



Client / Server Programming with TCP/IP Sockets

Author: Rajinder Yadav

Date: Sept 9, 2007

Revision: Mar 11, 2008

Web: <http://devmentor.org>

Email: rajinder@devmentor.org

Table of Content

Networks.....	2
Diagram 1 – Communication Link.....	2
IPv4 Internet Protocol.....	2
IP Address.....	2
IP Classes.....	3
Subnets.....	3
IP and the Hardware.....	4
IP and DNS.....	4
loopback.....	5
IP from hostname.....	5
hostname from IP.....	5
hostent structure.....	5
TCP Transmission Control Protocol.....	6
Diagram 3 – Network Hop Topology.....	6
TCP Window.....	6
Byte Ordering.....	7
Little-Endian: the LSB (Least Significant Byte) is at the lowest address.....	7
Big-Endian: The MSB (Most Significant Byte) is at the lowest address.....	7
Diagram 4 – Address Byte Order.....	7
htons() and htonl().....	7
ntohs() and ntohl().....	7
Sockets.....	7
TCP Ports.....	8
Table 1 – Well Known Ports.....	8
Opening and closing a socket.....	8
Initializing Winsock.....	9
Socket Address Structure.....	9
TCP Server.....	10
Naming the socket.....	11
TCP Client.....	13
TCP State Diagram.....	14
Connection Termination.....	14
Running The Programs.....	16
Network Utilities.....	17
ping.....	17
netstat.....	17
Server Socket Code.....	18
Client Socket Code.....	22

Introduction

In this article we will look at how to program using *sockets* by implementing an *echo server* along with a *client* that we will use to send and receive string messages. I will start off by giving a quick introduction to TCP/IP fundamentals and then explain how *sockets* fit into the picture. I am going to assume you already understand basic network concepts. When you get done reading this article you should be armed with sufficient information to be able to investigate other concepts in more detail on your own. Sample code can be downloaded for this article from my website for Windows and Linux. I have also included all the Winsock source code in print at the end of the article for others.

Networks

Most network applications can be divided into two pieces: a *client* and a *server*. A client is the side that initiates the communication process, whereas the server responds to incoming client requests.



Diagram 1 – Communication Link

There are numerous network protocols, such as Netbios, RPC (Remote Procedure Call), DCOM, Pipes, IPC (Inter-process Communication) that can be used for the Communication Link. We will only look at TCP/IP here. In particular we will look at sockets IPv4 since this is widely implemented by many socket vendors.

IPv4 Internet Protocol

The current version for IP supported by many modern networks is version 4, the next generation of IP is version 6 (**IPv6**) which is not widely supported as of this writing. IP is both a network addressing protocol and a network transport protocol. IP is a connection-less protocol that provides unreliable service for data communication. Most of the properties of data communication such as transmission reliability, flow control and error checking are provided by TCP which we will look at shortly.

Most people know IP in its basic form as a quad *dotted-numerical* string, such as “192.168.1.1”. Each integer value separated by a dot can have a value from 0 to 255 (8 bits). Thus IPv4 is a 32 bit unsigned integer values.

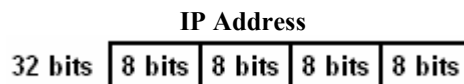


Diagram 2a – IP Address

IP Classes

The IP address is made of a network address and a host address. There can be many sub-networks connect together, so a network address help routers to redirect data packet to the proper destination network, from there the data packet is sent to the final destination host PC. The 4 IP *classes* are:

Class	Leftmost bit	Start Address	End Address
A	0xxx	0.0.0.0	127.255.255.255
B	10xx	128.0.0.0	191.255.255.255
C	110x	192.0.0.0	223.255.255.255
D	1110	224.0.0.0	223.255.255.255
E	1111	240.0.0.0	255.255.255.255

Class A network.local.local.local (small network, large hosts)
Class B network.network.local.local (medium network, medium hosts)
Class C network.network.network.local (large network, small hosts)
Class D network.network.network.network (multicast to many hosts)
Class E reserved

(*) A special type of IP address is the limited broadcast address 255.255.255.255

(*) IP Mapping: Class A, B, C 1-to-1, Class D is 1-to-many

Subnets

An IP address can have the host address subdivided into a subnet part and a host part using a subnet mask. The subnet mask is also a 32bit value, the bits for the network and subnet will be set to '1', this way from the mask and IP class we can determine the network address, the subnet address and the host number.



Diagram 2b – IP Address with Subnet

There IP address can be classified as unicast, broadcast and multicast. A unicast address has a 1-to-1 relationship. A broadcast can apply to: a) all hosts on a network, b) all hosts on a subnet, and c) all hosts on all subnets. For multicast, a host needs to belong to the multicast group in order to receive a packet. The main thing to note is that a broadcast or multicast packet is never forwarded by a router/gateway outside the network.

IP and the Hardware

IP is used to identify a PC on the network. This is done for us at the hardware level by the NIC (Network Interface Card) like a PC's Ethernet card or a Router. A machine NIC or Ethernet uses ARP (Address Resolution Protocol) to convert an IP address into a network address that can be understood by routers and gateways. Likewise the hardware layer use RARP (Reverse Address Resolution Protocol) to convert a hardware network address at the MAC (Media Access Control) level into an IP address. As data packets move around the network, the hardware layer (router) is checking if a packet is meant for it to process by checking the MAC address. If it's not the data packet is transmitted down the line. If the data packet is meant for a host in the network, then the IP address is checked using a lookup table to determine which host PC to send the data packet off to in the network.

