

**FINAL YEAR PROJECT REPORT
ON
DESIGN AND IMPLEMENTATION
OF ECHO SERVER**

*Submitted in Partial fulfillment
for the Requirement of
Degree of Bachelor of Engineering
in Computer Engineering*

By:

ANURAG GOEL	2K1/COE/013
MANHAR P AGGARWAL	2K1/COE/030
SAURABH OHRI	2K1/COE/050

Under the guidance of
Dr D.R. CHOUDHARY
Head of Department
Department of Computer Engineering
Delhi College of Engineering



**DEPARTMENT OF COMPUTER ENGINEERING
DELHI COLLEGE OF ENGINEERING
UNIVERSITY OF DELHI, DELHI
2001-05**

**FINAL YEAR PROJECT REPORT
ON
DESIGN AND IMPLEMENTATION
OF ECHO SERVER**

*Submitted in Partial fulfillment
for the Requirement of
Degree of Bachelor of Engineering
in Computer Engineering*

By:

ANURAG GOEL	2K1/COE/013
MANHAR P AGGARWAL	2K1/COE/030
SAURABH OHRI	2K1/COE/050

Under the guidance of
Dr D.R. CHOUDHARY
Head of Department
Department of Computer Engineering
Delhi College of Engineering



**DEPARTMENT OF COMPUTER ENGINEERING
DELHI COLLEGE OF ENGINEERING
UNIVERSITY OF DELHI, DELHI
2001-05**

ACKNOWLEDGMENT

We are most thankful to our mentor and guide Dr. D.R. Choudhary, Head of Department, Computer Engineering, who introduced us to the new and interesting field of Networking and specifically Client Server Programming. Without his able guidance and many a valuable piece of advice which he gave us whenever we required it, this project would have not been possible. We would also like to thank him for the endless hours he gave us to recommend changes and suggest new ideas to improve our project.

We would be failing in our duties if we don't thank everybody in Delhi College of Engineering who have helped us in some form or other.

ANURAG GOEL

MANHAR P AGGARWAL

SAURABH OHRI

**DEPARTMENT OF COMPUTER
ENGINEERING
DELHI COLLEGE OF ENGINEERING
DELHI**

CERTIFICATE

This is to certify that this project entitled “ **DESIGN AND IMPLEMENTATION OF ECHO SERVER** “ has been submitted by

Anurag Goel (2K1/COE/013)
Manhar P Aggarwal (2K1/COE/030)
Saurabh Ohri (2K1/COE/050)

of Delhi College of Engineering towards the partial fulfillment of the degree of the Bachelor of Engineering in Computer Engineering.

It may be noted that this project was carried out under my supervision and is totally their piece of work and has not been submitted to any other Institute or University.

Dr. D.R. Choudhary
(Project Guide)
Head of Department
Department of Computer Engineering
Delhi College of Engineering
Delhi

TABLE OF CONTENTS

TABLE OF CONTENTS

S.NO	CONTENTS
1.	ABOUT THE PROJECT a)Objective b)Implementation
2.	INTRODUCTION a)TCP/IP b)UDP c)Host Names d)Service Ports e)IP Addresses f)TTL g)Types of Network Prog. h)Client Server i)Intro. to Sockets j)Communication Protocols
3.	SOCKET PROG. IN UNIX C IMPLEMENTATION a)What is Unix b)Socket Prog c)Example d)Functions and Procedures
4.	SOCKET PROG IN WINDOWS JAVA IMPLEMENTATION a)Socket Prog b)Socket Terminology c)Datagrams d)Error Handling e)Explanation of Functions f)Conclusion
5.	V.B. IMPLEMENTATION a)Windows Sockets API b)Prog. With Sockets c)Echo Server d)Echo Client
6.	C CODE a)Server Code b)Client Code c)How to Execute d)Output

7.

JAVA CODE

- a)Server Code
- b)Client Code
- c)How to Execute
- d)Output

8.

VB CODE

- a)Server Code
- b)Client Code
- c)How to Execute
- d)Output

9.

SYSTEM SPECIFICATIONS

10.

RESEARCH APPLICATION

11.

BIBLIOGRAPHY

ABOUT THE PROJECT

ABOUT THE PROJECT

INTRODUCTION

Since birth of network programming, it has been error-prone, difficult, and complex. The programmer had to know many details about the network and sometimes even the hardware. You usually needed to understand the various layers of the networking protocol, and there were a lot to different functions in each networking library concerned with connecting, packing, unpacking blocks of information, handshaking, etc. It was a difficult task. However, the concept of networking is not so difficult. You want to get some information from that machine over there and move it to this machine here, or vice-versa. Its quite similar to reading and writing files, except that the files exist on the remote machine.

OBJECTIVE

The project demonstrates a TCP/IP **echo server**. It will accept a connection from a **client** application, receive one line of text, echo that line back to the **client** and close the connection.

IMPLEMENTATION

The echo server has been implemented in following platforms:

- ❖ **UNIX ENVIRONMENT**
C LANGUAGE
- ❖ **WINDOWS ENVIRONMENT**
JAVA (JDK1.4)
VISUAL BASIC (VB 6.0)

INTRODUCTION

NETWORKING CONCEPTS

TCP/IP

When two computers wish to exchange information over a network, there are several components that must be in place before the data can actually be sent and received. Of course, the physical hardware must exist, which is typically either a network interface card (NIC) or a serial communications port for dial-up networking connections. Beyond this physical connection, however, computers also need to use a protocol which defines the parameters of the communication between them. In short, a protocol defines the "rules of the road" that each computer must follow so that all of the systems in the network can exchange data. One of the most popular protocols in use today is TCP/IP, which stands for Transmission Control Protocol/Internet Protocol.

By convention, TCP/IP is used to refer to a suite of protocols, all based on the Internet Protocol (IP). Unlike a single local network, where every system is directly connected to each other, an internet is a collection of networks, combined into a single, virtual network. The Internet Protocol provides the means by which any system on any network can communicate with another as easily as if they were on the same physical network. Each system, commonly referred to as a host, is assigned a unique 32-bit number which can be used to identify it over the internetwork. Typically, this address is broken into four 8-bit numbers separated by periods. This is called dot-notation, and looks something like "192.43.19.64". Some parts of the address are used to identify the network that the system is connected to, and the remainder identifies the system itself. Without going into the minutia of the Internet addressing scheme, just be aware that there are three "classes" of addresses, referred to as "A", "B" and "C". The rule of thumb is that class "A" addresses are assigned to very large networks, class "B" addresses are assigned to medium sized networks, and class "C" addresses are assigned to smaller networks (networks with less than approximately 250 hosts).

When a system sends data over the network using the Internet Protocol, it is sent in discrete units called datagrams, also commonly referred to as packets. A datagram consists of a header followed by application-defined data. The header contains the addressing information which is used to deliver the datagram to its destination, much like an envelope is used to address and contain postal mail. And like postal mail, there is no guarantee that a datagram will actually arrive at its destination. In fact, datagrams may be lost, duplicated or delivered out of order during their travels over the network.

Needless to say, this kind of unreliability can cause a lot of problems for software developers. What's really needed is a reliable, straightforward way to exchange data without having to worry about lost packets or jumbled data.

To fill this need, the Transmission Control Protocol (TCP) was developed. Built on top of IP, TCP offers a reliable, full-duplex byte stream which may be read and written to in a fashion similar to reading and writing a file. The advantages to this are obvious: the application programmer doesn't need to write code to handle dropped or out-of-order datagrams, and instead can focus on the application itself. And because the data is presented as a stream of bytes, existing code can be easily adopted and modified to use TCP.

TCP is known as a connection-oriented protocol. In other words, before two programs can begin to exchange data they must establish a "connection" with each other. This is done with a three-way handshake in which both sides exchange packets and establish the initial packet sequence numbers (the sequence number is important because, as mentioned above, datagrams can arrive out of order; this number is used to ensure that data is received in the order that it was sent). When establishing a connection, one program must assume the role of the client, and the other the server. The client is responsible for initiating the connection, while the server's responsibility is to wait, listen and respond to incoming connections. Once the connection has been established, both sides may send and receive data until the connection is closed.

The primary function of the TCP/IP is to provide a point to point communication mechanism. One process on one machine communicates with the another process on another machine or within the same machine. This communication appears as two streams of data. One stream carries data from one process to the other, while the other carries data in the other direction. Each process can read the data that have been written by the other, and in normal conditions, the data received are the same, and in the same order, as when they are sent.

In order to tell one machine from another machine and to make sure that you are connected with the machine you want, there must be some way of uniquely identifying machines on a network. Early networks were satisfied to provide unique names for machines within the local network. However, Java works within the Internet, which requires a way to uniquely identify a machine from all the others in the world. This is accomplished with the IP(*Internet Protocol*) address, a 32 bit number.

IP Address in two forms :

The DNS (*Domain Name Service*) form. Suppose, if my domain name is cswl.com and if I have a computer called Hari in my domain. Its domain name would be *Hari.cswl.com*.

Alternatively, we can use dotted quad form, which is four numbers separated by dots, such as 199.2.24.246

In addition to the machine addresses provided by the Internet Protocol part of the network system, TCP/IP has a mechanism for identifying individual processes on a machine, analogous to an office block. The building has phone number , but each room inside is also identified by an extension number. When a call arrives at the building, it must be connected to the correct room for handling . Payment requests go to accounts payable, orders to sales, and so forth. In the TCP/IP system, the extension numbers are called ports, and they are represented by a 16-bit binary number. To communicate with the correct part of a particular computer, the sending machine must know both the machine address and the port number to which the message should be sent. Many common services have a dedicated port. Because some ports are reserved for common services, the programmer cannot use any port. Ports numbered under 1024 are often referred to as reserved ports, many of which are reserved for a specific program. It is important that you only attempt to use ports over number 1024.

USER DATAGRAM PROTOCOL

Unlike TCP, the User Datagram Protocol (UDP) does not present data as a stream of bytes, nor does it require that you establish a connection with another program in order to exchange information. Data is exchanged in discrete units called datagrams, which are similar to IP datagrams. In fact, the only features that UDP offers over raw IP datagrams are port numbers and an optional checksum.

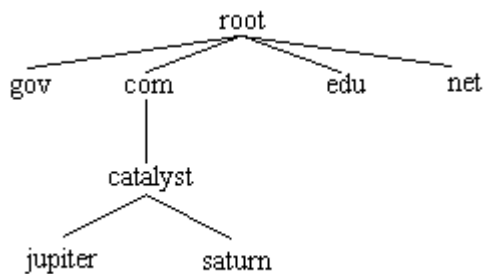
UDP is sometimes referred to as an unreliable protocol because when a program sends a UDP datagram over the network, there is no way for it to know that it actually arrived at its destination. This means that the sender and receiver must typically implement their own application protocol on top of UDP. Much of the work that TCP does transparently (such as generating checksums, acknowledging the receipt of packets, retransmitting lost packets and so on) must be performed by the application itself.

With the limitations of UDP, you might wonder why it's used at all. UDP has the advantage over TCP in two critical areas: speed and packet overhead. Because TCP is a reliable protocol, it goes through great lengths to insure that data arrives at its destination intact, and as a result it exchanges a fairly high number of packets over the network. UDP doesn't have this overhead, and is considerably faster than TCP. In those situations where speed is paramount, or the number of packets sent over the network must be kept to a minimum, UDP is the solution.

HOSTNAMES

In order for an application to send and receive data with a remote process, it must have several pieces of information. The first is the IP address of the system that the remote program is running on.

Although this address is internally represented by a 32-bit number, it is typically expressed in either dot-notation or by a logical name called a hostname. Like an address in dot-notation, hostnames are divided into several pieces separated by periods, called domains. Domains are hierarchical, with the top-level domains defining the type of organization that network belongs to, with sub-domains further identifying the specific network.



In this figure, the top-level domains are "gov" (government agencies), "com" (commercial organizations), "edu" (educational institutions) and "net" (Internet service providers). The fully qualified domain name is specified by naming the host and each parent sub-domain above it, separating them with periods. For example, the fully qualified domain name for the "jupiter" host would be "jupiter.catalyst.com". In other words, the system "jupiter" is part of the "catalyst" domain (a company's local network) which in turn is part of the "com" domain (a domain used by all commercial enterprises).

In order to use a hostname instead of a dot-address to identify a specific system or network, there must be some correlation between the two. This is accomplished by one of two means: a local host table or a name server. A host table is a text file that lists the IP address of a host, followed by the names that it's known by. Typically this file is named `hosts` and is found in the same directory in which the TCP/IP software has been installed. A name server, on the other hand, is a system (actually, a program running on a system) which can be presented with a hostname and will return that host's IP address. This approach is advantageous because the host information for the entire network is maintained in one centralized location, rather than being scattered about on every host on the network.

SERVICE PORTS

In addition to the IP address of the remote system, an application also needs to know how to address the specific program that it wishes to communicate with. This is accomplished by specifying a service port, a 16-bit number that uniquely identifies an application running on the system. Instead of numbers, however, service names are usually used instead. Like hostnames, service names are usually matched to port numbers through a local file, commonly called services. This file lists the logical service name, followed by the port number and protocol used by the server.

A number of standard service names are used by Internet-based applications and these are referred to as well-known services. These services are defined by a standards document and include common application protocols such as FTP, POP3, SMTP and HTTP.

