
                        Usage(argv[0]);
                        break;
                }
            }
            else
```

```
                Usage(argv[0]);
        }
    }

    if ((retval = WSAStartup(0x202, &wsaData)) != 0)
    {
       fprintf(stderr,"Client: WSAStartup() failed with error %d\n", retval);
        WSACleanup();
        return -1;
    }
    else
       printf("Client: WSAStartup() is OK.\n");

    if (port == 0)
    {
        Usage(argv[0]);
    }
    // Attempt to detect if we should call gethostbyname() or gethostbyaddr()
    if (isalpha(server_name[0]))
    {   // server address is a name
        hp = gethostbyname(server_name);
    }
    else
    { // Convert nnn.nnn address to a usable one
        addr = inet_addr(server_name);
        hp = gethostbyaddr((char *)&addr, 4, AF_INET);
    }
    if (hp == NULL )
    {
        fprintf(stderr,"Client: Cannot resolve address \"%s\": Error %d\n", server_name,
WSAGetLastError());
        WSACleanup();
        exit(1);
    }
    else
       printf("Client: gethostbyaddr() is OK.\n");
    // Copy the resolved information into the sockaddr_in structure
    memset(&server, 0, sizeof(server));
    memcpy(&(server.sin_addr), hp->h_addr, hp->h_length);
    server.sin_family = hp->h_addrtype;
    server.sin_port = htons(port);

    conn_socket = socket(AF_INET, socket_type, 0); /* Open a socket */
    if (conn_socket <0 )
    {
        fprintf(stderr,"Client: Error Opening socket: Error %d\n", WSAGetLastError());
        WSACleanup();
        return -1;
    }
    else
       printf("Client: socket() is OK.\n");

    // Notice that nothing in this code is specific to whether we
    // are using UDP or TCP.
    // We achieve this by using a simple trick.
    //    When connect() is called on a datagram socket, it does not
    //    actually establish the connection as a stream (TCP) socket
    //    would. Instead, TCP/IP establishes the remote half of the
    //    (LocalIPAddress, LocalPort, RemoteIP, RemotePort) mapping.
    //    This enables us to use send() and recv() on datagram sockets,
    //    instead of recvfrom() and sendto()
    printf("Client: Client connecting to: %s.\n", hp->h_name);
    if (connect(conn_socket, (struct sockaddr*)&server, sizeof(server)) == SOCKET_ERROR)
    {
        fprintf(stderr,"Client: connect() failed: %d\n", WSAGetLastError());
        WSACleanup();
        return -1;
    }
    else
       printf("Client: connect() is OK.\n");

    // Test sending some string
    loopcount = 0;
    while(1)
    {
         wsprintf(Buffer,"This is a test message from client #%d", loopcount++);
        retval = send(conn_socket, Buffer, sizeof(Buffer), 0);
        if (retval == SOCKET_ERROR)
        {
            fprintf(stderr,"Client: send() failed: error %d.\n", WSAGetLastError());
            WSACleanup();
            return -1;
        }
        else
          printf("Client: send() is OK.\n");
        printf("Client: Sent data \"%s\"\n", Buffer);
```

```
            retval = recv(conn_socket, Buffer, sizeof(Buffer), 0);
        if (retval == SOCKET_ERROR)
        {
            fprintf(stderr,"Client: recv() failed: error %d.\n", WSAGetLastError());
            closesocket(conn_socket);
            WSACleanup();
            return -1;
        }
        else
            printf("Client: recv() is OK.\n");

        // We are not likely to see this with UDP, since there is no
        // 'connection' established.
        if (retval == 0)
        {
            printf("Client: Server closed connection.\n");
            closesocket(conn_socket);
            WSACleanup();
            return -1;
        }

        printf("Client: Received %d bytes, data \"%s\" from server.\n", retval, Buffer);
        if (!loopflag)
        {
            printf("Client: Terminating connection...\n");
            break;
        }
        else
        {
            if ((loopcount >= maxloop) && (maxloop >0))
            break;
        }
    }
    closesocket(conn_socket);
    WSACleanup();

return 0;
}
```

The following is an output when the client program has been run twice using the default arguments. Remember that you have to run the server program first.

```
C:\>myclient -
Usage: myclient -p [protocol] -n [server name/IP] -e [port_num] -l [iterations]
Where:
        protocol is one of TCP or UDP
        - server is the IP address or name of server
        - port_num is the port to listen on
        - iterations is the number of loops to execute.
        - (-l by itself makes client run in an infinite loop,
        - Hit Ctrl-C to terminate it)
        - The defaults are TCP , localhost and 2007

C:\>myclient -p TCP -n 127.0.0.1 -e 5656 -l 3
Client: WSAStartup() is OK.
Client: gethostbyaddr() is OK.
Client: socket() is OK.
Client: Client connecting to: localhost.
Client: connect() is OK.
Client: send() is OK.
Client: Sent data "This is a test message from client #0"
Client: recv() is OK.
Client: Received 128 bytes, data "This is a test message from client #0" from server.
Client: send() is OK.
Client: Sent data "This is a test message from client #1"
Client: recv() failed: error 10053.

C:\>myclient -p TCP -n 127.0.0.1 -e 5656 -l 3
Client: WSAStartup() is OK.
Client: gethostbyaddr() is OK.
Client: socket() is OK.
Client: Client connecting to: localhost.
Client: connect() is OK.
Client: send() is OK.
Client: Sent data "This is a test message from client #0"
Client: recv() is OK.
Client: Received 128 bytes, data "This is a test message from client #0" from server.
Client: send() is OK.
Client: Sent data "This is a test message from client #1"
Client: recv() failed: error 10053.

C:\>
```

Notice that the iterations is not working. And the server console output is shown below after the same client program has been run twice.