We really don't need to be concerned with underlying details as all this is handled for us.

IP and DNS

Most people don't use IP directly, it's not an easy way to remember addresses, so to help humans the DNS (Domain Name System) maps a hostname strings like "yadav.shorturl.com" into an IP address. If you're developing any type of network application it's better to use the DSN name format to communicate with another computer. DSN mapping can be easily changed (in the router table) thus allowing one to redirect network traffic to another destination.

Host File

On your local PC the DNS mapping entries are found in the host file. On Windows NT, 2K, XP the file "hosts" is located at:

```
%WINDOWS%\system32\drivers\etc\
```

The host files (shown in blue) contains a space separated IP address and hostname. All this info ends up in the router table, where network traffic control occurs.

```
127.0.0.1    localhost
192.168.1.106 freedom.home.com
192.168.1.105 download.bigdaddy.com
192.168.1.100 sal.home.com
```

loopback

Note: the *loopback* address of “127.0.0.1” also known as “**localhost**” is the IP address used when communicating with other process running on the same PC. This is how we will test our client and server application, they will run on the same “local” host PC.

IP from hostname

The `gethostbyname` function retrieves host information corresponding to a host name from a host database.

```
struct hostent* FAR gethostbyname(  
    const char* name  
);
```


hostname from IP

The `gethostbyaddr` function retrieves the host information corresponding to a network address.

```
struct HOSTENT* FAR gethostbyaddr(  
    const char* addr,  
    int len,  
    int type  
);
```

hostent structure

[msdn] The `hostent` structure is used by functions to store information about a given host, such as host name, IP address. An application should never attempt to modify this structure or to free any of its components.

 Furthermore, only one copy of the `hostent` structure is allocated per thread, and an application should therefore copy any information that it needs before issuing any other Windows Sockets API calls.

```
typedef struct hostent {  
    char FAR* h_name;  
    char FAR FAR** h_aliases;  
    short h_addrtype;  
    short h_length;  
    char FAR FAR** h_addr_list;  
} hostent;
```

TCP Transmission Control Protocol

Although TCP can be implemented to work over any transport protocol, it's usually synonymous with IP. TCP is a connection-oriented stream protocol (like a telephone call). TCP communication happens using a handshake process, where each data that is sent is acknowledge by the recipient within the time of TCP's timer value. TCP provides many services such as data reliability, error checking, and flow control. If a data packet is corrupt or lost (not acknowledged), TCP will retransmitted the data from the client side automatically. Because the route a packet takes can be many, one packet may arrive before the one sent earlier. As data packets arrive, it is the job of TCP to assemble the packets into the proper order. This is shown below with a factious network topology layout, where the data packet takes (n) number of hops to get from the source to the destination. On a bigger network like the *Internet*, there are many routes a data packet can take to arrive at its final destination.

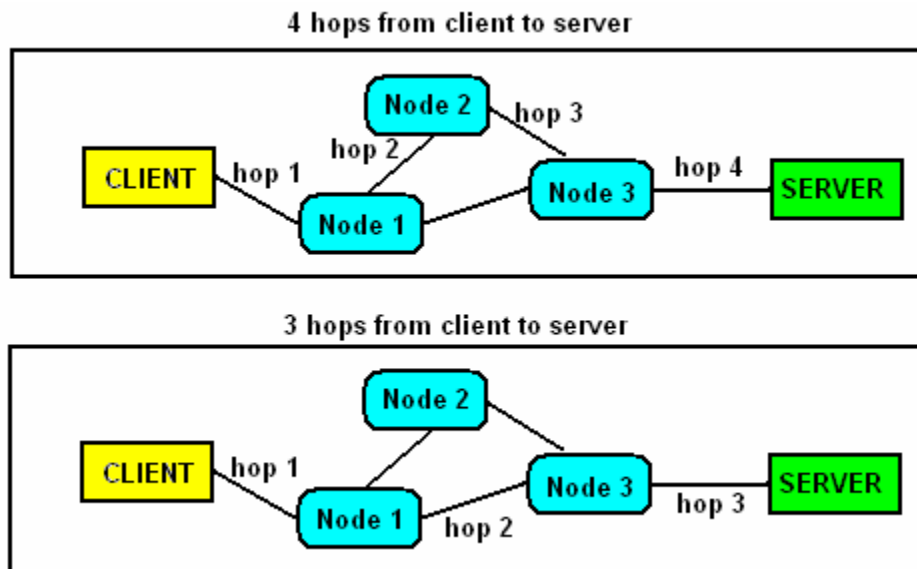


Diagram 3 – Network Hop Topology

TCP Window

Any duplicate data packet is silently dropped with no acknowledgement. TCP controls the flow of transmission by using a “window” that can grow or shrink based on how responsive the (next-hop) node is. If a lot of packets are getting dropped because the receiver's buffer is full, TCP will slow down the rate of transmission by decreasing the size of the “window”. Usually TCP will negotiate with the receiver for such things as the maximum transmission unit. If the sizes are different, such that the receiver node accepts a smaller sized packet, the out-bound packet will be fragmented by TCP. Again, the data packet “segments” can arrive at different times by taking different routes. So it's the job of TCP at the destination end to reassemble the original data packet as the segments arrive. The data is placed into a larger buffer by TCP to be read by the application. Data is streamed out from the client side and streamed in at the server side. This is why TCP is called a stream bases protocol, it's work just like a file I/O read and write operation, which is what provides synchronous data access.