DIFFERENT TYPES OF IP ADDRESSES

There are three types of IPv4 addresses: unicast, broadcast, and multicast

Unicast addresses are used for transmitting a message to a single destination node

Broadcast addresses are used when a message is supposed to be transmitted to all subnetwork

For delivering a message to a group of destination nodes which are not necessarily subnetwork, **multicast** addresses are used

Class A, B, and C IP addresses are used for unicast messages, whereas as class D those in the range 224.0.0.1 to 239.255.255.255, inclusive, and by a standard UDP 1 are used for multicast messages

IP Addresses of Class D - Multicasting

IP addresses of Class D have the following format

Bit no. 0 1 2 3 4 5 6 7 8 16 24 31

Class D 1 1 1 0 |-----multicast address (28)-----|

Class D addresses are identified by a one in bit 0,1 and 2 and a zero in bit 3 of the a means that 6.25% of all available IP addresses are of this class

The range of Class D addresses are in dotted decimal notation from 224.h.h.h.h t where h is a number from 0 to 255. Address 224.0.0.0 is reserved and can not be address 224.0.0.1 is used to address all hosts that take part in IP multicasting

Class D addresses are used for multicasting and does not have a network part and h multicasting makes it possible to send IP datagrams to a group of hosts, which ma across many networks

LIFE OF MULTICAST PACKETS (TTL)

Broadcast packets need to have a finite life in order to avoid bouncing of the packets around the network forever. Each packet has a time to live (TTL) value, a counter that is decremented every time the packet passes through an hop i.e a router between the network. Because of TTLs, each multicast packet is a ticking time bomb.

Take for example, a TV station where TTLs would be the station's signal area -- the limitation of how far the information can travel. As the packet moved around the company's internal network, its TTL would be notched down every time it passed through a router. When the packet's TTL reached 0, the packet would die and not be passed further. Generally multicast with long TTLs -- perhaps 200 - to guarantee that the information will reach around the world

TYPES OF NETWORK PROGRAMMING

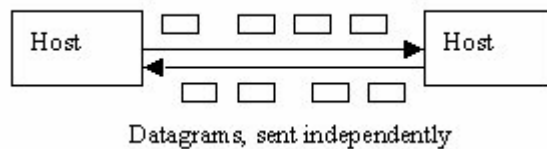
Two general types are :

- Connection-oriented programming
- Connectionless Programming

Connection-oriented Networking

The client and server have a communication link that is open and active from the time the application is executed until it is closed. Using Internet jargon, the Transmission control protocol is a connection oriented protocol. It is reliable connection - packets are guaranteed to arrive in the order they are sent.

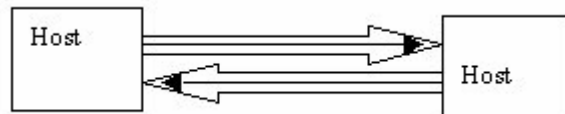
(e.g.) Telephone system.



Connection-less Networking

In this type each instance that packets are sent, they are transmitted individually. No link to the receiver is maintained after the packets arrive. The Internet equivalent is the User Datagram Protocol (UDP). Connectionless communication is faster but not reliable. Datagrams are used to implement a connectionless protocol, such as UDP.

(e.g.) Postal Service



Common Services Port Number

<i>Port Number</i>	<i>Service</i>
21	FTP
23	Telnet
25	SMTP(mail)
80	HTTP(Web)
119	NNTP(News)

CLIENT SERVER PROGRAMMING

The most common model of network programming is referred to as client-server programming. The concept is simple: A client machine makes a request for information or sends a command to a server; in return, the server passes back the data or results of the command. Most often, the server only responds to clients; it does not initiate communication.

So the job of the server is to listen for a connection, and that's performed by the special server object that we create. The job of the client is to try to make a connection to the server, and this is performed by the special client object we create. Once the connection is made, you will see that at the server and client ends, the connection is magically just turned into the IO Stream object, and from then onwards you can treat the connection as if you were reading and writing into the file.

INTRODUCTION TO SOCKETS

WHAT ARE SOCKETS

The *socket* is a software abstraction used to represent the "terminals" of a connection between two machines or processes. For a given connection, there's a socket on each machine, and you can imagine a hypothetical "cable" running between the two machines with each end of the "cable" plugged into the socket. Of course, the physical hardware and cabling between machines is completely unknown. A socket is one end-point of a two-way communication link between two programs running on the network. These two programs form a Client/Server application.

Definition: A *socket* is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent.

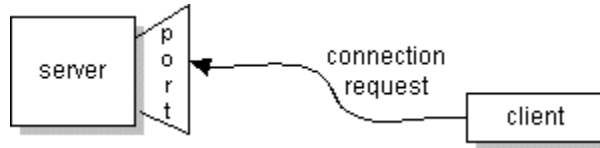
SOCKETS IN CLIENT SERVER APPLICATIONS

In Client/Server applications the server normally listens to a specific port waiting for connection requests from a client. When a connection request arrives, the client and the server establish a dedicated connection over which they can communicate. During the connection process, the client is assigned a local port number, and binds a socket to it. The client talks to the server by writing to the socket and gets information from the server by reading from it. Similarly, the server gets a new local port number to communicate with the client. The server also binds a socket to its local port and communicates with the client by reading from and writing to it. The server can not use its specific port to communicate with the client since it is dedicated only to listen for connection requests from other clients.

The client and the server must agree on a protocol, that is, they must agree on the language of the information transferred back and forth through the socket.

Normally, a server runs on a specific computer and has a socket that is bound to a specific port number. The server just waits, listening to the socket for a client to make a connection request.

On the client-side: The client knows the hostname of the machine on which the server is running and the port number to which the server is connected. To make a connection request, the client tries to rendezvous with the server on the server's machine and port.



If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bound to a different port. It needs a new socket (and consequently a different port number) so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client.



On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server. Note that the socket on the client side is not bound to the port number used to rendezvous with the server. Rather, the client is assigned a port number local to the machine on which the client is running.

The client and server can now communicate by writing to or reading from their sockets

TYPES OF SOCKETS

There are three types of sockets:

- **Stream socket (To listen)**
- **Stream socket**
- **Datagram socket**

COMMUNICATION PROTOCOLS

- **stream communication**
- **datagram communication.**

The stream communication protocol is known as **TCP** (transfer control protocol). Unlike UDP, TCP is a connection-oriented protocol. In order to do communication over the TCP protocol, a connection must first be established between the pair of sockets. While one of the sockets listens for a connection request (server), the other asks for a connection (client). Once two sockets have been connected, they can be used to transmit data in both (or either one of the) directions. TCP provides a reliable, point-to-point communication channel that client-server applications on the Internet use to communicate with each other. To communicate over TCP, a client program and a server program establish a connection to one another. Each program binds a socket to its end of the connection. To communicate, the client and the server each reads from and writes to the socket bound to the connection

The datagram communication protocol, known as **UDP** (user datagram protocol), is a connectionless protocol, meaning that each time you send datagrams, you also need to send the local socket descriptor and the receiving socket's address. As you can tell, additional data must be sent each time a communication is made.

UDP is an unreliable protocol, there is no guarantee that datagrams you have sent will be received in the same order by the receiving socket. On the other hand, TCP is a reliable protocol; it is guaranteed that the packets you send will be received in the order in which they were sent.

TCP is useful for implementing network services, such as remote login (rlogin, telnet) and file transfer (FTP) -- which require data of indefinite length to be transferred. UDP is less complex and is often used in implementing client/server applications in distributed systems built over local area networks.

SOCKET PROGRAMMING

IN

UNIX

C IMPLEMENTATION

WHAT IS UNIX ?

Unix is an operating system which like all other operating systems acts as a *master of ceremonies* since its job is to accept and dispatch user commands and directing the systems response to the appropriate place. Unix is referred to as a multi-user, multitasking operating system, since multiple users may each execute multiple commands seemingly simultaneously. Multitasking is achieved by running tasks in the background and when using X it can also mean running multiple windows each with foreground and/or background activity. It involves sharing the processor(s) among multiple different programs, and creating a place in memory for each of those programs (processes).

Preference for any operating system is mainly a product of its ease-of-use, flexibility, reliability, and powerfulness. Unix is flexible, reliable and powerful which can lead to ease-of-use if required. The **shell** is the particular program which

- 1.prompts for input
- 2.translates the any special characters in the command line
- 3.either executes the inputted command line or passes the request.

Apart from application programs we spend most of our time interacting with the shell, and hence is extensively covered in this document. There are two shells that are common across almost all implementations of Unix: the Bourne (**sh**) and the C-shell (**csh**). The C-shell has superior interactive features and the Bourne shell has extensive programmable features and runs more quickly. Generally the C-shell is more popular, although both shells have much in common. Take care if you are buying Unix books that they describe the right shell. In either case the shell insulates the user from the Unix kernel which is the software which dispatches the services. The kernel creates the illusion that all systems look like the same virtual machine by providing a consistent set of services irrespective of hardware details. Strictly speaking the kernel *is* Unix which is why standards like POSIX define exactly what services are provided by the kernel. It should be noted that the shell is no more privileged a program than any other and may be easily be replaced for users with particular needs, which can be useful for providing restricted services when needed.

SOCKET PROGRAMMING

TYPICAL CLIENT SERVER COMMUNICATION

Basic steps:

The basic steps in a typical sockets session can be summarized as follows:

- **Server:** Create socket, establish network addressability, wait for connection request
- **Client:** Create socket, send connection request to server
- **Server and Client:** Establish connection
- **Server and Client:** Transmit and receive data

Server and Client: Close connection.

A SIMPLE EXAMPLE

The following table describes a simple connection-oriented client-server sockets application.

CLIENT ACTION	CLIENT SYSTEM CALL	SERVER ACTION	SERVER SYSTEM CALL	DESCRIPTION
Create socket descriptor	<i>socket()</i>	Create socket descriptor	<i>socket()</i>	The <i>socket()</i> call creates a socket descriptor, which is similar to a file descriptor. The protocol family must be chosen when <i>socket()</i> is issued.
n/a	n/a	Associate network address with socket	Bind()	The server socket must be network-addressable.
n/a	n/a	Wait for incoming message	<i>listen()</i> and <i>accept()</i>	<i>Listen()</i> notifies the operating system that the server process is ready to receive messages. <i>Accept()</i> suspends the server process until a message arrives.
Contact server	<i>connect()</i>	n/a	n/a	<i>Connect()</i> establishes a network connection with the server process.
Transmit and receive data	<i>write() read()</i>	Transmit and receive data	<i>write() read()</i>	This is similar to file I/O. The socket connection is duplex.
Terminate connection	<i>Close()</i>	Terminate connection	<i>close()</i>	This is analogous to closing a file.

FUNCTIONS & PROCEDURES INVOLVED

SOCKET CREATION

socket() FUNCTION:

The server socket is created with the **socket()** call, which takes the following arguments:

AF_INET

The first argument, socket *domain*, selects the family of communication protocols that will be used to control the data flowing through the socket. AF_INET is a symbolic constant representing the Internet family of protocols. If the value of this argument is AF_UNIX, the socket will operate in the "Unix domain." This means it will communicate with other processes on the same Unix system only, and will not support communication across the network.

SOCK_STREAM

The symbolic constant SOCK_STREAM provides a value for socket *type*, which indicates whether communication through the socket will be connection-oriented or connectionless. SOCK_STREAM signifies that the communication will be connection-oriented, whereas SOCK_DGRAM signifies that communication will consist of the connectionless transmission of data packets called datagrams.

0

The *protocol* argument allows the programmer to specify a specific protocol within the protocol family.

For example, the symbolic constant IPPROTO_TCP specifies the Transmission Control Protocol (TCP). Typically this argument is set to zero, allowing the system to select a protocol.

BIND() FUNCTION

SYNTAX:

```
#include<sys/socket.h>
```

```
int bind(int sockfd , const struct sockaddr*myaddr , socklen_t_addrlen);
```

FUNCTION:

The `bind()` system call binds a socket to an address. It takes three arguments, the socket file descriptor, the address to which is bound, and the size of the address to which it is bound.

The second argument is a pointer to a structure of type `sockaddr`, but what is passed in is a structure of type `sockaddr_in`, and so this must be cast to the correct type. This can fail for a number of reasons, the most obvious being that this socket is already in use on this machine

LISTEN() FUNCTION

SYNTAX:

```
#include<sys/socket.h>
```

```
int listen(int sockfd , int backlog);
```

FUNCTION:

The `listen` system call allows the process to listen on the socket for connections. The first argument is the socket file descriptor, and the second is the size of the backlog queue, i.e., the number of connections that can be waiting while the process is handling a particular connection. This should be set to 5, the maximum size permitted by most systems. If the first argument is a valid socket, this call cannot fail, and so the code doesn't check for errors.

ACCEPT() FUNCTION

SYNTAX:

```
#include<sys/socket.h>
```

```
int accept(int sockfd , struct sockaddr *cliaddr , socklen_t *addrlen);
```

FUNCTION:

The `accept()` system call causes the process to block until a client connects to the server. Thus, it wakes up the process when a connection from a client has been successfully established. It returns a new file descriptor, and all communication on this connection should be done using the new file descriptor. The second argument is a reference pointer to the address of the client on the other end of the connection, and the third argument is the size of this structure.

CONNECT() FUNCTION

SYNTAX:

```
#include<sys/socket.h>
```

```
int connect(int sockfd , const struct sockaddr *servaddr , socklen_t addrlen);
```

FUNCTION:

The `connect` function is called by the client to establish a connection to the server. It takes three arguments, the socket file descriptor, the address of the host to which it wants to connect (including the port number), and the size of this address. This function returns 0 on success and -1 if it fails. Notice that the client needs to know the port number of the server, but it does not need to know its own port number. This is typically assigned by the system when `connect` is called.

CLOSE() FUNCTION

SYNTAX:

```
#include<unistd.h>
```

```
int close(int sockfd);
```

FUNCTION:

The default action of `close` with a TCP socket is to mark the socket as closed and return to the process immediately.

SOCKET PROGRAMMING
IN
WINDOWS

JAVA IMPLEMENTATION

SOCKET PROGRAMMING

The `java.net` package in the Java development environment provides a class `Socket` which implements the client side and the class `ServerSocket` class which implements the server side of the two-way link. The `Socket` class sits on top of a platform-dependent implementation, hiding the details of any particular system from your Java program. By using the `java.net.Socket` class instead of relying on native code, your Java programs can communicate over the network in a platform-independent fashion.