```
C:\>mywinsock -
Usage: mywinsock -p [protocol] -e [port_num] -i [ip_address]
Where:
        - protocol is one of TCP or UDP
        - port_num is the port to listen on
        - ip_address is the ip address (in dotted
          decimal notation) to bind to. But it is not useful here...
        - Hit Ctrl-C to terminate server program...
        - The defaults are TCP, 2007 and INADDR_ANY.

C:\>mywinsock -p TCP -e 5656
Server: WSAStartup() is OK.
Server: socket() is OK.
Server: bind() is OK.
Server: listen() is OK.
Server: mywinsock: I'm listening and waiting connection
on port 5656, protocol TCP
Server: accept() is OK.
Server: accepted connection from 127.0.0.1, port 1031
Server: recv() is OK.
Server: Received 128 bytes, data "This is a test message from client #0" from client
Server: Echoing the same data back to client...
Server: send() is OK.
Server: I'm waiting more connection, try running the client
Server: program from the same computer or other computer...
Server: accept() is OK.
Server: accepted connection from 127.0.0.1, port 1032
Server: recv() is OK.
Server: Received 128 bytes, data "This is a test message from client #0" from client
Server: Echoing the same data back to client...
Server: send() is OK.
Server: I'm waiting more connection, try running the client
Server: program from the same computer or other computer...
_
```

The following console outputs are the previous client-server program examples re-compiled using Visual C++ 6.0 and using public IPs.

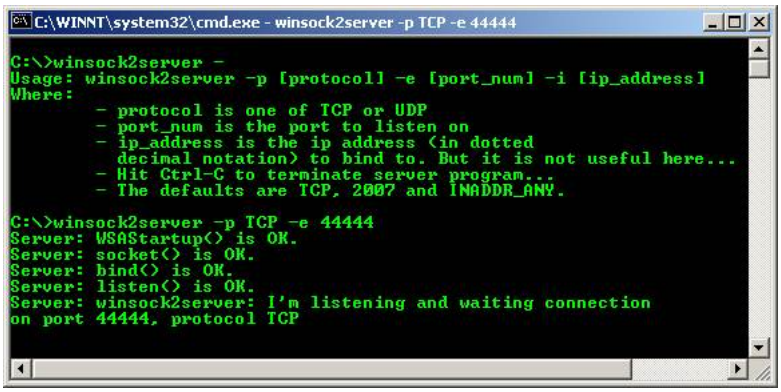


Figure 2

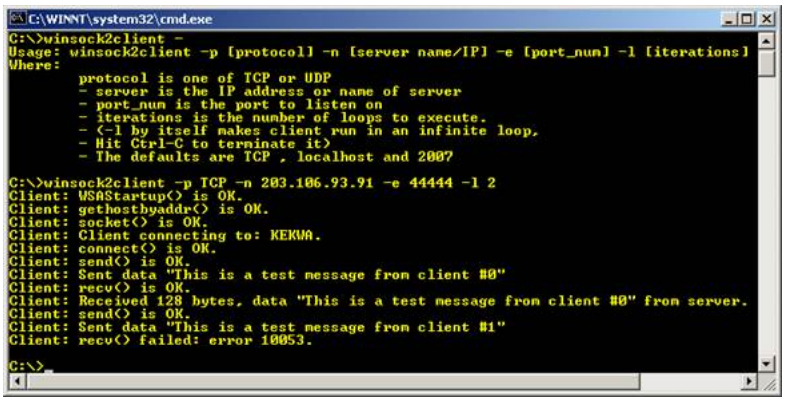


Figure 3

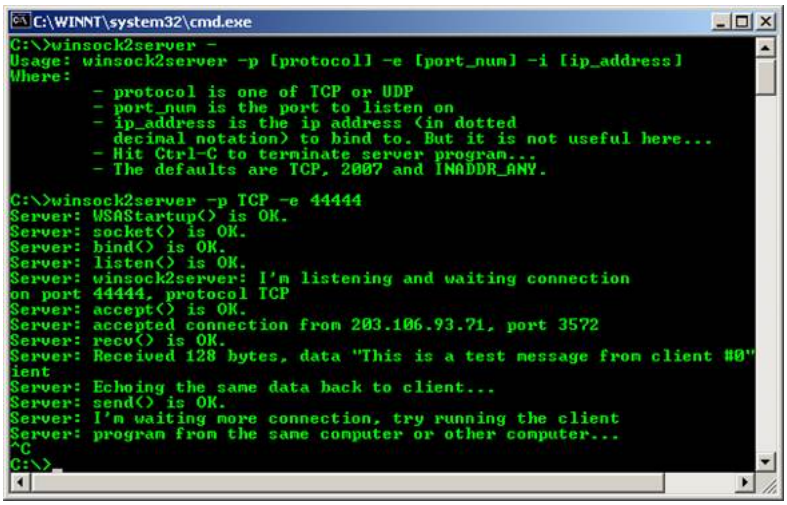


Figure 4

6. IPv6 Client-server Programs

The IPv6 Client Program Example
Testing the IPv6 Client-server Programs

Abilities:

- Able to understand Winsock implementation and operations of the IPv6 through the APIs and program examples.
- Able to gather, understand and use the Winsock functions, structures and macros in your programs.
- Able to build programs that use Microsoft C/Standard C programming language and Winsock APIs.

IPv6 Server Program Example

The following is a client-server program example for IPv6 enabled machines. The first one is the server program example run for IPv4. If you are in IPv6 network environment, please try running the program using the IPv6 address and PF_INET6 family.

```
// IPv6 server program example. Try running it on the IPv6 enabled machines using IPv6 arguments
#define WIN32_LEAN_AND_MEAN
#include <winsock2.h>
#include <ws2tcpip.h>
#ifndef IPPROTO_IPV6
// For IPv6.
#include <tpipv6.h>
#endif
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

// This code assumes that at the transport level, the system only supports
// one stream protocol (TCP) and one datagram protocol (UDP).  Therefore,
// specifying a socket type of SOCK_STREAM is equivalent to specifying TCP
// and specifying a socket type of
// SOCK_DGRAM is equivalent to specifying UDP.