Byte Ordering

There are two types of memory byte ordering in use today that are very much machine dependent. They are known as *little-endian* and *big-endian*, because of this we have to be very careful how we interpret *numerical data*. If we do not take into account the endiannes, the *numerical data* we read will be corrupt.

Little-Endian: the LSB (Least Significant Byte) is at the lowest address.

Big-Endian: The MSB (Most Significant Byte) is at the lowest address.

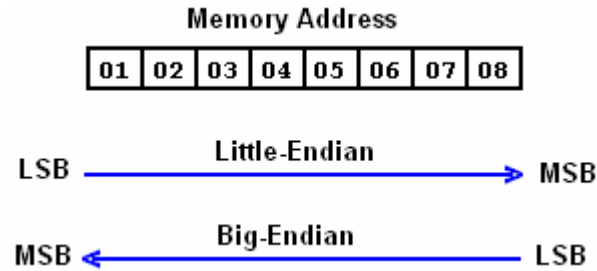


Diagram 4 – Address Byte Order

Generally when working with numeric data, one needs to convert from machine (host) byte order to network byte order when sending data (write-op), and then from network byte order to machine byte order when retrieving data (read-op). The APIs to make the conversion are:

htons() and htonl()

```
// host to network
uint16_t htons ( uint16_t host16bitvalue );
uint32_t htonl ( uint32_t host32bitvalue );
```

ntohs() and ntohl()

```
// network to host
uint16_t ntohs ( uint16_t net16bitvalue );
uint32_t ntohl ( uint32_t net32bitvalue );
```

Note: network byte order is in Big-Endian, CPU based on the x86 architecture use Little-Endian byte order.

Sockets

We are now ready to talk about what a socket is. A socket is made up of 3 identifying properties: **Protocol Family, IP Address, Port Number**

For TCP/IP Sockets:

- The protocol family is **AF_INET** (Address Family Internet)
- The IP Address identifies a host/service machine on the network
- Port defines the *Service* on the machine we're communicating to/from

TCP Ports

The port numbers from 0 to 255 are well-known ports, and the use of these port numbers in your application is highly discouraged. Many well-known services you use have assigned port numbers in this range.

Service Name	Port Number
ftp	21
telenet	23
www-http	80
irc	194

Table 1 – Well Known Ports

In recent years the range for assigned ports managed by IANA (Internet Assigned Numbers Authority) has been expanded to the range of 0 – 1023. To get the most recent listing of assigned port numbers, you can view the latest RFC 1700 at:

<http://www.faqs.org/rfcs/rfc1700.html>

In order for 2 machines to be able to communicate they need to be using the same type of sockets, both have to be TCP or UDP. On Windows, the socket definition is defined in the header file <winsock.h> or <winsock2.h>. Our program will be using Winsock 2.2 so we will need to include <winsock2.h> and link with **WS2_32.lib**

Opening and closing a socket

To create a TCP/IP socket, we use the **socket()** API.

```
SOCKET hSock = socket( AF_INET, SOCK_STREAM, IPPROTO_IP );
```

If the socket API fails, it will return a value of **INVALID_SOCKET**, otherwise it will return a descriptor value that we will need to use when referring to this socket. Once we are done with the socket we must remember to call the **closesocket()** API.

```
closesocket( hSock );
```

Note: On Unix/Linux the return type for *socket()* is an int, while the API to close the socket is *close(socket_descriptor)*;

Before we can begin to use any socket API in Windows we need to initialize the socket library, this is done by making a call to **WSAStartup()**. This step is not required on Unix/Linux.

Initializing Winsock

```
// Initialize WinSock2.2 DLL
// low-word = major, hi-word = minor
WSADATA wsaData = {0};
WORD wVer = MAKEWORD(2,2);
int nRet = WSASStartup( wVer, &wsaData );
```

Before we exit our program we need to release the Winsock DLL with the following call.

```
// Release WinSock DLL
WSACleanup();
```

With the basics out of the way, the process of preparing the client and server application is similar, but differ slightly in their steps. We will talk about how to write the server code first.