Additionally, `java.net` includes the `ServerSocket` class, which implements a socket that servers can use to listen for and accept connections to clients. This lesson shows you how to use the `Socket` and `ServerSocket` classes.

If you are trying to connect to the Web, the `URL` class and related classes (`URLConnection`, `URLEncoder`) are probably more appropriate than the socket classes. In fact, URLs are a relatively high-level connection to the Web and use sockets as part of the underlying implementation

There are three types of sockets:

- Stream socket (To listen): class `ServerSocket`.
- Stream socket: class `Socket`.
- Datagram socket: class `DatagramSocket`.

SOCKET TERMINOLOGY

In Java, we need to create a socket to make the connection to the other machine. Then you can get an **InputStream** and **OutputStream** from the socket in order to be able to treat the connection as an IOStream object. There are two stream based socket classes in the java.net package. They are java.net.**ServerSocket** that a server uses to listen for incoming connections and a java.net.Socket that a client uses in order to initiate a connection. Once a client makes a Socket connection, the ServerSocket returns a corresponding server side socket through which direct communications will take place.

When we create a ServerSocket, you give it only a port number. You don't have to give it an IP address because it's already on the machine it represents. When you create a Socket, however, you must give both the IP address and the port number where you're trying to connect.

Socket Classes

Socket
ServerSocket
DatagramSocket
MulticastSocket

Before discussing the constructors and methods of Socket and ServerSocket, the class InetAddress must be mentioned. An InetAddress represents the actual number, not the name or IP address of a computer. The name Hari.cswl.com is never used by your program; instead it uses the corresponding address, 199.2.24.246. The InetAddress class has no constructors, instead it has some methods that returns the InetAddress address.

Socket

Socket object is the Java representation of a TCP connection. when a socket is created, a connection is opened to the specified destination.

Constructors:

The Socket provides the programmer with four constructors. The address of the server may be specified as a string or an InetAddress, and the port number on the host to connect to. In each case, an optional Boolean parameter implements a connectionless socket if set to false.

Methods:

The two most important methods are `getInputStream()` and `getOutputStream()`, which return stream objects that can be used to communicate through the socket. A `close()` method is provided to tell the underlying operating system to terminate the connection. Methods are also provided to retrieve information about the connection to the local host and remote port numbers and an integer representing the remote host. Another method is `accept()`. It returns a `Socket` that is connected to the client. The `close()` method tells the operating system to stop listening for requests on the socket. Methods to retrieve the host name, the socket is listening on and the port number being listened to are also provided.

ServerSocket

The `ServerSocket` represents a listening TCP connection. Once an incoming connection is requested, the `ServerSocket` object will return a `Socket` object representing the connection.

DATAGRAMS

Datagrams are used to implement a connectionless protocol, such as UDP. Two classes are used to implement datagrams in Java:

1. java.net.DatagramPacket
2. java.net.DatagramSocket

DatagramPacket is the actual packet of information, an array of bytes, that is transmitted over the network. DatagramSocket is a socket that sends and receives DatagramPackets across the network. You can think of the DatagramPacket as a letter and a DatagramSocket as the mailbox that the mailcarrier uses to pick up and drop off your letters.

DatagramPacket

The DatagramPacket class provides the programmer with two constructors. The first is used for DatagramPackets that receive information. This constructor needs to be provided with an array to store the data and the amount of data to receive. The second is used to create DatagramPackets that send data. The constructor requires the same information, plus the destination address and the port number.

Methods:

There are four methods in this class - allowing the data, datagram length, and addressing (InetAddress) and port number information for the packet to be extracted.

DatagramSocket

The DatagramSocket represents a connectionless datagram socket. This class works with the DatagramPacket class to provide for communication using the UDP protocol. It provides two constructors, the programmer can specify a port to use or allow the system to randomly use one.

Methods:

The two most important methods are - send() and receive(). Each takes as an argument an appropriately constructed DatagramPacket. In the case of the send() method, the data contained in the packet is sent to the specified host and the port. The receive() method will block the execution until a packet is received by the underlying socket, at which time the data will be copied into the packet provided.

ERROR HANDLING

Error handling is done by the class called **SocketException** which extends IOException class. This exception is thrown when there is a problem using socket. (i.e.) error in the underlying protocol, such as a TCP error.

One possible cause is that the local port you are using for is already in use. Another cause is that the user cannot bind to that particular port. Because, on most operating systems, port numbers less than 1,024 cannot be used by the programmer except the super user. This is the security measure, because most well known services reside on the ports in this range.

Some New SocketExceptions

All the new exceptions extends the SocketException class: They are as follows:

BindException:

The local port is in use, or the requested bind address couldn't be assigned locally.

ConnectException:

This exception is raised when a connection is refused at the remote host (i.e., no process is listening on that port).

NoRouteToHostException:

The connect attempt timed out, or the remote host is otherwise unreachable.

EXPLANATION OF DIFFERENT FUNCTIONS

OPENING A SOCKET

If you are programming a client, then you would open a socket like this:

```
Socket MyClient;  
MyClient = new Socket("Machine name", PortNumber);  
or using the exception handling,  
Socket MyClient;  
try {  
    MyClient = new Socket("Machine name", PortNumber);  
}  
catch (IOException e) {  
    System.out.println(e);  
}
```

Where Machine name is the machine you are trying to open a connection to, and PortNumber is the port (a number) on which the server you are trying to connect to is running. When selecting a port number, you should note that port numbers between 0 and 1023 are reserved are reserved for standard services, such as email, FTP, and HTTP.

If you are programming a server, then this is how you open a socket:

```
ServerSocket MyService;  
try {  
    MyService = new ServerSocket(PortNumber);  
}  
    catch (IOException e) {  
        System.out.println(e);  
    }
```

When implementing a server you also need to create a socket object from the ServerSocket in order to listen for and accept connections from clients.

```
Socket clientSocket = null;  
try {  
    serviceSocket = MyService.accept();  
}  
catch (IOException e) {  
    System.out.println(e);  
}
```


CREATING AN INPUT STREAM

On the client side, you can use the `DataInputStream` class to create an input stream to receive response from the server:

```
DataInputStream input;  
try {  
    input = new DataInputStream(MyClient.getInputStream());  
}  
catch (IOException e) {  
    System.out.println(e);  
}
```

The class `DataInputStream` allows you to read lines of text and Java primitive data types in a portable way. It has methods such as `read`, `readChar`, `readInt`, `readDouble`, and `readLine`. Use whichever function you think suits your needs depending on the type of data that you receive from the server.

On the server side, you can use `DataInputStream` to receive input from the client:

```
DataInputStream input;  
try {  
    input = new DataInputStream(serviceSocket.getInputStream());  
}  
catch (IOException e) {  
    System.out.println(e);  
}
```

CREATE AN OUTPUT STREAM

On the client side, you can create an output stream to send information to the server socket using the class `PrintStream` or `DataOutputStream` of `java.io`:

```
PrintStream output;  
try {  
    output = new PrintStream(MyClient.getOutputStream());  
}  
catch (IOException e) {  
    System.out.println(e);  
}
```

The class `PrintStream` has methods for displaying textual representation of Java primitive data types. Its `write` and `println` methods are important here. Also, you may want to use the `DataOutputStream`:

```
DataOutputStream output;  
try {  
    output = new DataOutputStream(MyClient.getOutputStream());  
}  
catch (IOException e) {  
    System.out.println(e);  
}
```

The class `DataOutputStream` allows you to write Java primitive data types; many of its methods write a single Java primitive type to the output stream. The method `writeBytes` is a useful one.

On the server side, you can use the class `PrintStream` to send information to the client.

```
PrintStream output;  
try {  
    output = new PrintStream(serviceSocket.getOutputStream());  
}  
catch (IOException e) {  
    System.out.println(e);  
}
```

CLOSING SOCKETS

You should always close the output and input stream before you close the socket. On the client side:

```
try {
    output.close();
    input.close();
    MyClient.close();
}
catch (IOException e) {
    System.out.println(e);
}
```

On the server side:

```
try {
    output.close();
    input.close();
    serviceSocket.close();
    MyService.close();
}
catch (IOException e) {
    System.out.println(e);
}
```

CONCLUSION

Sockets allows to implement Client/Socket applications and provide a powerful and flexible infrastructure for network programming. There are some other classes for network programming. If you are trying to connect to the World Wide Web, the URL class and related classes (URLConnection, URLEncoder) are probably more suitable than the socket classes to what you are doing. In fact, URLs are a relatively high level connection to the Web and use sockets as part of the underlying implementation.

VISUAL BASIC IMPLEMENTATION

WINDOWS SOCKETS API

The Windows Sockets specification was created by a group of companies, including Microsoft, in an effort to standardize the TCP/IP suite of protocols under Windows. Prior to Windows Sockets, each vendor developed their own proprietary libraries, and although they all had similar functionality, the differences were significant enough to cause problems for the software developers that used them. The biggest limitation was that, upon choosing to develop against a specific vendor's library, the developer was "locked" into that particular implementation. A program written against one vendor's product would not work with another's. Windows Sockets was offered as a solution, leaving developers and their end-users free to choose any vendor's implementation with the assurance that the product will continue to work.

There are two general approaches that you can take when creating a program that uses Windows Sockets. One is to code directly against the API. The other is to use a component which provides a higher-level interface to the library by setting properties and responding to events. This can provide a more "natural" programming interface, and it allows you to avoid much of the error-prone drudgery commonly associated with sockets programming. By including the control in a project, setting some properties and responding to events, you can quickly and easily write an Internet-enabled application. And because of the nature of custom controls in general, the learning curve is low and experimentation is easy.

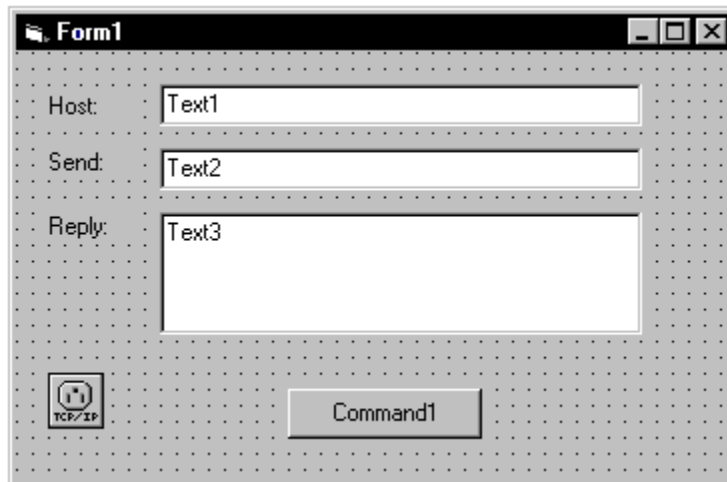
PROGRAMMING WITH SOCKETWRENCH IN VISUAL BASIC

A SAMPLE CLIENT PROGRAM

The program will be used to connect with an echo server, a program which echoes back any data that's sent to it.

The first step, after starting Visual Basic, is to include the SocketWrench control. In Visual Basic 4.0, you should select **Tools|Custom Controls**, while in Visual Basic 5.0 and Visual Basic 6.0, you should select **Project|Components**. A dialog will display all of the available ActiveX controls, then select the Catalyst SocketWrench Control.

To begin, create a form that has three labels, three text controls, a button and the SocketWrench control. The form look something like this:



When executed, the user will enter the name or IP address of the system in the Text1 control, the text that is to be echoed in the Text2 control, and the server's reply will be displayed in the Text3 control. The Command1 button will be used to establish a connection with the remote server. When you save your project, call it "Client".

First we should initialize the controls in the Form's Load subroutine. Note that we want to disable the Text2 and Text3 controls, since they only should be usable once a connection to a server has been established. The code should look like this:

```
Private Sub Form_Load()  
    Command1.Caption = "Connect"  
    Command1.Enabled = True  
    Text1.Enabled = True
```

```
Text2.Enabled = False
Text3.Enabled = False
End Sub
```

The next step is to write the code that actually establishes a connection with the remote server in the **Click** event for the Command1 button. The code should look like this:

```
Private Sub Command1_Click()

    If Not SocketWrench1.Connected Then
        Dim strRemoteHost As String
        Dim nError As Long

        strRemoteHost = Trim(Text1.Text)

        SocketWrench1.AutoResolve = False
        SocketWrench1.Blocking = False
        SocketWrench1.Protocol = swProtocolTcp

        nError = SocketWrench1.Connect(strRemoteHost, swPortEcho)

        If nError <> 0 Then
            MsgBox "Unable to connect to remote host", vbExclamation
            Exit Sub
        End If

        Command1.Enabled = False
    Else
        SocketWrench1.Disconnect
        Command1.Caption = "Connect"
        Text2.Enabled = False
        Text3.Enabled = False
    End If

End Sub
```

The first SocketWrench property that we encounter is the **Connected** property. This is a boolean flag which tells us if the control has established a connection to a remote host. We're using this to allow the Command1 button to function in one of two ways: if no connection has been established, then pressing the button will cause the client to make a connection to the server entered in the Text1 control. However, if there is an active connection, then pressing the button will disconnect the client from the server.

These next three SocketWrench properties are used to define some basic functions of the control, such as how host names are resolved and what network protocol is used. These properties are:

- AutoResolve** This property specifies that the control should not immediately attempt to resolve host names into IP addresses if the **HostName** and/or **HostAddress** property are set. In general it is recommended that you initialize this property value to False unless your application has a specific to automatically resolve host names.
- Blocking** This property specifies if the application should wait for a socket operation to complete before continuing. By setting this property to False, that indicates that the application will not wait for the operation to complete, and instead will respond to events generated by the control. This is the recommended approach to take when designing your application.
- Protocol** This property determines which protocol is going to be used to communicate with the remote application. Most commonly, the value **swProtocolTcp** is specified, which means that the stream-based Transmission Control Protocol will be used. To send UDP datagrams, this property can be set to the value **swProtocolUdp**.