// Accept either IPv4 or IPv6
#define DEFAULT_FAMILY     PF_UNSPEC
// TCP socket type
#define DEFAULT_SOCKTYPE   SOCK_STREAM
// Arbitrary, test port
#define DEFAULT_PORT       "2007"
// Set very small for demonstration purposes
#define BUFFER_SIZE        128

void Usage(char *ProgName)
{
    fprintf(stderr, "\nSimple socket server program.\n");
    fprintf(stderr, "\n%s [-f family] [-t transport] [-p port] [-a address]\n\n", ProgName);
    fprintf(stderr, "  family\tOne of PF_INET, PF_INET6 or PF_UNSPEC. (default %s)\n",
            (DEFAULT_FAMILY == PF_UNSPEC) ? "PF_UNSPEC" :
            ((DEFAULT_FAMILY == PF_INET) ? "PF_INET" : "PF_INET6"));
    fprintf(stderr, "  transport\tEither TCP or UDP. (default: %s)\n", (DEFAULT_SOCKTYPE == SOCK_STREAM) ?
"TCP" : "UDP");
    fprintf(stderr, "  port\t\tPort on which to bind. (default %s)\n", DEFAULT_PORT);
```

```
    fprintf(stderr, "  address\tIP address on which to bind.  (default: unspecified address)\n");
    WSACleanup();
    exit(1);
}

LPSTR DecodeError(int ErrorCode)
{
    static char Message[1024];

    // If this program was multi-threaded, we'd want to use FORMAT_MESSAGE_ALLOCATE_BUFFER
    // instead of a static buffer here.
    // (And of course, free the buffer when we were done with it)
    FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_IGNORE_INSERTS |
                  FORMAT_MESSAGE_MAX_WIDTH_MASK, NULL, ErrorCode,
                  MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPSTR)Message, 1024, NULL);
    return Message;
}

int main(int argc, char **argv)
{
    char Buffer[BUFFER_SIZE], Hostname[NI_MAXHOST];
    int Family = DEFAULT_FAMILY;
    int SocketType = DEFAULT_SOCKTYPE;
    char *Port = DEFAULT_PORT;
    char *Address = NULL;
    int i, NumSocks, RetVal, FromLen, AmountRead;
    SOCKADDR_STORAGE From;
    WSADATA wsaData;
    ADDRINFO Hints, *AddrInfo, *AI;
    SOCKET ServSock[FD_SETSIZE];
    fd_set SockSet;

       printf("Usage: %s [-f family] [-t transport] [-p port] [-a address]\n", argv[0]);
       printf("Example: %s -f PF_INET6 -t TCP -p 1234 -a 127.0.0.1\n", argv[0]);
       printf("Else, default values used. By the way, the -a not usable for server...\n");
       printf("Ctrl + C to terminate the server program.\n\n");

    // Parse arguments
    if (argc > 1)
    {
        for(i = 1;i < argc; i++)
       {
            if ((argv[i][0] == '-') || (argv[i][0] == '/') &&
                (argv[i][1] != 0) && (argv[i][2] == 0))
           {
                switch(tolower(argv[i][1]))
               {
                    case 'f':
                        if (!argv[i+1])
                            Usage(argv[0]);
                        if (!stricmp(argv[i+1], "PF_INET"))
                            Family = PF_INET;
                        else if (!stricmp(argv[i+1], "PF_INET6"))
                            Family = PF_INET6;
                        else if (!stricmp(argv[i+1], "PF_UNSPEC"))
                            Family = PF_UNSPEC;
                        else
                            Usage(argv[0]);
                        i++;
                        break;

                    case 't':
                        if (!argv[i+1])
                            Usage(argv[0]);
                        if (!stricmp(argv[i+1], "TCP"))
                            SocketType = SOCK_STREAM;
                        else if (!stricmp(argv[i+1], "UDP"))
                            SocketType = SOCK_DGRAM;
                        else
                            Usage(argv[0]);
                        i++;
                        break;

                    case 'a':
                        if (argv[i+1])
                      {
                            if (argv[i+1][0] != '-')
                           {
                                Address = argv[++i];
                                break;
                            }
                        }
                        Usage(argv[0]);
                        break;

                    case 'p':
                        if (argv[i+1])
```

```
                        {
                            if (argv[i+1][0] != '-')
                          {
                                Port = argv[++i];
                                break;
                            }
                        }
                        Usage(argv[0]);
                        break;

                    default:
                        Usage(argv[0]);
                        break;
                }
            } else
                Usage(argv[0]);
        }
    }

    // Ask for Winsock version 2.2...
    if ((RetVal = WSAStartup(MAKEWORD(2, 2), &wsaData)) != 0)
    {
       fprintf(stderr, "Server: WSAStartup() failed with error %d: %s\n", RetVal, DecodeError(RetVal));
        WSACleanup();
        return -1;
    }
    else
         printf("Server: WSAStartup() is OK.\n");

    if (Port == NULL)
    {
        Usage(argv[0]);
    }

    // By setting the AI_PASSIVE flag in the hints to getaddrinfo(),
    // we're indicating that we intend to use the resulting address(es) to bind
    // to a socket(s) for accepting incoming connections.  This means
    // that when the Address parameter is NULL, getaddrinfo() will return one
    // entry per allowed protocol family containing the unspecified address for that family.
    memset(&Hints, 0, sizeof(Hints));
    Hints.ai_family   = Family;
    Hints.ai_socktype = SocketType;
    Hints.ai_flags    = AI_NUMERICHOST | AI_PASSIVE;
    RetVal = getaddrinfo(Address, Port, &Hints, &AddrInfo);
    if (RetVal != 0)
    {
        fprintf(stderr, "getaddrinfo() failed with error %d: %s\n", RetVal, gai_strerror(RetVal));
        WSACleanup();
        return -1;
    }
    else
       printf("Server: getaddrinfo() is OK.\n");