Socket Address Structure

For IPv4 the socket structure is defined as:

```
struct sockaddr
{
    u_short sa_family;    /* address family */
    char    sa_data[14]; /* up to 14 bytes of direct address */
};
```

This is the generic structure that most socket APIs accept, but the structure you will work with is **sockaddr_in** (socket address internet).

```
struct sockaddr_in
{
    short    sin_family;
    u_short  sin_port;
    struct  in_addr sin_addr;
    char    sin_zero[8];
};
```

Note: **sockaddr_in** and the more generic **sockaddr** struct are the same size. Padding is used to maintain the size by `sin_zero[8]`. You will need to typecast between the two in your program.

struct **in_addr** found inside **sockaddr_in** is a union defined as:

```
struct in_addr
{
    union
    {
        struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;
        struct { u_short s_w1,s_w2; } S_un_w;
        u_long S_addr;
    } S_un;

#define s_addr S_un.S_addr
#define s_host S_un.S_un_b.s_b2
#define s_net S_un.S_un_b.s_b1
#define s_imp S_un.S_un_w.s_w2
#define s_impno S_un.S_un_b.s_b4
#define s_lh S_un.S_un_b.s_b3
};
```

This structure holds the IP address which can be accessed in many ways. You can use the **inet_addr()** API to convert a IP *dotted-numerical* string into a 32bit IP address and assign it to the **s_addr** member. The use of this API is shown when we discuss the client code. To do an opposite conversion from network to IP string, use the **inet_ntoa()** API.

TCP Server

The steps to get a server up and running are shown below (read from top to bottom). This is how our sample code is written, so it's a good idea to get familiar with the process.

```
    socket( )
    bind( )
+---->listen( )
|
|    accept( )
|    (block until connection from client )
|    read( )
|    write( )
+-----close( )
    close( )
```

1. Create a server socket
2. Name the socket
3. Prepare the socket to listen
4. Wait for a request to connect, a new client socket is created here
5. Read data sent from client
6. Send data back to client
7. Close client socket
8. Loop back if not told to exit
9. Close server socket is exit command given by client

Naming the socket

When we prepare the socket for the server, we use **INADDR_ANY** for the IP address to tell the TCP stack to assign an IP address to listen on for incoming connection requests. Do not assign a hard-coded value, otherwise the program will only run on the server defined by the IP address, and if the server is multi-homed then we are restricting it to listen on only one IP address rather than allow the administrator to set the default IP address.

```
// name socket
sockaddr_in saListen = {0};

saListen.sin_family      = PF_INET;
saListen.sin_port        = htons( 10000 );
saListen.sin_addr.s_addr = htonl( INADDR_ANY );
```

Once we have initialize the **sockaddr_in** struct, we need to name the socket by calling the **bind()** API.

```
bind( hSock, (sockaddr*)&saListen, sizeof(sockaddr) );
```

Notice how we are required to typecast the socket address to (**sockaddr***). Here **hSock** is the server socket we created earlier and this binds (names) the socket with the information provided in the socket structure. After the socket has been named, we then wait for incoming connection requests by making the next 2 calls.

```
listen( hSock, 5 );

sockaddr_in saClient = {0};
int nSALen = sizeof( sockaddr );
SOCKET hClient = accept( hSock, (sockaddr*)&saClient, &nSALen );
```

The values of '5' passed to **listen()** is the *backlog* of connection request that will get queued waiting to be processed. Once this limit is exceeded, all new calls will fail. In the call to **accept()**, we do not need to initialize **saClient**, the **sockaddr** struct because when this calls returns it will fill **saClient** with the name of the client's socket.

The sample echo server provided is a single thread (iterative) application, it can only handle and process one connection at a time. In a real world application, one would general use *multithreading* to handle incoming client requests. The *primary* thread would listen to incoming calls and block on the call to **accept()**. When a connection request came in, the main thread would be given the client's socket *descriptor*. At this point a new thread should be created and the client socket passed to it for processing the request. The main thread would then loop back and listen for the next connection.

When a client connection has been established, data is read from the TCP buffer by making a call to `recv()`. This function returns the number of bytes read.

```
nSent = recv( hClient, wzRec, nSize, 0 );
```

The first parameter is the socket to the connected client, the second parameter is a text buffer, followed by its size, and finally *flags* on how `recv()` behaves. The *flags* value is normally zero. Other values for the *flag* that can be ORed are:

MSG_PEEK : Copy data to buffer, but do not remove from TCP buffer.

MSG_OOB : Process out of bound data


MSG_WAITALL : Wait for the buffer to be completely filled

Note: the call to `recv()` will only return the data that has arrived, this means the call can return before the entire data has been received into the TCP buffer. So the best way to write a data fetch routine is by using a fixed buffer size that is able to hold the largest transmitted data set. Then to read using a loop that waits till all the data has arrived before handing off the read buffer.

```
// process data
char wzRec[512] = {0};
int nLeft = 512;
int iPos = 0;
int nData = 0;
do
{
    nData = recv( hClient, &wzRec[iPos], nLeft, 0 );

    nLeft -= nData;
    iPos += nData;
} while( nLeft > 0 );
```

Likewise we do the same when sending data with `send()`, the parameter values are the same as they are for `recv()`.

 Just because `send()` return successfully saying it sent the data, this does not mean the data was placed on the network. It simply means the TCP buffer had enough room to copy the data. There is a TCP option **TCP_NODELAY** that will cause the data to appear on the network immediately, but its use is only for specialized applications and should be avoided.

Once we are done, we need to close the client's socket and then close the server socket. In the sample program of the echo server, the server loops back into listen mode until a string containing "**!shutdown**" is received which tells the server to stop listening and to shutdown.