To establish the connection to the server, the **Connect** method is called, passing the name of the server to connect to and the port number of the echo server. It should be noted that there are a number of optional arguments to this method, but for the purposes of this example, only the host name and port number are needed. If the connection attempt is successful, the method will return a value of zero. However, if an error occurs the method will return a non-zero value which specifies an error code.

If the connection attempt is successful, then the Command1 button is disabled. Because the socket is non-blocking (that is, the **Blocking** property is False), when the **Connect** method returns it does not mean that the connection has actually completed. Instead, it means that the connection process has begun, and completion is signaled by the control's **OnConnect** event firing. So between the time that the **Connect** method is called to establish a connection and the time that the **OnConnect** event is fired to indicate that the connection has been completed, the user should not be able to press the Command1 button because it would result in the **Connect** method being called again.

To update our form when a connection has been established, we need to add some code to the control's **OnConnect** event. Remember, this event is only called after a connection attempt has completed on a non-blocking socket:

```
Private Sub SocketWrench1_Connect()  
    Command1.Caption = "Disconnect"  
    Command1.Enabled = True  
    Text2.Enabled = True  
    Text3.Enabled = True
```

```
MsgBox "Connect to remote host", vbInformation
End Sub
```

This will change the caption of our Command1 button to "Disconnect" (informing the user that when they press it, now it will disconnect the current session), and enable our Text2 and Text3 controls. We also display a message box indicating that the connection has completed.

There is a possibility that the remote host may terminate our connection, and our client application needs to be able to handle this. If this happens, for example if the server is stopped, then the control's **OnDisconnect** event will fire. In our code, we'll reset our command button's caption, disable the Text2 and Text3 controls and display a message box indicating that the connection has been lost. The code would look like this:

```
Private Sub SocketWrench1_Disconnect()
    SocketWrench1.Disconnect
    Command1.Caption = "Connect"
    Command1.Enabled = True
    Text2.Enabled = False
    Text3.Enabled = False
    MsgBox "Disconnected from remote host", vbInformation
End Sub
```

When the **OnDisconnect** event fires, what the control is telling you is that the other socket, in this case the server's socket, has been closed. However, until you call the **Disconnect** method it will remain open on the client side. For the connection to be completely terminated, the sockets on both ends of the connection need to be closed.

What happens if there is an error while the client attempts to connect to the server? It is possible for the **Connect** method to return zero (indicating success), and then once the connection attempt begins, an error occurs. For example, this can happen if there is no server listening on the specified port number. To be able to handle this, the control has an event called **OnError** which is fired whenever an error such as this occurs. Let's add some code to the event to report any errors:

```
Private Sub SocketWrench1_Error(ByVal Error As Variant, _
    ByVal Description As Variant)

    If Error <> swErrorOperationWouldBlock Then
        SocketWrench1.Disconnect
        Command1.Caption = "Connect"
        Command1.Enabled = True
        Text2.Enabled = True
        Text3.Enabled = True
        MsgBox Description, vbExclamation, "Error " & CStr(Error)
    End If
End Sub
```

The **OnError** event has two arguments passed to it, an error code and a textual description of the error. The first thing that we do is compare this error against one of our predefined error constants `swErrorOperationWouldBlock` which occurs if a socket operation would cause a non-blocking socket to block. For example, attempting to read data from a non-blocking

socket and there is no data available at that time would result in this error. The reason that we're specifically checking for it is because this particular error code is really more of a warning to the application, not a fatal error. In all other cases, we disconnect the client session and report the error.

Now that the code to establish the connection has been written, the next step is to actually send and receive data to and from the server. To do this, the Text2 control should have the following code added to its **KeyPress** event:

```
Private Sub Text2_KeyPress(KeyAscii As Integer)
    If KeyAscii = 13 Then
        Dim strBuffer As String
        Dim cchBuffer As Long, nResult As Long

        strBuffer = Text2.Text & vbCrLf
        cchBuffer = Len(strBuffer)
        Text2.Text = ""
        KeyAscii = 0

        nResult = SocketWrench1.Write(strBuffer, cchBuffer)
        If nResult = -1 Then
            MsgBox "Unable to send data to server"
            Exit Sub
        End If
    End If
End Sub
```

The **Write** method is used to send data to the remote server. The first argument is the buffer that contains the data (in this case, a string variable) and the second argument is the number of bytes to write. Note that the second argument is optional and if it is omitted the entire buffer is written. For clarity, it is recommended that the buffer length be specified. Note that in addition to strings, the **Write** method will also accept bytes and byte arrays as parameters.

Because our example is connecting to an echo service, once the data has been sent to the remote host, it immediately sends the data back to the client. This generates an **OnRead** event in SocketWrench, which should have the following code:

```
Private Sub SocketWrench1_Read()
    Dim strBuffer As String
    Dim nResult As Long

    nResult = SocketWrench1.Read(strBuffer, 1024)
    If nResult > 0 Then
        Text3.Text = Text3.Text + strBuffer
    End If
End Sub
```

The **OnRead** event indicates that data has arrived and is available to be read by the control. The **Read** method then reads the data sent by the server and stores it in the buffer specified in the first parameter. The second parameter specifies the maximum number of bytes to read. Note that in this case, it is an arbitrary value of 1,024 bytes. One important thing to note is that requesting to read a specified number of bytes does not guarantee that you will actually receive that amount. Because TCP is a stream-oriented protocol, there is no concept of a

"message boundary" or a one-to-one relationship between the amount of data written to the socket and the amount of data read from it. In other words, the server sends four pieces of data in 512 byte blocks, there is no guarantee that your program will get four **OnRead** events for that number of bytes per read. Instead, you may get more than four events (in which the data sent is received in smaller blocks) or you may get fewer events, with the data being combined. This is the nature of how TCP/IP works, and must be accounted for in the design of you application. Typically this means buffering the data in the program and either looking for special "end of message" characters in the data stream, accumulating data in fixed sizes and processing it as the buffer is filled.

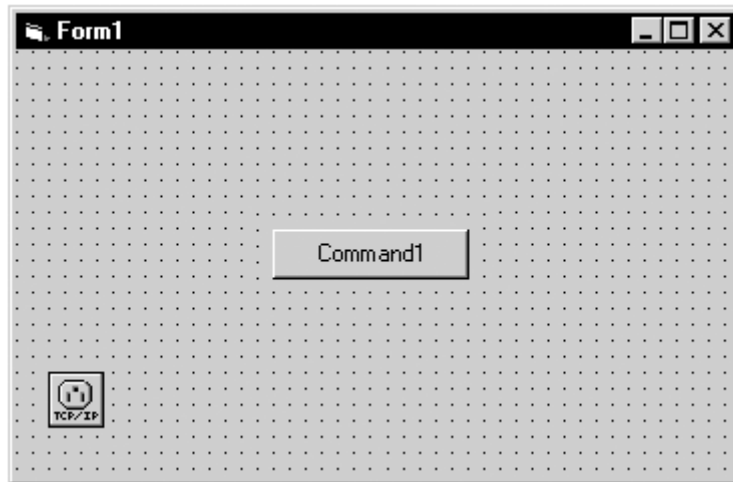
The last piece of code to add to the sample is to handle closing the socket when the program is terminated by selecting Close on the system menu. The best place to put socket cleanup code is in the form's **Unload** event, such as:

```
Sub Form_Unload (Cancel As Integer)
    If SocketWrench1.Connected Then SocketWrench1.Disconnect
End
End Sub
```

If the **Connected** property returns True, then a connection has been established and we should disconnect from the server before the program terminates. With all of the properties and event code needed for the sample client application completed, all that's left to do is run the program! Of course, in a real application you'd need to provide extensive error checking. SocketWrench errors start at 10,000 and correspond to the error codes used by the Windows Sockets API. Most errors will occur when setting the host name, address, service port or using one of the methods.

BUILDING AN ECHO SERVER

The next step is to implement your own echo server. The server will listen on the echo port, accept connections from one or more clients and echo back any data that is sent to it. First, start a new Visual Basic project with a single form, a button in the center of the form and the SocketWrench control. It might look something like this:



The first that we need to do is create a global variable called **LastSocket** which we will use to keep track of the number of clients that have connected to our server. This should be done in the general declaration section, as follows:

```
Dim LastSocket As Integer
```

Next, we will initialize the form in the Load subroutine with the following code:

```
Private Sub Form_Load()  
    Command1.Caption = "Listen"  
    LastSocket = 0  
End Sub
```

When the user presses the command button, we want the server to begin listening for connections. Remember that the first thing that a server application must do is listen on a local port for incoming connections from a client. You'll know that a client is attempting to connect with you when the **OnAccept** event is generated for the SocketWrench control.

To accept the connection, your program calls the **Accept** method, passing the listening socket handle as a parameter. As you'll recall from the TCP/IP tutorial, the act of accepting a connection causes a second socket to be created. The original listening socket continues to listen for more connections, while the second socket can be used to communicate with the client that connected to you. If you use the **Accept** method to accept the connection on the

same instance of the control, you're effectively telling the control to *close* the original listening socket and from that point on the control can be used to communicate with the client. While this is convenient, it is also limiting -- since the listening socket has been closed, no more clients can connect with your program, effectively limiting it to a single client connection.

A better approach is to create an additional instance of the control and have it accept the connection, leaving the original listening socket available so that more clients can establish a connection with your server. The problem is, how many clients are going to attempt to connect to you? Of course, you could drop a fixed number of SocketWrench controls on your form, thereby limiting the number of connections, but that's not a very good design. The better approach is to create a *control array* which can be dynamically loaded when a connection is attempted by a client, and unloaded when the connection is closed. This is the approach that we'll take in our echo server.

In order to implement a dynamically-loaded control array, set the Index property of SocketWrench1 to 0. This will also cause VB to include the parameter Index in events that you implement.

To have the server begin listening when the button is pressed, we need to add code to the button's **Click** event. Initially there will only be one instance of the control in our control array, identified as SocketWrench1(0) and it will be used to listen for connections:

```
Private Sub Command1_Click()  
    If Not SocketWrench1(0).Listening Then  
        Dim nError As Long  
  
        SocketWrench1(0).AutoResolve = False  
        SocketWrench1(0).Blocking = False  
        SocketWrench1(0).Protocol = swProtocolTcp  
        SocketWrench1(0).LocalPort = swPortEcho  
  
        nError = SocketWrench1(0).Listen()  
        If nError <> 0 Then  
            MsgBox "Unable to listen for connections", vbExclamation  
            Exit Sub  
        End If  
  
        Command1.Caption = "Disconnect"  
    Else  
        SocketWrench1(0).Disconnect  
        Command1.Caption = "Listen"  
    End If  
End Sub
```

There are two new properties here, the **Listening** property and the **LocalPort** property. The **Listening** property is a boolean flag, similar to the **Connected** property in our client example. It will return True if the control is currently listening for client connections. The **LocalPort** property is used by server applications to specify the local port that it's listening on for connections. By specifying the standard port used by echo servers (port 7), any other system can connect to yours and expect the program to echo back whatever is sent to it.

If the control is listening for connections and you press the button, it will disconnect the socket. This stops the control from listening for new client connections, however it will not interrupt any clients that have already connected to the server. The reason for this is because the client connections are actually managed on separate sockets which are not affected by closing the listening socket.

When our server program is executed and you press the button, the control will begin listening for client connections. When this occurs, the control's **OnAccept** event will fire. The code for this event should look like this:

```
Private Sub SocketWrench1_Accept(Index As Integer, ByVal Handle As Variant)
    Dim I As Integer

    For I = 1 To LastSocket
        If Not SocketWrench1(I).Connected Then Exit For
    Next I

    If I > LastSocket Then
        LastSocket = LastSocket + 1: I = LastSocket
        Load SocketWrench1(I)
    End If

    SocketWrench1(I).AutoResolve = False
    SocketWrench1(I).Blocking = False
    SocketWrench1(I).Protocol = swProtocolTcp
    SocketWrench1(I).Accept Handle
End Sub
```

Next, we initialize the control's properties, and then the Accept method is called with the Handle parameter that is passed to the control. After executing this statement, the control is now ready to start communicating with the client program. Since it's the job of an echo server to echo back whatever is sent to it, we have to add code to the control's **OnRead** event, which tells it that the client has sent some data to us:

```
Private Sub SocketWrench1_Read(Index As Integer)
    Dim strBuffer As String
    Dim cbBuffer As Long

    cbBuffer = SocketWrench1(Index).Read(strBuffer, 1024)
    If cbBuffer > 0 Then
        SocketWrench1(Index).Write strBuffer, cbBuffer
    End If
End Sub
```

Finally, when the client closes the connection, the socket control must also close its end of the connection. This is accomplished by adding a line of code in the control's **OnDisconnect** event:

```
Private Sub SocketWrench1_Disconnect(Index As Integer)
    SocketWrench1(Index).Disconnect
End Sub
```

To make sure that all of the socket connections are closed when the application is terminated, the following code should be included in the form's **Unload** event:

```
Private Sub Form_Unload (Cancel As Integer)
    Dim I As Integer
    If SocketWrench1(0).Listening Then SocketWrench1( 0).Disconnect
```

```
For I = 1 To LastSocket
  If SocketWrench1( I).Connected Then SocketWrench1( I).Disconnect
Next I
End
End Sub
```

This will disconnect the listening socket so that no more clients can establish connections, and will then disconnect from each of the clients.