    // For each address getaddrinfo returned, we create a new socket,
    // bind that address to it, and create a queue to listen on.
    for (i = 0, AI = AddrInfo; AI != NULL; AI = AI->ai_next, i++)
       {
        // Highly unlikely, but check anyway.
        if (i == FD_SETSIZE)
       {
            printf("Server: getaddrinfo() returned more addresses than we could use.\n");
            break;
        }

        // This example only supports PF_INET and PF_INET6.
        if ((AI->ai_family != PF_INET) && (AI->ai_family != PF_INET6))
            continue;

        // Open a socket with the correct address family for this address.
        ServSock[i] = socket(AI->ai_family, AI->ai_socktype, AI->ai_protocol);
        if (ServSock[i] == INVALID_SOCKET)
       {
            fprintf(stderr, "Server: socket() failed with error %d: %s\n", WSAGetLastError(),
DecodeError(WSAGetLastError()));
            continue;
        }
       else
           printf("Server: socket() is OK.\n");

        // bind() associates a local address and port combination with
        // the socket just created. This is most useful when
        // the application is a server that has a well-known port that
        // clients know about in advance.
        if (bind(ServSock[i], AI->ai_addr, int(AI->ai_addrlen)) == SOCKET_ERROR)
       {
            fprintf(stderr,"Server: bind() failed with error %d: %s\n", WSAGetLastError(),
DecodeError(WSAGetLastError()));
```

```
            continue;
        }
       else
           printf("Server: bind() is OK.\n");

        // So far, everything we did was applicable to TCP as well as UDP.
        // However, there are certain fundamental differences between stream
        // protocols such as TCP and datagram protocols such as UDP.
        //
        // Only connection orientated sockets, for example those of
        // type SOCK_STREAM, can listen() for incoming connections.
        if (SocketType == SOCK_STREAM)
       {
            if (listen(ServSock[i], 5) == SOCKET_ERROR)
           {
              fprintf(stderr, "Server: listen() failed with error %d: %s\n", WSAGetLastError(),
DecodeError(WSAGetLastError()));
                continue;
            }
           else
              printf("Server: listen() is OK.\n");
        }

        printf("I'm listening and waiting on port %s, protocol %s, protocol family %s\n",
               Port, (SocketType == SOCK_STREAM) ? "TCP" : "UDP",
               (AI->ai_family == PF_INET) ? "PF_INET" : "PF_INET6");
    }

    freeaddrinfo(AddrInfo);

    if (i == 0)
   {
        fprintf(stderr, "Fatal error: unable to serve on any address.\n");
        WSACleanup();
        return -1;
    }
    NumSocks = i;

    // We now put the server into an eternal loop, serving requests as they arrive.
    FD_ZERO(&SockSet);
    while(1)
    {
        FromLen = sizeof(From);
        // For connection orientated protocols, we will handle
        // the packets comprising a connection collectively.  For datagram
        // protocols, we have to handle each datagram individually.
        //
        // Check to see if we have any sockets remaining to be
        // served from previous time through this loop.  If not, call select()
        // to wait for a connection request or a datagram to arrive.
        for (i = 0; i < NumSocks; i++)
       {
            if (FD_ISSET(ServSock[i], &SockSet))
                break;
        }
        if (i == NumSocks)
       {
            for (i = 0; i < NumSocks; i++)
                FD_SET(ServSock[i], &SockSet);
            if (select(NumSocks, &SockSet, 0, 0, 0) == SOCKET_ERROR)
           {
              fprintf(stderr, "Server: select() failed with error %d: %s\n", WSAGetLastError(),
DecodeError(WSAGetLastError()));
                WSACleanup();
                return -1;
            }
            else
              printf("Server: select() is OK.\n");

        }
        for (i = 0; i < NumSocks; i++)
       {
            if (FD_ISSET(ServSock[i], &SockSet))
           {
                FD_CLR(ServSock[i], &SockSet);
                break;
            }
        }

        if (SocketType == SOCK_STREAM)
       {
            SOCKET ConnSock;

            // Since this socket was returned by the select(), we know we
            // have a connection waiting and that this accept() won't block.
            ConnSock = accept(ServSock[i], (LPSOCKADDR)&From, &FromLen);
            if (ConnSock == INVALID_SOCKET)
```

```
            {
               fprintf(stderr, "Server: accept() failed with error %d: %s\n", WSAGetLastError(),
DecodeError(WSAGetLastError()));
                WSACleanup();
                return -1;
            }
            else
              printf("Server: accept() is OK.\n");

            if (getnameinfo((LPSOCKADDR)&From, FromLen, Hostname, sizeof(Hostname), NULL, 0,
NI_NUMERICHOST) != 0)
                strcpy(Hostname, "<unknown>");
            printf("\nAccepted connection from %s.\n", Hostname);

            // This sample server only handles connections sequentially.
            // To handle multiple connections simultaneously, a server
            // would likely want to launch another thread or process at
            // this point to handle each individual connection.  Alternatively,
            // it could keep a socket per connection and use select() on the
            // fd_set to determine which to read from next.
            //
            // Here we just loop until this connection terminates.
            while (1)
           {
                // We now read in data from the client.
                //  Because TCP does NOT maintain message boundaries, we may recv()
                // the client's data grouped differently than it was sent.
                //  Since all this server does is echo the data it
                // receives back to the client, we don't need to concern
                // ourselves about message boundaries.  But it does mean
                // that the message data we print for a particular recv()
                // below may contain more or less data than was contained
                // in a particular client send().
                AmountRead = recv(ConnSock, Buffer, sizeof(Buffer), 0);
                if (AmountRead == SOCKET_ERROR)
               {
                 fprintf(stderr, "Server: recv() failed with error %d: %s\n", WSAGetLastError(),
DecodeError(WSAGetLastError()));
                    closesocket(ConnSock);
                    break;
                }
                else
                  printf("Server: recv() is OK.\n");

                if (AmountRead == 0)
               {
                    printf("Server: Client closed the connection.\n");
                    closesocket(ConnSock);
                    break;
                }

                printf("Server: Received %d bytes from client: \"%.*s\"\n", AmountRead, AmountRead,
Buffer);
                printf("Server: Echoing the same data back to client...\n");