TCP Client

Now let's take a look at what steps the client needs to take in order to communicate with the server.

```
socket( )
connect( )
write( )
read( )
close( )
```

1. Create a socket with the server IP address
2. Connect to the server, this step also names the socket
3. Send data to the server
4. Read data returned (echoed) back from the server
5. Close the socket

The initialization of the **sockaddr_in** structure is different for the client. With the client the socket address needs to assign the *port* number along with the IP address of the server. Since we're testing on the *localhost*, we hard-core the IP address of "127.0.0.1"

```
sockaddr_in saServer = {0};

saServer.sin_family      = PF_INET;
saServer.sin_port        = htons( 10000 );
saServer.sin_addr.s_addr = inet_addr( "127.0.0.1" );
```

Notice the use of **htons()** when setting the *port* number. The call to **inet_addr()** converts a *dotted-numerical* string into a 32bit IP value that gets assigned to the **s_addr** member of the **sin_addr** struct.

We have now covered how both the client and server code is written, lets take a quick look at how to use the both the programs.

TCP State Diagram

Connection Establishment

TCP uses a 3 way handshake. A Server makes a passive open by call bind(). The Client initiates an active open by calling connect(). For the connection to be established, the client send a SYN to the server. The Server replies with SYN/ACK and finally the Client replied with ACK.