C IMPLEMENTATION IN UNIX

SERVER CODE

```
#ifndef unix
#define WIN32
#include <windows.h>
#include <winsock.h>
#else
#define closesocket close
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#endif

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

#define PROTOPORT 6500 /* default protocol port number */
#define QLEN 6 /* size of request queue */

int visits = 0; /* counts client connections */
/*-----
 * Program: server
 *
 * Purpose: allocate a socket and then repeatedly execute the following:
 * (1) wait for the next connection from a client
 * (2) send a short message to the client
 * (3) close the connection
 * (4) go back to step (1)
 *
 * Syntax: server [ port ]
 *
 * port - protocol port number to use
 *
 * Note: The port argument is optional. If no port is specified,
 * the server uses the default given by PROTOPORT.
 *-----
 */
main(argc, argv)
int argc;
char *argv[];
{
    struct hostent *ptrh; /* pointer to a host table entry */

```

```

struct protoent *ptrp; /* pointer to a protocol table entry */
struct sockaddr_in sad; /* structure to hold server's address */
struct sockaddr_in cad; /* structure to hold client's address */
int sd, sd2; /* socket descriptors */
int port; /* protocol port number */
int alen; /* length of address */
char buf[1000]; /* buffer for string the server sends */
int n; /* number of characters received */

#ifdef WIN32
WSADATA wsaData;
WSAStartup(0x0101, &wsaData);
#endif
memset((char *)&sad,0,sizeof(sad)); /* clear sockaddr structure */
sad.sin_family = AF_INET; /* set family to Internet */
sad.sin_addr.s_addr = INADDR_ANY; /* set the local IP address */

/* Check command-line argument for protocol port and extract
/* port number if one is specified. Otherwise, use the default
/* port value given by constant PROTOPORT */

if (argc > 1) { /* if argument specified */
port = atoi(argv[1]); /* convert argument to binary */
} else {
port = PROTOPORT; /* use default port number */
}
if (port > 0) /* test for illegal value */
sad.sin_port = htons((u_short)port);
else { /* print error message and exit */
fprintf(stderr,"bad port number %s\n",argv[1]);
exit(1);
}

/* Map TCP transport protocol name to protocol number */

if ( ((int)(ptrp = getprotobyname("tcp"))) == 0) {
fprintf(stderr, "cannot map \"tcp\" to protocol number");
exit(1);
}

/* Create a socket */

sd = socket(PF_INET, SOCK_STREAM, ptrp->p_proto);
if (sd < 0) {
fprintf(stderr, "socket creation failed\n");
exit(1);
}

```

```

}

/* Bind a local address to the socket */

if (bind(sd, (struct sockaddr *)&sad, sizeof(sad)) < 0) {
    fprintf(stderr, "bind failed\n");
    exit(1);
}

/* Specify size of request queue */

if (listen(sd, QLEN) < 0) {
    fprintf(stderr, "listen failed\n");
    exit(1);
}

/* Main server loop - accept and handle requests */

while (1) {
    alen = sizeof(cad);
    if ((sd2=accept(sd, (struct sockaddr *)&cad, &alen)) < 0) {
        fprintf(stderr, "accept failed\n");
        exit(1);
    }
    n = recv(sd2, buf, sizeof(buf), 0);
    while (n > 0)
    {
        send(sd2, buf, n, 0);
        n = recv(sd2, buf, sizeof(buf), 0);
    }
    closesocket(sd2);
}
}

```

CLIENT CODE

```
#ifndef unix
#define WIN32
#include <windows.h>
#include <winsock.h>
#else
#define closesocket close
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#endif

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

#define PROTOPORT 6500 /* default protocol port number */

extern int errno;
char localhost[] = "localhost"; /* default host name */
/*-----
 * Program: client
 *
 * Purpose: allocate a socket, connect to a server, and print all output
 *
 * Syntax: client [ host [port] ]
 *
 * host - name of a computer on which server is executing
 * port - protocol port number server is using
 *
 * Note: Both arguments are optional. If no host name is specified,
 * the client uses "localhost"; if no protocol port is
 * specified, the client uses the default given by PROTOPORT.
 *-----
 */
main(argc, argv)
int argc;
char *argv[];
{
    struct hostent *ptrh; /* pointer to a host table entry */

```

```

struct protoent *ptrp; /* pointer to a protocol table entry */
struct sockaddr_in sad; /* structure to hold an IP address */
int sd; /* socket descriptor */
int port; /* protocol port number */
char *host; /* pointer to host name */
int n; /* number of characters read */
char buf[1000]; /* buffer for data from the server */
char *text; /* pointer to user's line of text */
#ifdef WIN32
WSADATA wsaData;
WSAStartup(0x0101, &wsaData);
#endif
memset((char *)&sad,0,sizeof(sad)); /* clear sockaddr structure */
sad.sin_family = AF_INET; /* set family to Internet */

/* Check command-line argument for protocol port and extract */
/* port number if one is specified. Otherwise, use the default */
/* port value given by constant PROTOPORT */

if (argc > 2) { /* if protocol port specified */
    port = atoi(argv[2]); /* convert to binary */
} else {
    port = PROTOPORT; /* use default port number */
}
if (port > 0) /* test for legal value */
    sad.sin_port = htons((u_short)port);
else { /* print error message and exit */
    fprintf(stderr,"bad port number %s\n",argv[2]);
    exit(1);
}

/* Check host argument and assign host name. */

if (argc > 1) {
    host = argv[1]; /* if host argument specified */
} else {
    host = localhost;
}

/* Convert host name to equivalent IP address and copy to sad. */

ptrh = gethostbyname(host);
if ( ((char *)ptrh) == NULL ) {
    fprintf(stderr,"invalid host: %s\n", host);
    exit(1);
}

```

```

memcpy(&sad.sin_addr, ptrh->h_addr, ptrh->h_length);

/* Map TCP transport protocol name to protocol number. */

if ( ((int)(ptrp = getprotobyname("tcp"))) == 0) {
    fprintf(stderr, "cannot map \"tcp\" to protocol number");
    exit(1);
}

/* Create a socket. */

sd = socket(PF_INET, SOCK_STREAM, ptrp->p_proto);
if (sd < 0) {
    fprintf(stderr, "socket creation failed\n");
    exit(1);
}

/* Connect the socket to the specified server. */

if (connect(sd, (struct sockaddr *)&sad, sizeof(sad)) < 0) {
    fprintf(stderr, "connect failed\n");
    exit(1);
}

/* Repeatedly read data from user and send it to server. */

text = fgets(buf, sizeof(buf), stdin);
while (text != NULL) {
    send(sd, buf, strlen(buf), 0);
    n = recv(sd, buf, sizeof(buf), 0);
    write(1, buf, n);
    text = fgets(buf, sizeof(buf), stdin);
}

/* Close the socket. */

closesocket(sd);

/* Terminate the client program gracefully. */

exit(0);
}

```

HELPER FILE

```
#  
# makefile for echo client and server  
#  
  
CC = gcc  
LIBS = -lsocket -lnsl -lpthread  
  
all: EchoServer EchoClient  
  
EchoServer: EchoServer.c  
    $(CC) -o EchoServer EchoServer.c $(LIBS)  
  
EchoClient: EchoClient.c  
    $(CC) -o EchoClient EchoClient.c $(LIBS)  
  
clean:  
    /bin/rm -f EchoServer EchoClient core *.o *~
```


EXECUTING THE CODE

SERVER

- 1) In the terminal window of Linux execute the command
gcc echoserver.c
./a.out
- 2) Make sure all the files are in root directory. This will start the server.

CLIENT

- 1) In a separate terminal window of Linux execute the command
gcc echoclient.c
./a.out
- 2) Make sure all the files are in root directory. This will start the client

SERVER STARTED

```
[root@localhost root]# gcc echoserver.c  
[root@localhost root]# ./a.out
```

CLIENT STARTED

```
[root@localhost root]# gcc echoclient.c  
[root@localhost root]# ./a.out
```

COMMUNICATING

```
[root@localhost root]# gcc  
echoclient.c  
[root@localhost root]# ./a.out  
hi  
hi  
confirm connection  
confirm connection  
connection confirmed  
connection confirmed  
executing  
executing  
success  
success
```

TELNET

```
[root@localhost root]# telnet 127.0.0.1 6500
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'
hi
hi
confirm connection
confirm connection
connection confirmed
connection confirmed
executing
executing
success
success
```

JAVA IMPLEMENTATION IN WINDOWS

SERVER CODE

```
/**
 * An echo server listening on the port specified in the command line.
 * This server reads from the client and echoes back the result. When the
 * client enters the character '.' the server closes the connection.
 */

import java.net.*;
import java.io.*;

public class EchoServer
{
    public static void main(String[] args) throws IOException {
        if (args.length < 1) {
            System.err.println("Usage: java EchoServer <Port Number> ");
            System.exit(0);
        }

        ServerSocket sock = null;

        try {
            // establish the socket
            sock = new ServerSocket(Integer.parseInt(args[0]));

            /**
             * listen for new connection requests.
             * when a request arrives, service it
             * and resume listening for more requests.
             */
            while (true) {
                // now listen for connections
                Socket client = sock.accept();

                // service the connection
                ServiceConnection(client);
            }
        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
        finally {
            if (sock != null)
                sock.close();
        }
    }
}
```

```

public static void ServiceConnection(Socket client)
{
    BufferedReader networkBin = null;
    OutputStreamWriter networkPout = null;

    try {
        /**
         * get the input and output streams associated with the socket.
         */
        networkBin = new BufferedReader(new
InputStreamReader(client.getInputStream()));
        networkPout = new OutputStreamWriter(client.getOutputStream());

        /**
         * the following successively reads from the input stream and returns
         * what was read. The loop terminates when we read a period "."
         * from the input stream.
         */
        boolean done = false;
        while (!done) {
            String line = networkBin.readLine();
            if ( (line == "") || line.equals(".") ) {
                done = true;
                networkPout.write("BYE\r\n");
            }
            else
                networkPout.write("[ "+line+" ]\r\n");

            networkPout.flush();
        }
    }
    catch (IOException ioe) {
        System.err.println(ioe);
    }
    finally {
        try {
            if (networkBin != null)
                networkBin.close();
            if (networkPout != null)
                networkPout.close();
            if (client != null)
                client.close();
        }
        catch (IOException ioee) {
            System.err.println(ioee);
        }
    }
}

```

```
        }  
    } // end try  
} // end ServiceConnection  
}
```


CLIENT CODE

```
/**
 * An echo client. The client enters data to the server, and the
 * server echoes the data back to the client.
 */

import java.net.*;
import java.io.*;

public class EchoClient
{
    public static void main(String[] args) throws IOException {
        if (args.length < 2) {
            System.err.println("Usage: java EchoClient <IP address> <Port
number>");
            System.exit(0);
        }

        BufferedReader networkBin = null;
        PrintWriter networkPout = null;
        BufferedReader localBin = null;
        Socket sock = null;

        try {
            sock = new Socket(args[0], Integer.parseInt(args[1]));

            // set up the necessary communication channels
            networkBin = new BufferedReader(new
InputStreamReader(sock.getInputStream()));
            localBin = new BufferedReader(new
InputStreamReader(System.in));
            networkPout = new PrintWriter(sock.getOutputStream(),true);

            boolean done = false;
            while (!done) {
                String line = localBin.readLine();
                if (line.equals("."))
                    done = true;
                networkPout.println(line);
                System.out.println("Server: " + networkBin.readLine());
            }
        }
        catch (IOException ioe) {
```

```
        System.err.println(ioe);
    }
    finally {
        if (networkBin != null)
            networkBin.close();
        if (localBin != null)
            localBin.close();
        if (networkPout != null)
            networkPout.close();
        if (sock != null)
            sock.close();
    }
}
}
```

EXECUTING THE CODE

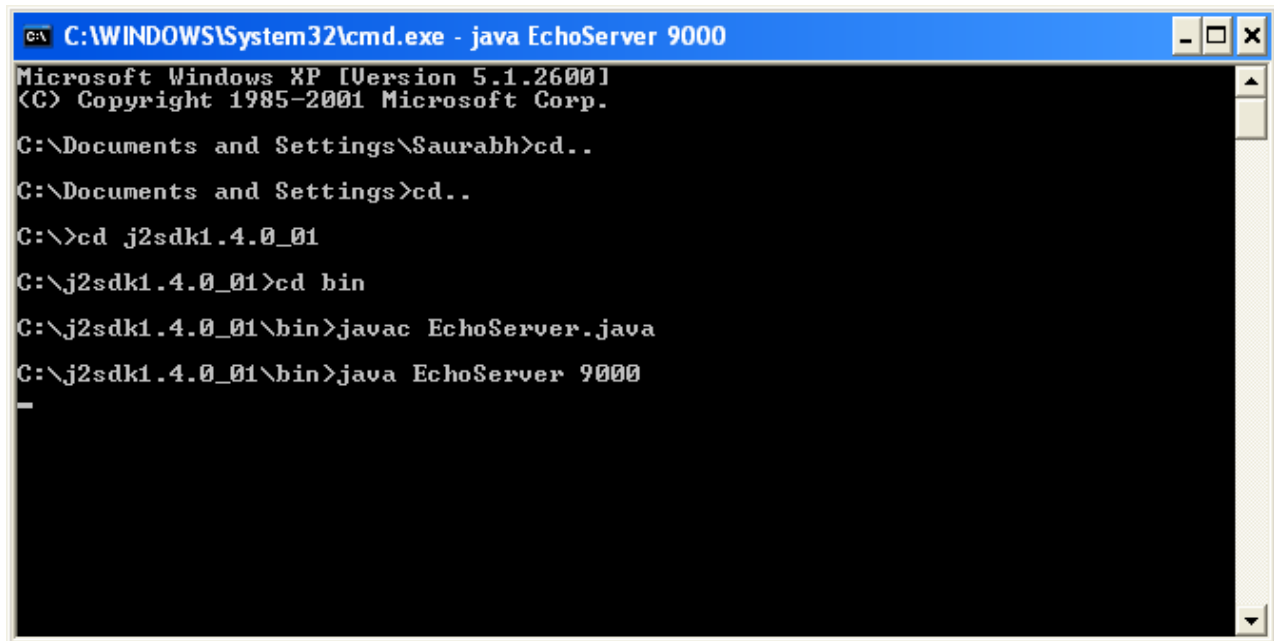
SERVER

- 1) In the command prompt execute the command
javac EchoServer.java
java EchoServer *any port number*
- 2) Server will be successfully started