                RetVal = send(ConnSock, Buffer, AmountRead, 0);
                if (RetVal == SOCKET_ERROR)
               {
                    fprintf(stderr, "Server: send() failed: error %d: %s\n", WSAGetLastError(),
DecodeError(WSAGetLastError()));
                    closesocket(ConnSock);
                    break;
                }
               else
                  printf("Server: send() is OK.\n");
            }

        }
              else
              {
            // Since UDP maintains message boundaries, the amount of data
            // we get from a recvfrom() should match exactly the amount
            // of data the client sent in the corresponding sendto().
            AmountRead = recvfrom(ServSock[i], Buffer, sizeof(Buffer), 0, (LPSOCKADDR)&From, &FromLen);
            if (AmountRead == SOCKET_ERROR)
           {
             fprintf(stderr, "Server: recvfrom() failed with error %d: %s\n", WSAGetLastError(),
DecodeError(WSAGetLastError()));
                closesocket(ServSock[i]);
                break;
            }
            else
              printf("Server: recvfrom() is OK.\n");

            if (AmountRead == 0)
           {
                // This should never happen on an unconnected socket, but...
```

```
                printf("Server: recvfrom() returned zero, aborting...\n");
                closesocket(ServSock[i]);
                break;
            }
            else
              printf("Server: recvfrom() is OK, returning non-zero.\n");

            RetVal = getnameinfo((LPSOCKADDR)&From, FromLen, Hostname, sizeof(Hostname), NULL, 0,
NI_NUMERICHOST);
            if (RetVal != 0)
           {
                fprintf(stderr, "Server: getnameinfo() failed with error %d: %s\n", RetVal,
DecodeError(RetVal));
                strcpy(Hostname, "<unknown>");
            }
           else
              printf("Server: getnameinfo() is OK.\n");

            printf("Server: Received a %d byte datagram from %s: \"%.*s\"\n", AmountRead, Hostname,
AmountRead, Buffer);
            printf("Server: Echoing the same data back to client\n");

            RetVal = sendto(ServSock[i], Buffer, AmountRead, 0, (LPSOCKADDR)&From, FromLen);
            if (RetVal == SOCKET_ERROR)
           {
               fprintf(stderr, "Server: sendto() failed with error %d: %s\n", WSAGetLastError(),
DecodeError(WSAGetLastError()));
            }
           else
              printf("Server: sendto() is OK.\n");
        }
    }

    return 0;
}
```

Sample output:

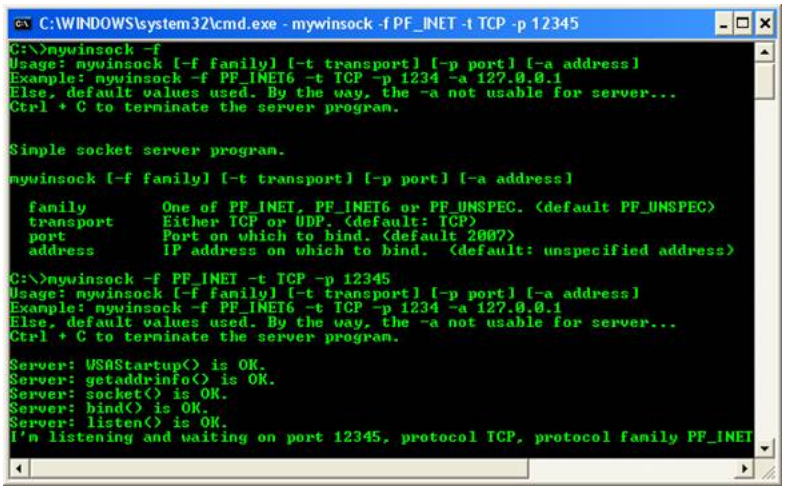


Figure 9

The IPv6 Client Program Example

And the client program for IPv6 example.

```
// Client for IPv6 enabled program example
#define WIN32_LEAN_AND_MEAN
#include <winsock2.h>
#include <ws2tcpip.h>
#ifndef IPPROTO_IPV6
// For IPv6
#include <tpipv6.h>
#endif
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

// This code assumes that at the transport level, the system only supports
// one stream protocol (TCP) and one datagram protocol (UDP).  Therefore,
// specifying a socket type of SOCK_STREAM is equivalent to specifying TCP
// and specifying a socket type of SOCK_DGRAM is equivalent to specifying UDP.
//
// Will use the loopback interface
#define DEFAULT_SERVER     NULL
```

```
// Accept either IPv4 or IPv6
#define DEFAULT_FAMILY     PF_UNSPEC
// TCP socket type
#define DEFAULT_SOCKTYPE   SOCK_STREAM
// Arbitrary, test port
#define DEFAULT_PORT       "2007"
// Number of "extra" bytes to send
#define DEFAULT_EXTRA      0
#define BUFFER_SIZE        65536