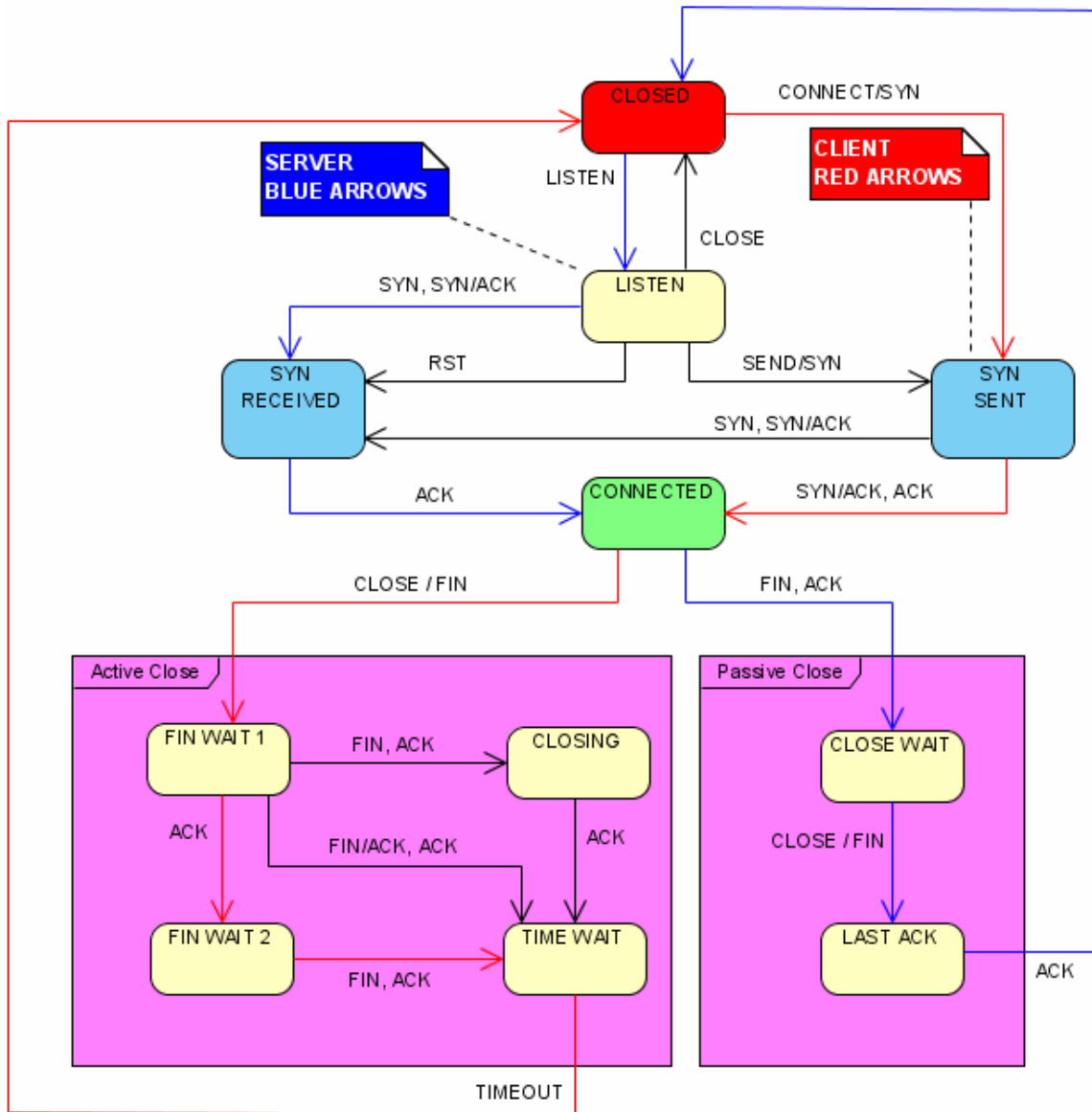


Diagram-5 Created using Visual Paradigm

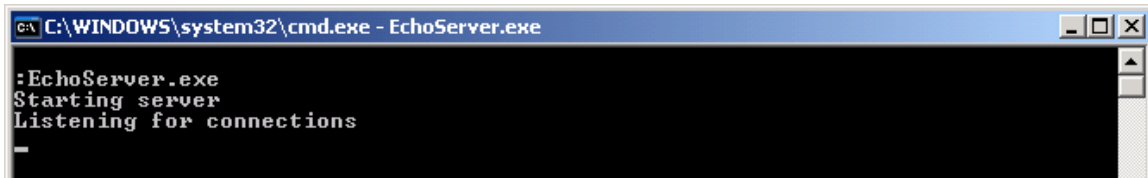
Connection Termination

TCP uses a 4 way handshake to close the connection. When an endpoint wants to close, it send out a FIN, the other side then replied with an ACK. A connection is “half-open” when one side has close the connection. The other side is still free to send. A situation can occur where one side closes the connection and then reopens it immediately, so any lost packets that now arrive will not belong to this “new” connection and thus TCP we need to insure these packets do not get mixed in with the new packets. So the “TIME WAIT” state allows a connection to remain open long enough for such packets to be removed from

the network. This state usually lasts for about 2 times the “round-trip”, some implementation hardcode the default value to be anywhere from 30 to 120 seconds. You can use the **netstat** utility to see TCP/IP states.

Running The Programs

Bring up two command prompts and go the folder were the client and server executable are each located. In one of the shell type, "**EchoServer.exe**" to start the server. You should see the following output:



```
C:\WINDOWS\system32\cmd.exe - EchoServer.exe
:EchoServer.exe
Starting server
Listening for connections
_
```

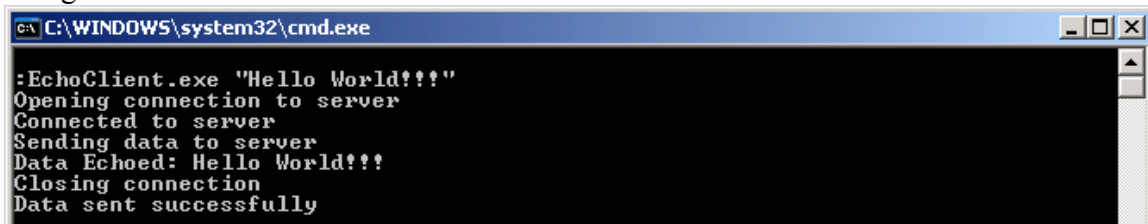
In the other shell type: **EchoClient.exe "Hello World!!!"**

At this point the server should display that a connection has been made follow by the data sent.



```
C:\WINDOWS\system32\cmd.exe - EchoServer.exe
:EchoServer.exe
Starting server
Listening for connections
Connection established
Data Recieved: Hello World!!!
Listening for connections
_
```

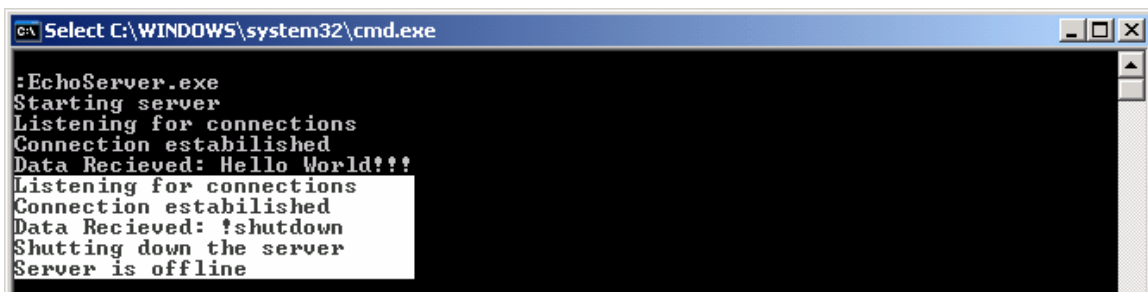
The client shell should state that a connection has been made and what the echo message string is.



```
C:\WINDOWS\system32\cmd.exe
:EchoClient.exe "Hello World!!!"
Opening connection to server
Connected to server
Sending data to server
Data Echoed: Hello World!!!
Closing connection
Data sent successfully
```

If we type in the client shell: **EchoClient.exe !shutdown**

The server should now shutdown and exit. Highlighted is the output from the server of the second client connection.



```
Select C:\WINDOWS\system32\cmd.exe
:EchoServer.exe
Starting server
Listening for connections
Connection established
Data Recieved: Hello World!!!
Listening for connections
Connection established
Data Recieved: !shutdown
Shutting down the server
Server is offline
```

The source code on the following pages were colored using Carlos Aguilar Mares "CodeColorizer" tool, which can be found at <http://www.carlosag.net/>