CLIENT

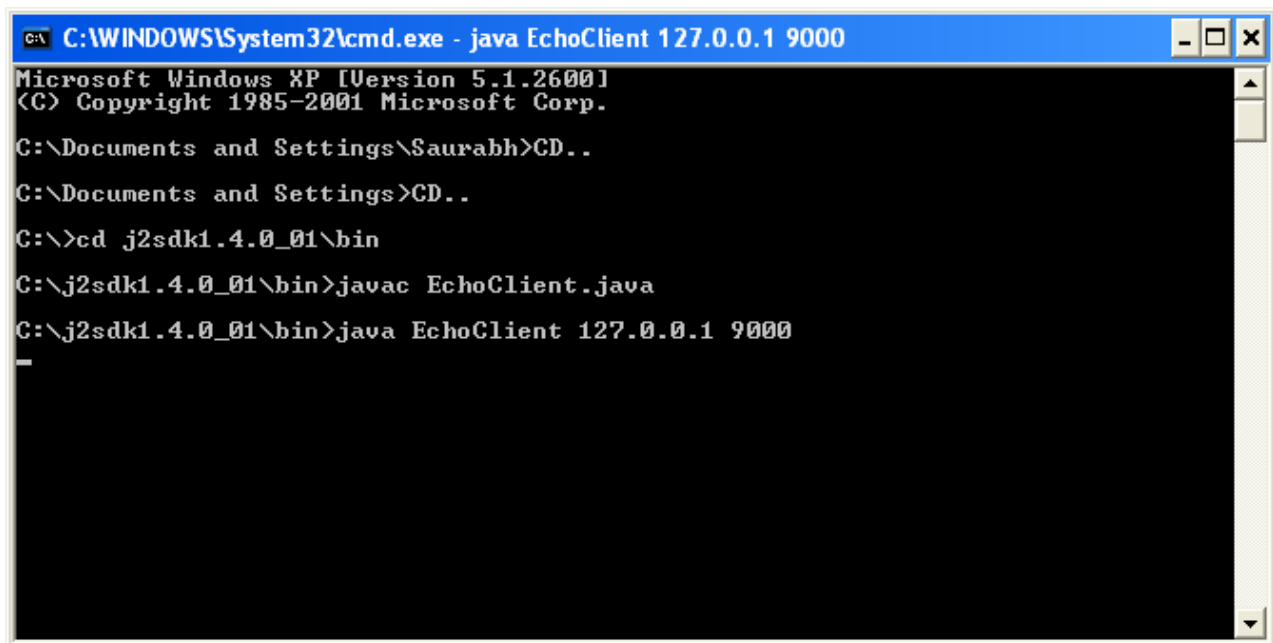
- 1) In the command prompt (separate window) execute the command
javac EchoClient.java
java EchoClient *hostaddress portnumber*
- 2) Client will be successfully started communicating with above server.

EXECUTION OF SERVER LISTENING TO PORT 9000



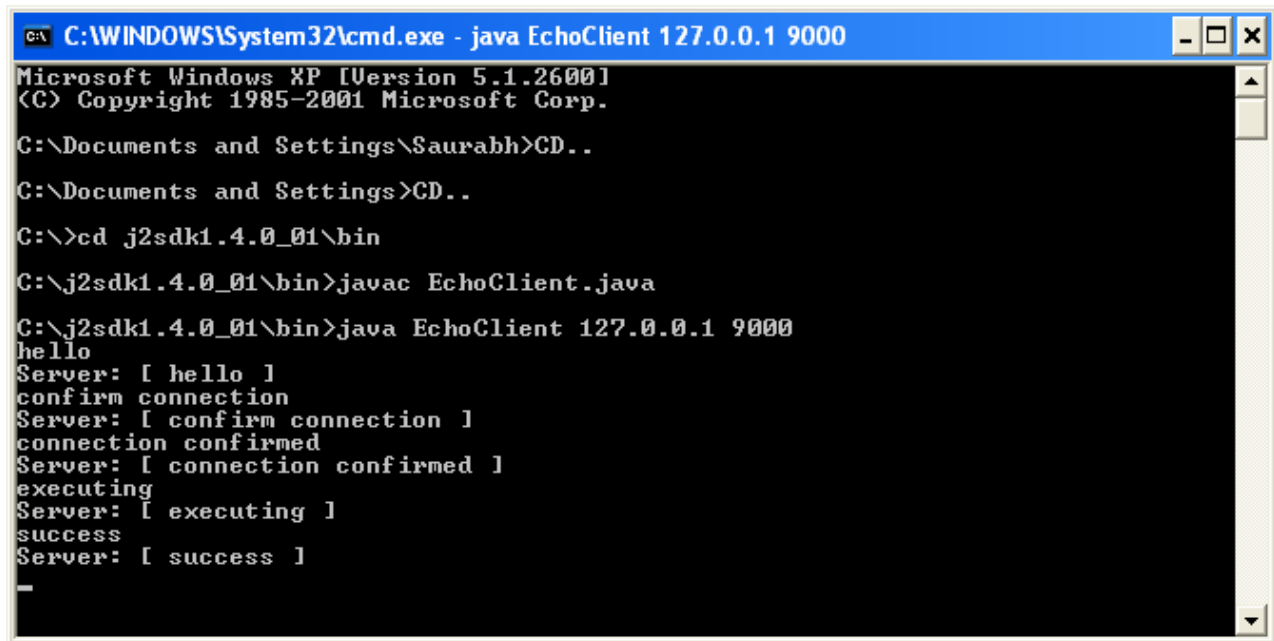
```
C:\WINDOWS\System32\cmd.exe - java EchoServer 9000
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
C:\Documents and Settings\Saurabh>cd..
C:\Documents and Settings>cd..
C:\>cd j2sdk1.4.0_01
C:\j2sdk1.4.0_01>cd bin
C:\j2sdk1.4.0_01\bin>javac EchoServer.java
C:\j2sdk1.4.0_01\bin>java EchoServer 9000
_
```

EXECUTION OF CLIENT RUNNING IN LOCAL HOST(127.0.0.1)
CONNECTED TO PORT 9000



```
C:\> C:\WINDOWS\System32\cmd.exe - java EchoClient 127.0.0.1 9000
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
C:\Documents and Settings\Saurabh>CD..
C:\Documents and Settings>CD..
C:\>cd j2sdk1.4.0_01\bin
C:\j2sdk1.4.0_01\bin>javac EchoClient.java
C:\j2sdk1.4.0_01\bin>java EchoClient 127.0.0.1 9000
-
```

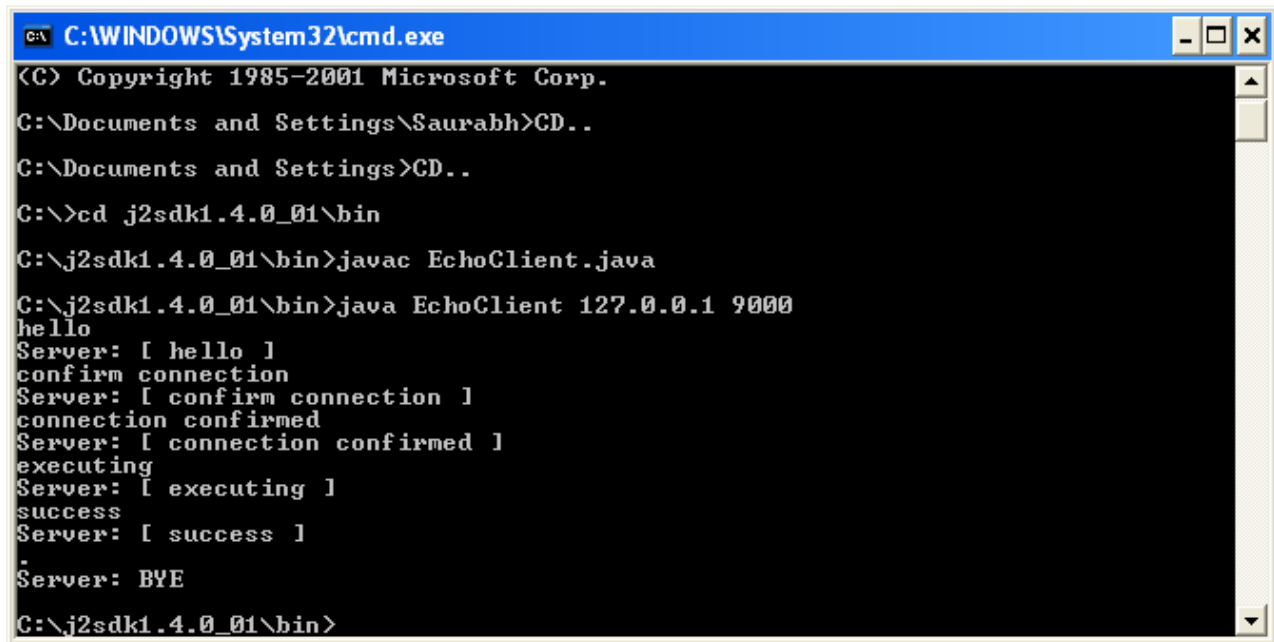
ECHO SERVER ECHOING BACK EVERY DATA IT RECEIVES BACK TO THE CLIENT



```
C:\WINDOWS\System32\cmd.exe - java EchoClient 127.0.0.1 9000
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Saurabh>CD..
C:\Documents and Settings>CD..
C:\>cd j2sdk1.4.0_01\bin
C:\j2sdk1.4.0_01\bin>javac EchoClient.java
C:\j2sdk1.4.0_01\bin>java EchoClient 127.0.0.1 9000
hello
Server: [ hello ]
confirm connection
Server: [ confirm connection ]
connection confirmed
Server: [ connection confirmed ]
executing
Server: [ executing ]
success
Server: [ success ]
-
```

CONNECTION TERMINATED WITH SERVER REPLYING "BYE"



```
C:\> C:\WINDOWS\System32\cmd.exe
(C) Copyright 1985-2001 Microsoft Corp.
C:\Documents and Settings\Saurabh>CD..
C:\Documents and Settings>CD..
C:\>cd j2sdk1.4.0_01\bin
C:\j2sdk1.4.0_01\bin>javac EchoClient.java
C:\j2sdk1.4.0_01\bin>java EchoClient 127.0.0.1 9000
hello
Server: [ hello ]
confirm connection
Server: [ confirm connection ]
connection confirmed
Server: [ connection confirmed ]
executing
Server: [ executing ]
success
Server: [ success ]
-
Server: BYE
C:\j2sdk1.4.0_01\bin>
```

VISUAL BASIC
IMPLEMENTATION
IN WINDOWS

SERVER CODE

frmServer Code (server.frm file)