void Usage(char *ProgName)
{
    fprintf(stderr, "\nSimple socket IPv6 client program.\n");
    fprintf(stderr, "\n%s [-s server] [-f family] [-t transport] [-p port] [-b bytes] [-n number]\n\n",
ProgName);
    fprintf(stderr, "  server\tServer name or IP address.  (default: %s)\n",
            (DEFAULT_SERVER == NULL) ? "loopback address" : DEFAULT_SERVER);
    fprintf(stderr, "  family\tOne of PF_INET, PF_INET6 or PF_UNSPEC.  (default: %s)\n",
            (DEFAULT_FAMILY == PF_UNSPEC) ? "PF_UNSPEC" :
            ((DEFAULT_FAMILY == PF_INET) ? "PF_INET" : "PF_INET6"));
    fprintf(stderr, "  transport\tEither TCP or UDP.  (default: %s)\n",
            (DEFAULT_SOCKTYPE == SOCK_STREAM) ? "TCP" : "UDP");
    fprintf(stderr, "  port\t\tPort on which to connect.  (default: %s)\n",
            DEFAULT_PORT);
    fprintf(stderr, "  bytes\t\tBytes of extra data to send.  (default: %d)\n",
            DEFAULT_EXTRA);
    fprintf(stderr, "  number\tNumber of sends to perform.  (default: 1)\n");
    fprintf(stderr, "  (-n by itself makes client run in an infinite loop,");
    fprintf(stderr, " Hit Ctrl-C to terminate)\n");
    WSACleanup();
    exit(1);
}




LPSTR DecodeError(int ErrorCode)
{
    static char Message[1024];

    // If this program was multi-threaded, we'd want to use
    // FORMAT_MESSAGE_ALLOCATE_BUFFER instead of a static buffer here.
    // (And of course, free the buffer when we were done with it)

    FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_IGNORE_INSERTS |
                  FORMAT_MESSAGE_MAX_WIDTH_MASK,
                  NULL, ErrorCode, MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
                  (LPSTR)Message, 1024, NULL);
    return Message;
}

int ReceiveAndPrint(SOCKET ConnSocket, char *Buffer, int BufLen)
{
    int AmountRead;

    AmountRead = recv(ConnSocket, Buffer, BufLen, 0);
   if (AmountRead == SOCKET_ERROR)
    {
       fprintf(stderr, "Client: recv() failed with error %d: %s\n",  WSAGetLastError(),
DecodeError(WSAGetLastError()));
        closesocket(ConnSocket);
        WSACleanup();
        exit(1);
    }
    else
       printf("Client: recv() is OK.\n");

    // We are not likely to see this with UDP, since there is no 'connection' established.
    if (AmountRead == 0)
    {
        printf("Client: Server closed the connection...\n");
        closesocket(ConnSocket);
        WSACleanup();
        exit(0);
    }
    printf("Client: Received %d bytes from server: \"%.*s\"\n", AmountRead, AmountRead, Buffer);
    return AmountRead;
}

int main(int argc, char **argv)
{
    char Buffer[BUFFER_SIZE], AddrName[NI_MAXHOST];
    char *Server = DEFAULT_SERVER;
    int Family = DEFAULT_FAMILY;
    int SocketType = DEFAULT_SOCKTYPE;
    char *Port = DEFAULT_PORT;
    int i, RetVal, AddrLen, AmountToSend;
```

```
    int ExtraBytes = DEFAULT_EXTRA;
    unsigned int Iteration, MaxIterations = 1;
    BOOL RunForever = FALSE;
    WSADATA wsaData;
    ADDRINFO Hints, *AddrInfo, *AI;
    SOCKET ConnSocket;
    struct sockaddr_storage Addr;

       printf("Usage: %s [-s server] [-f family] [-t transport] [-p port] [-b bytes] [-n number]\n",
argv[0]);
       printf("Example: %s -s 127.0.0.1 -f PF_INET6 -t TCP -p 1234 -b 1024 -n 4\n", argv[0]);
       printf("Else, default values used.\n\n");
       if (argc > 1)
       {
        for (i = 1;i < argc; i++)
              {
            if (((argv[i][0] == '-') || (argv[i][0] == '/')) &&
                (argv[i][1] != 0) && (argv[i][2] == 0))
                    {
                switch(tolower(argv[i][1]))
                        {
                    case 'f':
                        if (!argv[i+1])
                            Usage(argv[0]);
                        if (!stricmp(argv[i+1], "PF_INET"))
                            Family = PF_INET;
                        else if (!stricmp(argv[i+1], "PF_INET6"))
                            Family = PF_INET6;
                        else if (!stricmp(argv[i+1], "PF_UNSPEC"))
                            Family = PF_UNSPEC;
                        else
                            Usage(argv[0]);
                        i++;
                        break;
                    case 't':
                        if (!argv[i+1])
                            Usage(argv[0]);
                        if (!stricmp(argv[i+1], "TCP"))
                            SocketType = SOCK_STREAM;
                        else if (!stricmp(argv[i+1], "UDP"))
                            SocketType = SOCK_DGRAM;
                        else
                            Usage(argv[0]);
                        i++;
                        break;
                    case 's':
                        if (argv[i+1])
                        {
                            if (argv[i+1][0] != '-')
                            {
                                Server = argv[++i];
                                break;
                            }
                        }
                        Usage(argv[0]);
                        break;
                    case 'p':
                        if (argv[i+1])
                        {
                            if (argv[i+1][0] != '-')
                            {
                                Port = argv[++i];
                                break;
                            }
                        }
                        Usage(argv[0]);
                        break;
                    case 'b':
                        if (argv[i+1])
                        {
                            if (argv[i+1][0] != '-')
                            {
                                ExtraBytes = atoi(argv[++i]);
                                if (ExtraBytes > sizeof(Buffer) - sizeof("Message #4294967295"))
                                    Usage(argv[0]);
                                break;
                            }
                        }
                        Usage(argv[0]);
                        break;
                    case 'n':
                        if (argv[i+1])
                        {
                            if (argv[i+1][0] != '-')
                            {
                                MaxIterations = atoi(argv[++i]);
                                break;
```