Network Utilities

Some of the basic tools you will need to know about when working with networking applications are: **ping** and **netstat**

ping

This tool lets you determine if a destination node is reachable. If you're application client application is unable to connect to a server located on another machine over the network you would first try to ping the node to see if the connection was alright. Below is an example of me pinging my router, it shows that all pings were acknowledged, along with the time it took.

```
Pinging 192.168.1.1 with 32 bytes of data:
Reply from 192.168.1.1: bytes=32 time<1ms TTL=127
Reply from 192.168.1.1: bytes=32 time<1ms TTL=127
Reply from 192.168.1.1: bytes=32 time<1ms TTL=127
Reply from 192.168.1.1: bytes=32 time<1ms TTL=127
Ping statistics for 192.168.1.1:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 0ms, Maximum = 0ms, Average = 0ms
```

netstat

This utility allows you to view protocol statistics and current TCP/IP connection states. I like to use it with the "-na" option to show "all network addresses". If we were you make a call to netstate right after we started the server, here is what one might expect to see.

```
C:\Documents and Settings\Developer>netstat -na
Active Connections

```

Proto	Local Address	Foreign Address	State
TCP	0.0.0.0:135	0.0.0.0:0	LISTENING
TCP	0.0.0.0:445	0.0.0.0:0	LISTENING
TCP	0.0.0.0:523	0.0.0.0:0	LISTENING
TCP	0.0.0.0:2869	0.0.0.0:0	LISTENING
TCP	0.0.0.0:6789	0.0.0.0:0	LISTENING
TCP	0.0.0.0:10000	0.0.0.0:0	LISTENING
TCP	127.0.0.1:1029	0.0.0.0:0	LISTENING
TCP	127.0.0.1:1048	127.0.0.1:1049	ESTABLISHED

I highlighted the line where the echo server is listening in a passive mode on port 10000. Notice how the outputs are shown as a IP:Port pairs.

If we call EchoClient.exe and then use **netstate**, we will see a **TIME_WAIT** state as we discussed earlier. Also there will be another line showing the echo server in the listening state (shown by the 2 yellow markers).

```
Active Connections

```

Proto	Local Address	Foreign Address	State
TCP	0.0.0.0:135	0.0.0.0:0	LISTENING
TCP	0.0.0.0:445	0.0.0.0:0	LISTENING
TCP	0.0.0.0:523	0.0.0.0:0	LISTENING
TCP	0.0.0.0:2869	0.0.0.0:0	LISTENING
TCP	0.0.0.0:6789	0.0.0.0:0	LISTENING
> TCP	0.0.0.0:10000	0.0.0.0:0	LISTENING
TCP	127.0.0.1:1029	0.0.0.0:0	LISTENING
TCP	127.0.0.1:1048	127.0.0.1:1049	ESTABLISHED
TCP	127.0.0.1:1049	127.0.0.1:1048	ESTABLISHED
TCP	127.0.0.1:1050	127.0.0.1:1051	ESTABLISHED
TCP	127.0.0.1:1051	127.0.0.1:1050	ESTABLISHED
> TCP	127.0.0.1:10000	127.0.0.1:1339	TIME_WAIT

Server Socket Code

```
// Module: EchoServer.cpp
// Author: Rajinder Yadav
// Date:   Sept 5, 2007
//
#include <winsock2.h>
#include <iostream>
#include <process.h>
#include <stdio.h>
#include <tchar.h>
#include <windows.h>

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    // Initialize WinSock2.2 DLL
    // low word = major, highword = minor
    WSADATA wsaData = {0};
    WORD wVer = MAKEWORD(2,2);

    int nRet = WSASStartup( wVer, &wsaData );

    if( nRet == SOCKET_ERROR ) {
        // WSAGetLastError()
        cout << "Failed to init Winsock library" << endl;
        return -1;
    }

    cout << "Starting server" << endl;

    // name a socket
    WORD WSAEvent = 0;
    WORD WSAErr   = 0;

    // open a socket
    //
    // for the server we do not want to specify a network address
    // we should always use INADDR_ANY to allow the protocol stack
    // to assign a local IP address
    SOCKET hSock = {0};
    hSock = socket( AF_INET, SOCK_STREAM, IPPROTO_IP );

    if( hSock == INVALID_SOCKET ) {
        cout << "Invalid socket, failed to create socket" << endl;
        return -1;
    }
}
```

```

// name socket
sockaddr_in saListen = {0};

saListen.sin_family      = PF_INET;
saListen.sin_port        = htons( 10000 );
saListen.sin_addr.s_addr = htonl( INADDR_ANY );

// bind socket's name
nRet = bind( hSock, (sockaddr*)&saListen, sizeof(sockaddr) );

if( nRet == SOCKET_ERROR ) {
    cout << "Failed to bind socket" << endl;
    //shutdown( hSock );
    closesocket( hSock );
    return -1;
}

while( true )
{
    cout << "Listening for connections" << endl;

    // listen
    nRet = listen( hSock, 5 ); // connection backlog queue set to 10

    if( nRet == SOCKET_ERROR )
    {
        int nErr = WSAGetLastError();
        if( nErr == WSAECONNREFUSED ) {
            cout << "Failed to listen, connection refused" << endl;
        }
        else {
            cout << "Call to listen failed" << endl;
        }
        closesocket( hSock );
        return -1;
    }

    // connect
    sockaddr_in saClient = {0};
    int nSALen = sizeof( sockaddr );
    SOCKET hClient = accept( hSock, (sockaddr*)&saClient, &nSALen );

    if( hClient == INVALID_SOCKET ) {
        cout << "Invalid client socket, connection failed" << endl;
        closesocket( hSock );
        return -1;
    }

    cout << "Connection established" << endl;
}