Option Explicit

```
Private intLastSocket As Integer
Private strBufferArray() As String
Private lTotalConnections As Long
```

```
Private Sub Dropdown_Click()
lear the text box
    Text.Text = ""
End Sub
```

```
Private Sub Form_Load()
    'Set the status
    Me.Status.Panels(1).Text = "Ready."

    'Prepare our listening socket
    sockConn(0).AddressFamily = AF_INET
    sockConn(0).Protocol = IPPROTO_IP
    sockConn(0).SocketType = SOCK_STREAM
    sockConn(0).Blocking = False
    sockConn(0).AutoResolve = False
    sockConn(0).LocalPort = 7777
    sockConn(0).Listen
```

```
    'Set the last socket index to zero
    intLastSocket = 0
End Sub
```

```
Private Sub Form_Resize()
```

```
    'If the window can be resized...
    If Me.WindowState <> vbMinimized And _
        Me.Visible = True Then
```

```
        'Base control spacing on the dropdown's left position
        'The tops of controls is assumed not to change
        Dropdown.Width = Me.ScaleWidth - (Dropdown.Left * 3) - Kick.Width
        Kick.Left = Dropdown.Width + (Dropdown.Left * 2)
        Text.Width = Me.ScaleWidth - (Dropdown.Left * 2)
```

```
Text.Height = Me.ScaleHeight - (Text.Top + Dropdown.Left + Status.Height)
```

```
End If
```

```
End Sub
```

```
Private Sub Form_Unload(Cancel As Integer)
```

```
'Disconnect all sockets
```

```
DisconnectAll
```

```
End Sub
```

```
Private Sub Kick_Click()
```

```
Dim intIndex As Integer
```

```
Dim objSocket As Integer
```

```
'Loop through the sockets
```

```
For objSocket = 1 To intLastSocket
```

```
'If this socket is connected...
```

```
If sockConn(objSocket).Connected Then
```

```
'Loop through the dropdown
```

```
For intIndex = 0 To Dropdown.ListCount - 1
```

```
'If the handle matches
```

```
If sockConn(objSocket).Handle = Dropdown.ItemData(intIndex) Then
```

```
'And if the index matches
```

```
If Dropdown.ListIndex = intIndex Then
```

```
'Disconnect this user
```

```
Log "KICK: " & sockConn(objSocket).PeerAddress
```

```
sockConn_Disconnect objSocket
```

```
Exit For
```

```
End If
```

```
End If
```

```
Next 'intIndex
```

```
End If
```

```
Next 'objSocket
```

```
End Sub
```

```
Private Function NewSocket() As Integer
```

```
Dim intIndex As Integer
```

```
'Look for the first available socket control
```

```
For intIndex = 1 To intLastSocket
    If Not sockConn(intIndex).Connected Then
        strBufferArray(intIndex) = ""
        Exit For
    End If
Next 'intIndex
```

```
'If we need to add another socket control...
If intIndex > intLastSocket Then
```

```
    'Create the control and increase the buffer array
    intLastSocket = intLastSocket + 1
    intIndex = intLastSocket
    Load sockConn(intIndex)
    ReDim strBufferArray(intIndex + 1) As String
```

```
End If
```

```
'Setup the new connection
sockConn(intIndex).AddressFamily = AF_INET
sockConn(intIndex).Protocol = IPPROTO_IP
sockConn(intIndex).SocketType = SOCK_STREAM
sockConn(intIndex).LocalPort = IPPORT_ANY
sockConn(intIndex).Binary = True
sockConn(intIndex).BufferSize = 1024
sockConn(intIndex).Blocking = False
sockConn(intIndex).AutoResolve = False
```

```
'Increase the number of connections and set the status
lTotalConnections = lTotalConnections + 1
setStatus
```

```
'Return the new index
NewSocket = intIndex
End Function
```

```
Private Sub sockConn_Accept(Index As Integer, SocketId As Integer)
    Dim objSocket As Integer
```

```
    'Create a new socket connection
    'and free the listening socket
    objSocket = NewSocket()
    sockConn(objSocket).Accept = SocketId
End Sub
```

```
Private Sub sockConn_Connect(Index As Integer)
```

```

'If this address is not banned...
If Not IsBanned(sockConn(Index).PeerAddress) Then

    'Add the connection to the drop down
    Dropdown.AddItem sockConn(Index).PeerAddress
    Dropdown.ItemData(Dropdown.NewIndex) = sockConn(Index).Handle
    Dropdown.Enabled = True

    'If nothing is selected, select this connection
    If Dropdown.ListIndex = -1 Then
        Dropdown.ListIndex = Dropdown.NewIndex
    End If

End If

'Display the connection
DisplayCommand "Connection from: " & sockConn(Index).PeerAddress, Index

End Sub

Private Sub sockConn_Disconnect(Index As Integer)
    Dim intListIndex As Integer

    'Loop through the dropdown items
    For intListIndex = 0 To Dropdown.ListCount - 1

        'If we found the matching connection...
        If Dropdown.ItemData(intListIndex) = sockConn(Index).Handle Then

            'Remove the item from the drop down
            Dropdown.RemoveItem intListIndex

            'If there are no more drop down items...
            If Dropdown.ListCount < 1 Then

                'Disable the drop down
                Dropdown.ListIndex = -1
                Dropdown.Enabled = False

            End If

            'Exit the loop
            Exit For

        End If

    End For

End Sub

```

Next 'intListIndex

'Decrease the total number of connections
lTotalConnections = lTotalConnections - 1

'Show the disconnect
DisplayCommand "Disconnected: " & sockConn(Index).PeerAddress, Index
SetStatus

'Disconnect
sockConn(Index).Disconnect

'Clear the text box
Text.Text = ""

End Sub

Private Sub sockConn_LastError(Index As Integer, ErrorCode As Integer, ErrorString As String, Response As Integer)

'Show the error and disconnect
'MsgBox ErrorString, vbExclamation
Log "Error: [" & sockConn(Index).PeerAddress & "]" & ErrorString
sockConn_Disconnect Index

End Sub

Private Sub sockConn_Read(Index As Integer, DataLength As Integer, IsUrgent As Integer)

Dim strBuffer As String
Dim strTemp As String
Dim strChar As String
Dim lngPos As Long
Dim lngChar As Long

'Read the data
Call sockConn(Index).Read(strBuffer, 1024)

'Save the text but skip any Chr\$(10)
strBufferArray(Index) = strBufferArray(Index) & _
Replace(strBuffer, "", Chr\$(10))

'Did we encounter an end of line? - Chr\$(13)
lngPos = InStr(strBufferArray(Index), Chr\$(13))
Do Until lngPos = 0

'Get the command
strTemp = Left\$(strBufferArray(Index), lngPos - 1)

```

'Clip the command off the buffer
strBufferArray(Index) = Mid$(strBufferArray(Index), lngPos + 2)

'Display the command
DisplayCommand " IN: " & strTemp, Index

'If this socket is connected...
If sockConn(Index).Connected Then

    'Execute the command
    strTemp = ExecuteCommand(strTemp, sockConn(Index).PeerAddress)

    'Display the results
    DisplayCommand "OUT: " & strTemp, Index

    'Send the results back to the client
    strTemp = strTemp & vbCrLf
    For lngChar = 1 To Len(strTemp)
        strChar = Mid$(strTemp, lngChar, 1)
        Call sockConn(Index).Write(strChar, 1)
    Next 'lngChar

End If

'Try to find another command
lngPos = InStr(strBufferArray(Index), Chr$(13))
Loop

```

End Sub

```

Private Sub SetStatus()
    'Set the status based on the number of connections
    If lTotalConnections > 1 Then
        Me.Status.Panels(1).Text = CStr(lTotalConnections) & " connections"
    ElseIf lTotalConnections = 1 Then
        Me.Status.Panels(1).Text = "1 connection"
    Else
        Me.Status.Panels(1).Text = "No connections"
    End If
End Sub

```

```

Private Sub Status_PanelDbClick(ByVal Panel As MSComctlLib.Panel)
    'Close all connections
    If MsgBox("Close all connections?", vbQuestion + vbYesNo) = vbYes Then
        DisconnectAll
    End If
End Sub

```

```
End If
End Sub
```

```
Private Sub DisconnectAll()
    Dim objSocket As Integer
```

```
    'Disconnect the listener
    sockConn(0).Disconnect
```

```
    'Loop through and disconnect all socket connections
```

```
    For objSocket = 1 To intLastSocket
        If sockConn(objSocket).Connected Then
            sockConn(objSocket).Disconnect
        End If
    Next 'objSocket
```

```
    'Set the status
    lTotalConnections = 0
    SetStatus
```

```
End Sub
```

```
Private Sub DisplayCommand(Command As String, Index As Integer)
```

```
    'If we are looking at this connection...
    If Dropdown.List(Dropdown.ListIndex) = CStr(sockConn(Index).PeerAddress) Then
```

```
        'If the text is too large for the text box
        If Len(Text.Text) > 65000 Then
```

```
            'Trim 1000 characters off the front
            Text.Text = Mid$(Text.Text, 1000)
```

```
        End If
```

```
        'Append the text to the text box
        Text.SelStart = 65535
        Text.SelLength = 0
        Text.SelText = Command & vbCrLf
```

```
    End If
```

```
    'Log the text
    Log Command
```

```
End Sub
```

```
'-----
```

' To Do: Fill in the code below

'-----

Private Function IsBanned(Address As String) As Boolean

'This is where you would check for banned addresses

'Right now, let all connections come through

IsBanned = False

End Function

Private Function ExecuteCommand(Command As String, Address As String) As String

'This is where you would execute a command and return a string

'Right now, return the command as an echo

ExecuteCommand = "[" & Address & "]" & Command

End Function

Private Sub Log(Value As String)

'Code to log all text goes here

End Sub

Constants (constants.bas file)

Option Explicit

' General constants used with most of the controls

Global Const INVALID_HANDLE = -1

Global Const CONTROL_ERRIGNORE = 0

Global Const CONTROL_ERRDISPLAY = 1

' SocketWrench Control Actions

Global Const SOCKET_OPEN = 1

Global Const SOCKET_CONNECT = 2

Global Const SOCKET_LISTEN = 3

Global Const SOCKET_ACCEPT = 4

Global Const SOCKET_CANCEL = 5

Global Const SOCKET_FLUSH = 6

Global Const SOCKET_CLOSE = 7

Global Const SOCKET_DISCONNECT = 7

Global Const SOCKET_ABORT = 8

Global Const SOCKET_STARTUP = 9

Global Const SOCKET_CLEANUP = 10

' SocketWrench Control States

Global Const SOCKET_NONE = 0

Global Const SOCKET_IDLE = 1

Global Const SOCKET_LISTENING = 2

Global Const SOCKET_CONNECTING = 3

Global Const SOCKET_ACCEPTING = 4

Global Const SOCKET_RECEIVING = 5

Global Const SOCKET_SENDING = 6

Global Const SOCKET_CLOSING = 7

' Socket Address Families

Global Const AF_UNSPEC = 0

Global Const AF_UNIX = 1

Global Const AF_INET = 2

' Socket Types

Global Const SOCK_STREAM = 1

Global Const SOCK_DGRAM = 2

Global Const SOCK_RAW = 3

Global Const SOCK_RDM = 4

Global Const SOCK_SEQPACKET = 5

' Protocol Types

Global Const IPPROTO_IP = 0
Global Const IPPROTO_ICMP = 1
Global Const IPPROTO_GGP = 2
Global Const IPPROTO_TCP = 6
Global Const IPPROTO_PUP = 12
Global Const IPPROTO_UDP = 17
Global Const IPPROTO_IDP = 22
Global Const IPPROTO_ND = 77
Global Const IPPROTO_RAW = 255
Global Const IPPROTO_MAX = 256

' Well-Known Port Numbers

Global Const IPPORT_ANY = 0
Global Const IPPORT_ECHO = 7
Global Const IPPORT_DISCARD = 9
Global Const IPPORT_SYSTAT = 11
Global Const IPPORT_DAYTIME = 13
Global Const IPPORT_NETSTAT = 15
Global Const IPPORT_CHARGEN = 19
Global Const IPPORT_FTP = 21
Global Const IPPORT_TELNET = 23
Global Const IPPORT_SMTP = 25
Global Const IPPORT_TIMESERVER = 37
Global Const IPPORT_NAMESERVER = 42
Global Const IPPORT_WHOIS = 43
Global Const IPPORT_MTP = 57
Global Const IPPORT_TFTP = 69
Global Const IPPORT_FINGER = 79
Global Const IPPORT_HTTP = 80
Global Const IPPORT_POP3 = 110
Global Const IPPORT_NNTP = 119
Global Const IPPORT_SNMP = 161
Global Const IPPORT_EXEC = 512
Global Const IPPORT_LOGIN = 513
Global Const IPPORT_SHELL = 514
Global Const IPPORT_RESERVED = 1024
Global Const IPPORT_USERRESERVED = 5000

' Network Addresses

Global Const INADDR_ANY = "0.0.0.0"
Global Const INADDR_LOOPBACK = "127.0.0.1"
Global Const INADDR_NONE = "255.255.255.255"

' Shutdown Values

Global Const SOCKET_READ = 0
Global Const SOCKET_WRITE = 1

Global Const SOCKET_READWRITE = 2

' Byte Order

Global Const LOCAL_BYTE_ORDER = 0

Global Const NETWORK_BYTE_ORDER = 1

' SocketWrench Error Response

Global Const SOCKET_ERRIGNORE = 0

Global Const SOCKET_ERRDISPLAY = 1

' SocketWrench Error Codes

Global Const WSABASEERR = 24000

Global Const WSAEINTR = 24004

Global Const WSAEBADF = 24009

Global Const WSAEACCES = 24013

Global Const WSAEFAULT = 24014

Global Const WSAEINVAL = 24022

Global Const WSAEMFILE = 24024

Global Const WSAEWOULDBLOCK = 24035

Global Const WSAEINPROGRESS = 24036

Global Const WSAEALREADY = 24037

Global Const WSAENOTSOCK = 24038

Global Const WSAEDESTADDRREQ = 24039

Global Const WSAEMSGSIZE = 24040

Global Const WSAEPROTOTYPE = 24041

Global Const WSAENOPROTOOPT = 24042

Global Const WSAEPROTONOSUPPORT = 24043

Global Const WSAESOCKTNOSUPPORT = 24044

Global Const WSAEOPNOTSUPP = 24045

Global Const WSAEPFNOSUPPORT = 24046

Global Const WSAEAFNOSUPPORT = 24047

Global Const WSAEADDRINUSE = 24048

Global Const WSAEADDRNOTAVAIL = 24049

Global Const WSAENETDOWN = 24050

Global Const WSAENETUNREACH = 24051

Global Const WSAENETRESET = 24052

Global Const WSAECONNABORTED = 24053

Global Const WSAECONNRESET = 24054

Global Const WSAENOBUFS = 24055

Global Const WSAEISCONN = 24056

Global Const WSAENOTCONN = 24057

Global Const WSAESHUTDOWN = 24058

Global Const WSAETOOMANYREFS = 24059

Global Const WSAETIMEDOUT = 24060

Global Const WSAECONNREFUSED = 24061

Global Const WSAELOOP = 24062

Global Const WSAENAMETOOLONG = 24063
Global Const WSAEHOSTDOWN = 24064
Global Const WSAEHOSTUNREACH = 24065
Global Const WSAENOTEMPTY = 24066
Global Const WSAEPROCLIM = 24067
Global Const WSAEUSERS = 24068
Global Const WSAEDQUOT = 24069
Global Const WSAESTALE = 24070
Global Const WSAEREMOTE = 24071
Global Const WSASYSNOTREADY = 24091
Global Const WSAVERNOTSUPPORTED = 24092
Global Const WSANOTINITIALISED = 24093
Global Const WSAHOST_NOT_FOUND = 25001
Global Const WSATRY_AGAIN = 25002
Global Const WSANO_RECOVERY = 25003
Global Const WSANO_DATA = 25004
Global Const WSANO_ADDRESS = 25004

CLIENT CODE

frmClient Code (client.frm file)

Option Explicit

Private Sub cmdSend_Click()

Dim strTemp As String

strTemp = txtInput.Text & vbCrLf

Call sockConn.Write(strTemp, Len(strTemp) + 2)

txtInput.Text = ""

txtInput.SetFocus

End Sub

Private Sub Form_Load()