```
                        }
                    }
                    RunForever = TRUE;
                    break;
                default:
                    Usage(argv[0]);
                    break;
            }
        }
        else
            Usage(argv[0]);
    }
}
// Request for Winsock version 2.2.
if ((RetVal = WSAStartup(MAKEWORD(2, 2), &wsaData)) != 0)
{
    fprintf(stderr, "Client: WSAStartup() failed with error %d: %s\n", RetVal, DecodeError(RetVal));
    WSACleanup();
    return -1;
}
else
   printf("Client: WSAStartup() is OK.\n");
// By not setting the AI_PASSIVE flag in the hints to getaddrinfo, we're
// indicating that we intend to use the resulting address(es) to connect
// to a service.  This means that when the Server parameter is NULL,
// getaddrinfo will return one entry per allowed protocol family
// containing the loopback address for that family.
memset(&Hints, 0, sizeof(Hints));
Hints.ai_family = Family;
Hints.ai_socktype = SocketType;
RetVal = getaddrinfo(Server, Port, &Hints, &AddrInfo);
if (RetVal != 0)
{
    fprintf(stderr, "Client: Cannot resolve address [%s] and port [%s], error %d: %s\n", Server, Port,
RetVal, gai_strerror(RetVal));
    WSACleanup();
    return -1;
}
else
   printf("Client: getaddrinfo() is OK, name resolved.\n");
// Try each address getaddrinfo returned, until we find one to which
// we can successfully connect.
for (AI = AddrInfo; AI != NULL; AI = AI->ai_next)
{
    // Open a socket with the correct address family for this address.
    ConnSocket = socket(AI->ai_family, AI->ai_socktype, AI->ai_protocol);
    if (ConnSocket == INVALID_SOCKET)
    {
        fprintf(stderr,"Client: Error Opening socket, error %d: %s\n", WSAGetLastError(),
DecodeError(WSAGetLastError()));
        continue;
    }
   else
      printf("Client: socket() is OK.\n");
    // Notice that nothing in this code is specific to whether we
    // are using UDP or TCP.
    //
    // When connect() is called on a datagram socket, it does not
    // actually establish the connection as a stream (TCP) socket
    // would. Instead, TCP/IP establishes the remote half of the
    // (LocalIPAddress, LocalPort, RemoteIP, RemotePort) mapping.
    // This enables us to use send() and recv() on datagram sockets,
    // instead of recvfrom() and sendto().
    printf("Client: Attempting to connect to: %s\n", Server ? Server : "localhost");
    if (connect(ConnSocket, AI->ai_addr, int(AI->ai_addrlen)) != SOCKET_ERROR)
        break;
    else
       printf("Client: connect() is OK.\n");

    i = WSAGetLastError();
    if (getnameinfo(AI->ai_addr, int(AI->ai_addrlen), AddrName, sizeof(AddrName), NULL, 0,
NI_NUMERICHOST) != 0)
    strcpy(AddrName, "<unknown>");
    fprintf(stderr, "Client: connect() to %s failed with error %d: %s\n", AddrName, i, DecodeError(i));
    closesocket(ConnSocket);
}

if (AI == NULL)
{
    fprintf(stderr, "Client: Fatal error: unable to connect to the server.\n");
    WSACleanup();
    return -1;
}
// This demonstrates how to determine to where a socket is connected.
AddrLen = sizeof(Addr);
if (getpeername(ConnSocket, (LPSOCKADDR)&Addr, &AddrLen) == SOCKET_ERROR)
{
```

```
        fprintf(stderr, "Client: getpeername() failed with error %d: %s\n", WSAGetLastError(),
DecodeError(WSAGetLastError()));
    }
    else
    {
        if (getnameinfo((LPSOCKADDR)&Addr, AddrLen, AddrName, sizeof(AddrName), NULL, 0, NI_NUMERICHOST) !=
0)
        strcpy(AddrName, "<unknown>");
        printf("Client: Connected to %s, port %d, protocol %s, protocol family %s\n", AddrName,
ntohs(SS_PORT(&Addr)), (AI->ai_socktype == SOCK_STREAM) ? "TCP" : "UDP", (AI->ai_family == PF_INET) ?
"PF_INET" : "PF_INET6");
    }
    // We are done with the address info chain, so we can free it.
    freeaddrinfo(AddrInfo);
    // Find out what local address and port the system picked for us.
    AddrLen = sizeof(Addr);
    if (getsockname(ConnSocket, (LPSOCKADDR)&Addr, &AddrLen) == SOCKET_ERROR)
    {
        fprintf(stderr, "Client: getsockname() failed with error %d: %s\n", WSAGetLastError(),
DecodeError(WSAGetLastError()));
    }
    else
    {
       printf("Client: getsockname() is OK.\n");
        if (getnameinfo((LPSOCKADDR)&Addr, AddrLen, AddrName, sizeof(AddrName), NULL, 0, NI_NUMERICHOST) !=
0)
            strcpy(AddrName, "<unknown>");
        printf("Client: Using local address %s, port %d\n", AddrName, ntohs(SS_PORT(&Addr)));
    }
    // Send and receive in a loop for the requested number of iterations.
    for (Iteration = 0; RunForever || Iteration < MaxIterations; Iteration++)
    {
        // compose a message to send.
        AmountToSend = sprintf(Buffer, "This is message #%u", Iteration + 1);
        for (i = 0; i < ExtraBytes; i++)
       {
            Buffer[AmountToSend++] = (char)((i & 0x3f) + 0x20);
        }
        // Send the message.  Since we are using a blocking socket, this
        // call shouldn't return until it's able to send the entire amount.
        RetVal = send(ConnSocket, Buffer, AmountToSend, 0);
        if (RetVal == SOCKET_ERROR)
       {
            fprintf(stderr, "Client: send() failed with error %d: %s\n", WSAGetLastError(),
DecodeError(WSAGetLastError()));
            WSACleanup();
            return -1;
        }
       else
          printf("Client: send() is OK.\n");