```

```

// process data
char wzRec[512] = {0};
int nLeft = 512;
int iPos = 0;
int nData = 0;
do
{
    nData = recv( hClient, &wzRec[iPos], nLeft, 0 );

    if( nData == SOCKET_ERROR ) {
        cout << "Error receiving data" << endl;
        memset( &wzRec, 0, sizeof( wzRec ) );
        break;
    }
    nLeft -= nData;
    iPos += nData;

} while( nLeft > 0 );

cout << "Data Recieved: " << wzRec << endl;

// echo data back to client
iPos = 0;
nLeft = 512;
do
{
    nData = send( hClient, &wzRec[iPos], nLeft, 0 );

    if( nData == SOCKET_ERROR ) {
        cout << "Error sending data" << endl;
        break;
    }
    nLeft -= nData;
    iPos += nData;

} while( nLeft > 0 );

// close client connection
closesocket( hClient );
hClient = 0;

// perform a lowercase comparison
if( _stricmp( wzRec, "!shutdown" ) == 0 ) {
    break;
}

// clear data buffer
memset( &wzRec, 0, sizeof( wzRec ) );
} // loop

cout << "Shutting down the server" << endl;

```

```
// close server socket
nRet = closesocket( hSock );
hSock = 0;
if( nRet == SOCKET_ERROR ) {
    cout << "Error failed to close socket" << endl;
}

// Release WinSock DLL
nRet = WSACleanup();
if( nRet == SOCKET_ERROR ) {
    cout << "Error cleaning up Winsock Library" << endl;
    return -1;
}

cout << "Server is offline" << endl;
return 0;
}
```

Client Socket Code

```
// Module: EchoClient.cpp
// Author: Rajinder Yadav
// Date: Sept 5, 2007
//
#include <winsock2.h>
#include <iostream>
#include <stdio.h>
#include <tchar.h>
#include <windows.h>

using namespace std;

int main(int argc, char* argv[])
{
    // Initialize WinSock2.2 DLL
    // low word = major, highword = minor
    WSADATA wsaData = {0};
    WORD wVer = MAKEWORD(2,2);

    int nRet = WSASStartup( wVer, &wsaData );

    if( nRet == SOCKET_ERROR ) {
        cout << "Failed to init Winsock library" << endl;
        return -1;
    }

    cout << "Opening connection to server" << endl;

    WORD WSAEvent = 0;
    WORD WSAErr = 0;

    SOCKET hServer = {0};

    // open a socket
    //
    // for the server we do not want to specify a network address
    // we should always use INADDR_ANY to allow the protocol stack
    // to assign a local IP address
    hServer = socket( AF_INET, SOCK_STREAM, IPPROTO_IP );

    if( hServer == INVALID_SOCKET ) {
        cout << "Invalid socket, failed to create socket" << endl;
        return -1;
    }

    // name a socket
    sockaddr_in saServer = {0};

    saServer.sin_family = PF_INET;
    saServer.sin_port = htons( 10000 );
    saServer.sin_addr.s_addr = inet_addr( "127.0.0.1" );
```

```

// connect
nRet = connect( hServer, (sockaddr*)&saServer, sizeof( sockaddr ) );

if( nRet == SOCKET_ERROR ) {
    cout << "Connection to server failed" << endl;
    closesocket( hServer );
    return -1;
}

cout << "Connected to server" << endl;
cout << "Sending data to server" << endl;

// process data
char wzRec[1024] = "Hello from client!!!";
int nLeft = 512;
int iPos = 0;
int nData = 0;

if( argc == 2 ) {
    // copy input string from command argument
    strcpy_s( wzRec, 1024, argv[1] );
}

do
{
    nData = send( hServer, &wzRec[iPos], nLeft, 0 );

    if( nData == SOCKET_ERROR ) {
        cout << "Error sending data" << endl;
        break;
    }
    nLeft -= nData;
    iPos += nData;
} while( nLeft > 0 );

// clear data buffer
memset( &wzRec, 0, sizeof( wzRec ) );

nLeft = 512;
iPos = 0;
do
{
    nData = recv( hServer, &wzRec[iPos], nLeft, 0 );

    if( nData == SOCKET_ERROR ) {
        cout << "Error receiving data" << endl;
        break;
    }
    nLeft -= nData;
    iPos += nData;
} while( nLeft > 0 );

cout << "Data Echoed: " << wzRec << endl;

cout << "Closing connection" << endl;

```

```

// shutdown socket
nRet = shutdown( hServer, SD_BOTH );

if( nRet == SOCKET_ERROR ) {
    // WSAGetLastError()
    cout << "Error trying to perform shutdown on socket" << endl;
    return -1;
}

// close server socket
nRet = closesocket( hServer );
hServer = 0;

if( nRet == SOCKET_ERROR ) {
    cout << "Error failed to close socket" << endl;
}

// Release WinSock DLL
nRet = WSACleanup();

if( nRet == SOCKET_ERROR ) {
    cout << "Error cleaning up Winsock Library" << endl;
    return -1;
}

cout << "Data sent successfully" << endl;
return 0;
}

```