        printf("Client: Sent %d bytes (out of %d bytes) of data: \"%.*s\"\n", RetVal, AmountToSend,
AmountToSend, Buffer);
        // Clear buffer just to prove we're really receiving something.
        memset(Buffer, 0, sizeof(Buffer));
        // Receive and print server's reply.
        ReceiveAndPrint(ConnSocket, Buffer, sizeof(Buffer));
    }
    // Tell system we're done sending.
    printf("Client: Sending done...\n");
    shutdown(ConnSocket, SD_SEND);
    // Since TCP does not preserve message boundaries, there may still
    // be more data arriving from the server.  So we continue to receive
    // data until the server closes the connection.
    if (SocketType == SOCK_STREAM)
        while (ReceiveAndPrint(ConnSocket, Buffer, sizeof(Buffer)) != 0)
                ;
    closesocket(ConnSocket);
    WSACleanup();
    return 0;
}
```

Testing the IPv6 Client-server Programs

Run the previous server program example, then run the client program. The following is the client program's output.

```
C:\>myclient -s 127.0.0.1 -f PF_INET -t TCP -p 12345 -b 32 -n 4
Usage: myclient [-s server] [-f family] [-t transport] [-p port] [-b bytes] [-n number]
Example: myclient -s 127.0.0.1 -f PF_INET6 -t TCP -p 1234 -b 1024 -n 4
Else, default values used.

Client: WSAStartup() is OK.
Client: getaddrinfo() is OK, name resolved.
Client: socket() is OK.
```

```
Client: Attempting to connect to: 127.0.0.1
Client: Connected to 127.0.0.1, port 12345, protocol TCP, protocol family PF_INET
Client: getsockname() is OK.
Client: Using local address 127.0.0.1, port 1032
Client: send() is OK.
Client: Sent 50 bytes (out of 50 bytes) of data: "This is message #1 !"#$%&'()*+,-./0123456789:;<=>?"
Client: recv() is OK.
Client: Received 50 bytes from server: "This is message #1 !"#$%&'()*+,-./0123456789:;<=>?"
Client: send() is OK.
Client: Sent 50 bytes (out of 50 bytes) of data: "This is message #2 !"#$%&'()*+,-./0123456789:;<=>?"
Client: recv() is OK.
Client: Received 50 bytes from server: "This is message #2 !"#$%&'()*+,-./0123456789:;<=>?"
Client: send() is OK.
Client: Sent 50 bytes (out of 50 bytes) of data: "This is message #3 !"#$%&'()*+,-./0123456789:;<=>?"
Client: recv() is OK.
Client: Received 50 bytes from server: "This is message #3 !"#$%&'()*+,-./0123456789:;<=>?"
Client: send() is OK.
Client: Sent 50 bytes (out of 50 bytes) of data: "This is message #4 !"#$%&'()*+,-./0123456789:;<=>?"
Client: recv() is OK.
Client: Received 50 bytes from server: "This is message #4 !"#$%&'()*+,-./0123456789:;<=>?"
Client: Sending done...
Client: recv() is OK.
Client: Server closed the connection...

C:\>
```

Finally the previous server output.

```
C:\>mywinsock -f
Usage: mywinsock [-f family] [-t transport] [-p port] [-a address]
Example: mywinsock -f PF_INET6 -t TCP -p 1234 -a 127.0.0.1
Else, default values used. By the way, the -a not usable for server...
Ctrl + C to terminate the server program.

Simple socket server program.

mywinsock [-f family] [-t transport] [-p port] [-a address]

  family        One of PF_INET, PF_INET6 or PF_UNSPEC. (default PF_UNSPEC)
  transport     Either TCP or UDP. (default: TCP)
  port          Port on which to bind. (default 2007)
  address       IP address on which to bind.  (default: unspecified address)

C:\>mywinsock -f PF_INET -t TCP -p 12345
Usage: mywinsock [-f family] [-t transport] [-p port] [-a address]
Example: mywinsock -f PF_INET6 -t TCP -p 1234 -a 127.0.0.1
Else, default values used. By the way, the -a not usable for server...
Ctrl + C to terminate the server program.

Server: WSAStartup() is OK.
Server: getaddrinfo() is OK.
Server: socket() is OK.
Server: bind() is OK.
Server: listen() is OK.
I'm listening and waiting on port 12345, protocol TCP, protocol family PF_INET
Server: select() is OK.
Server: accept() is OK.

Accepted connection from 127.0.0.1.
Server: recv() is OK.
Server: Received 50 bytes from client: "This is message #1 !"#$%&'()*+,-./0123456789:;<=>?"
Server: Echoing the same data back to client...
Server: send() is OK.
Server: recv() is OK.
Server: Received 50 bytes from client: "This is message #2 !"#$%&'()*+,-./0123456789:;<=>?"
Server: Echoing the same data back to client...
Server: send() is OK.
Server: recv() is OK.
Server: Received 50 bytes from client: "This is message #3 !"#$%&'()*+,-./0123456789:;<=>?"
Server: Echoing the same data back to client...
Server: send() is OK.
Server: recv() is OK.
Server: Received 50 bytes from client: "This is message #4 !"#$%&'()*+,-./0123456789:;<=>?"
Server: Echoing the same data back to client...
Server: send() is OK.
Server: recv() is OK.
Server: Client closed the connection.
^C
C:\>
```

7. EXERCISE

1 Programming a Simple WebServer

[1] http://diendan.congdongcviet.com/threads/t4033::simple-webserver-cpp-ma-nguon-webserver-don-gian.cpp

[2] http://diendan.congdongcviet.com/threads/t7362::lap-trinh-mang-thu-vien-winsock-tren-visual-cpp.cpp

Programming a winsock sniffer

[1] https://copynull.tistory.com/108

[2] https://www.binarytides.com/packet-sniffer-code-in-c-using-winsock/

3 Programming an IoT webserver :

[1] https://www.hdhprojects.nl/2018/02/08/programming-an-iot-webserver/