'Setup the new connection

sockConn.AddressFamily = AF_INET

sockConn.Protocol = IPPROTO_IP

sockConn.SocketType = SOCK_STREAM

sockConn.LocalPort = IPPORT_ANY

sockConn.Binary = True

sockConn.BufferSize = 1024

sockConn.Blocking = False

sockConn.AutoResolve = False

frmConnect.Show vbModal

If frmConnect.Cancel Then

End

End If

sockConn.HostName = frmConnect.Address

sockConn.RemoteService = frmConnect.Port

sockConn.Connect

Unload frmConnect

End Sub

Private Sub Form_QueryUnload(Cancel As Integer, UnloadMode As Integer)

```
sockConn.Disconnect  
End Sub
```

```
Private Sub sockConn_Disconnect()  
    MsgBox "Disconnected!"  
    Unload Me  
End Sub
```

```
Private Sub sockConn_Read(DataLength As Integer, IsUrgent As Integer)  
    Dim strBuffer As String  
  
    Call sockConn.Read(strBuffer, DataLength)  
    txtOutput.SelStart = 65535  
    txtOutput.SelLength = 0  
    txtOutput.SelText = strBuffer  
End Sub
```

frmConnect Code (connect.frm file)

Option Explicit

Private blnCancel As Boolean

Private strAddress As String

Private lngPort As Long

Public Property Get Cancel() As Boolean

 Cancel = blnCancel

End Property

Public Property Let Cancel(Value As Boolean)

 blnCancel = Value

End Property

Public Property Get Address() As String

 Address = txtAddress.Text & ""

End Property

Public Property Let Address(Value As String)

 txtAddress.Text = Value & ""

End Property

Public Property Get Port() As Long

 If IsNumeric(txtPort.Text & "") Then

 Port = CLng(txtPort.Text)

 Else

 Port = 7777

 End If

End Property

Public Property Let Port(Value As Long)

 txtPort.Text = CStr(Value)

End Property

Private Sub cmdCancel_Click()

 Cancel = True

 Me.Visible = False

End Sub

```
Private Sub cmdConnect_Click()  
    If Me.Address() = "" Then  
        MsgBox "Please enter a valid address."  
    Else  
        Cancel = False  
        Me.Visible = False  
    End If  
End Sub
```


Constants (constants.bas file)

Option Explicit

' General constants used with most of the controls

Global Const INVALID_HANDLE = -1

Global Const CONTROL_ERRIGNORE = 0

Global Const CONTROL_ERRDISPLAY = 1

' SocketWrench Control Actions

Global Const SOCKET_OPEN = 1

Global Const SOCKET_CONNECT = 2

Global Const SOCKET_LISTEN = 3

Global Const SOCKET_ACCEPT = 4

Global Const SOCKET_CANCEL = 5

Global Const SOCKET_FLUSH = 6

Global Const SOCKET_CLOSE = 7

Global Const SOCKET_DISCONNECT = 7

Global Const SOCKET_ABORT = 8

Global Const SOCKET_STARTUP = 9

Global Const SOCKET_CLEANUP = 10

' SocketWrench Control States

Global Const SOCKET_NONE = 0

Global Const SOCKET_IDLE = 1

Global Const SOCKET_LISTENING = 2

Global Const SOCKET_CONNECTING = 3

Global Const SOCKET_ACCEPTING = 4

Global Const SOCKET_RECEIVING = 5

Global Const SOCKET_SENDING = 6

Global Const SOCKET_CLOSING = 7

' Socket Address Families

Global Const AF_UNSPEC = 0

Global Const AF_UNIX = 1

Global Const AF_INET = 2

' Socket Types

Global Const SOCK_STREAM = 1

Global Const SOCK_DGRAM = 2

Global Const SOCK_RAW = 3

Global Const SOCK_RDM = 4

Global Const SOCK_SEQPACKET = 5

' Protocol Types

Global Const IPPROTO_IP = 0
Global Const IPPROTO_ICMP = 1
Global Const IPPROTO_GGP = 2
Global Const IPPROTO_TCP = 6
Global Const IPPROTO_PUP = 12
Global Const IPPROTO_UDP = 17
Global Const IPPROTO_IDP = 22
Global Const IPPROTO_ND = 77
Global Const IPPROTO_RAW = 255
Global Const IPPROTO_MAX = 256

' Well-Known Port Numbers

Global Const IPPORT_ANY = 0
Global Const IPPORT_ECHO = 7
Global Const IPPORT_DISCARD = 9
Global Const IPPORT_SYSTAT = 11
Global Const IPPORT_DAYTIME = 13
Global Const IPPORT_NETSTAT = 15
Global Const IPPORT_CHARGEN = 19
Global Const IPPORT_FTP = 21
Global Const IPPORT_TELNET = 23
Global Const IPPORT_SMTP = 25
Global Const IPPORT_TIMESERVER = 37
Global Const IPPORT_NAMESERVER = 42
Global Const IPPORT_WHOIS = 43
Global Const IPPORT_MTP = 57
Global Const IPPORT_TFTP = 69
Global Const IPPORT_FINGER = 79
Global Const IPPORT_HTTP = 80
Global Const IPPORT_POP3 = 110
Global Const IPPORT_NNTP = 119
Global Const IPPORT_SNMP = 161
Global Const IPPORT_EXEC = 512
Global Const IPPORT_LOGIN = 513
Global Const IPPORT_SHELL = 514
Global Const IPPORT_RESERVED = 1024
Global Const IPPORT_USERRESERVED = 5000

' Network Addresses

Global Const INADDR_ANY = "0.0.0.0"
Global Const INADDR_LOOPBACK = "127.0.0.1"
Global Const INADDR_NONE = "255.255.255.255"

' Shutdown Values

Global Const SOCKET_READ = 0
Global Const SOCKET_WRITE = 1
Global Const SOCKET_READWRITE = 2

' Byte Order

Global Const LOCAL_BYTE_ORDER = 0
Global Const NETWORK_BYTE_ORDER = 1

' SocketWrench Error Response

Global Const SOCKET_ERRIGNORE = 0
Global Const SOCKET_ERRDISPLAY = 1

' SocketWrench Error Codes

Global Const WSABASEERR = 24000
Global Const WSAEINTR = 24004
Global Const WSAEBADF = 24009
Global Const WSAEACCES = 24013
Global Const WSAEFAULT = 24014
Global Const WSAEINVAL = 24022
Global Const WSAEMFILE = 24024
Global Const WSAEWOULDBLOCK = 24035
Global Const WSAEINPROGRESS = 24036
Global Const WSAEALREADY = 24037
Global Const WSAENOTSOCK = 24038
Global Const WSAEDESTADDRREQ = 24039
Global Const WSAEMSGSIZE = 24040
Global Const WSAEPROTOTYPE = 24041
Global Const WSAENOPROTOOPT = 24042
Global Const WSAEPROTONOSUPPORT = 24043
Global Const WSAESOCKTNOSUPPORT = 24044
Global Const WSAEOPNOTSUPP = 24045
Global Const WSAEPFNOSUPPORT = 24046
Global Const WSAEAFNOSUPPORT = 24047
Global Const WSAEADDRINUSE = 24048
Global Const WSAEADDRNOTAVAIL = 24049
Global Const WSAENETDOWN = 24050
Global Const WSAENETUNREACH = 24051
Global Const WSAENETRESET = 24052
Global Const WSAECONNABORTED = 24053
Global Const WSAECONNRESET = 24054
Global Const WSAENOBUFS = 24055
Global Const WSAEISCONN = 24056
Global Const WSAENOTCONN = 24057
Global Const WSAESHUTDOWN = 24058
Global Const WSAETOOMANYREFS = 24059
Global Const WSAETIMEDOUT = 24060

Global Const WSAECONNREFUSED = 24061
Global Const WSAELOOP = 24062
Global Const WSAENAMETOOLONG = 24063
Global Const WSAEHOSTDOWN = 24064
Global Const WSAEHOSTUNREACH = 24065
Global Const WSAENOTEMPTY = 24066
Global Const WSAEPROCLIM = 24067
Global Const WSAEUSERS = 24068
Global Const WSAEDQUOT = 24069
Global Const WSAESTALE = 24070
Global Const WSAEREMOTE = 24071
Global Const WSASYSNOTREADY = 24091
Global Const WSAVERNOTSUPPORTED = 24092
Global Const WSANOTINITIALISED = 24093
Global Const WSAHOST_NOT_FOUND = 25001
Global Const WSATRY_AGAIN = 25002
Global Const WSANO_RECOVERY = 25003
Global Const WSANO_DATA = 25004
Global Const WSANO_ADDRESS = 25004

EXECUTING THE CODE

Make sure that socket wrench library is installed in VB otherwise the code wont run.

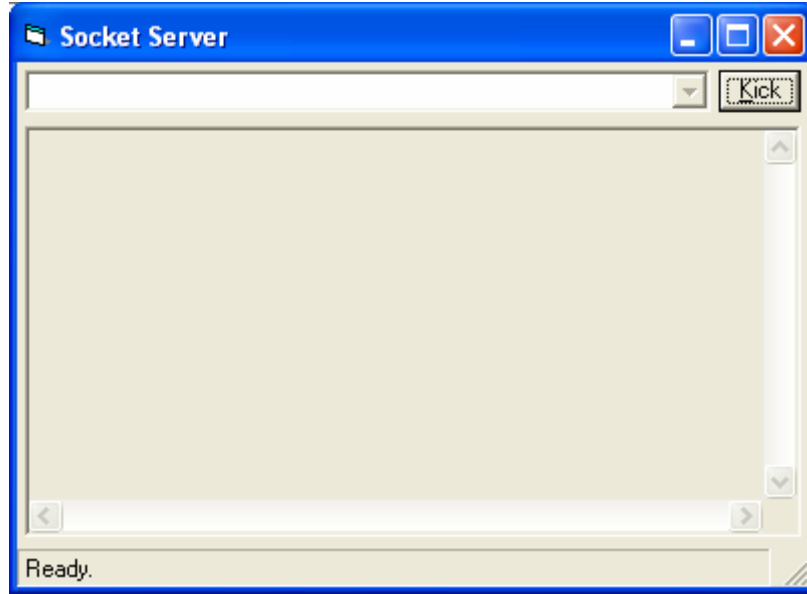
SERVER

- 1) Run the server project file.
- 2) It will show a server waiting for clients.

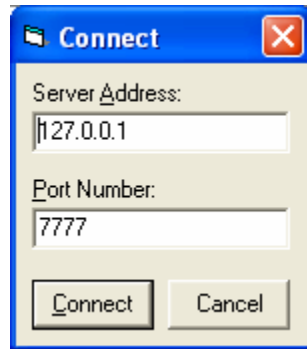
CLIENT

- 1) Run the client project file.
- 2) It will show a client with a connect button.
- 3) Specify the host address and port number and press connect.

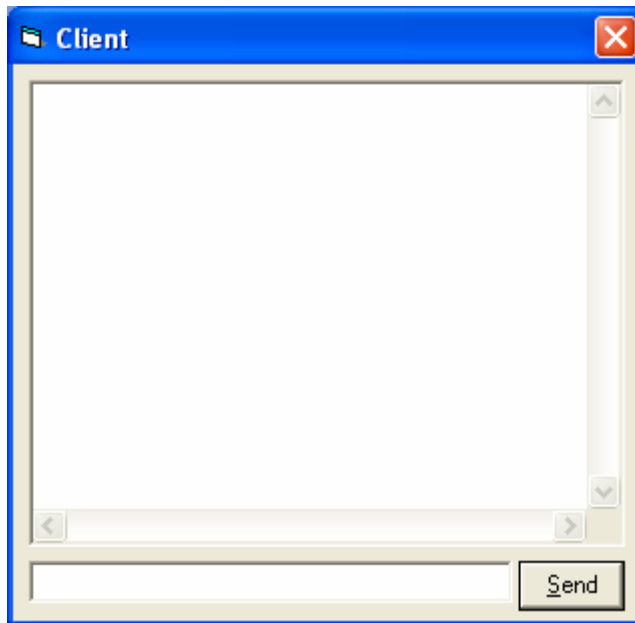
SERVER STARTED



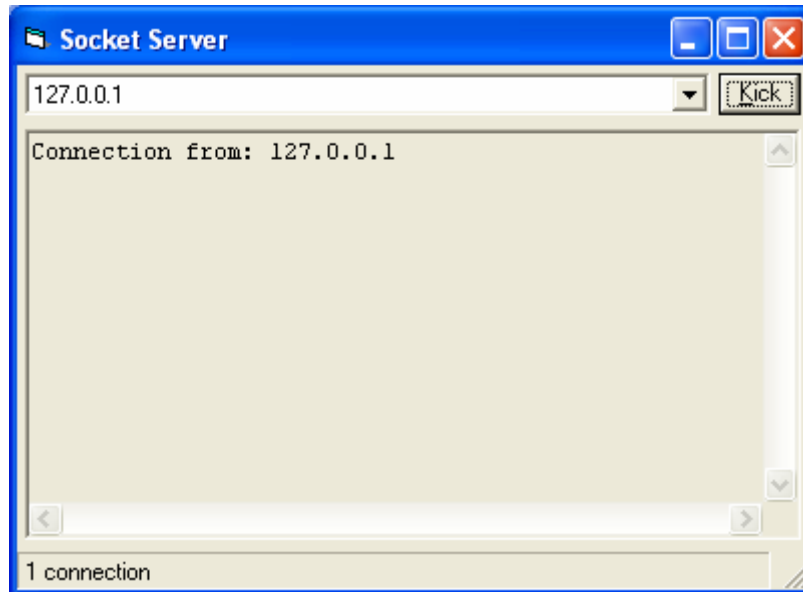
CLIENT STARTED



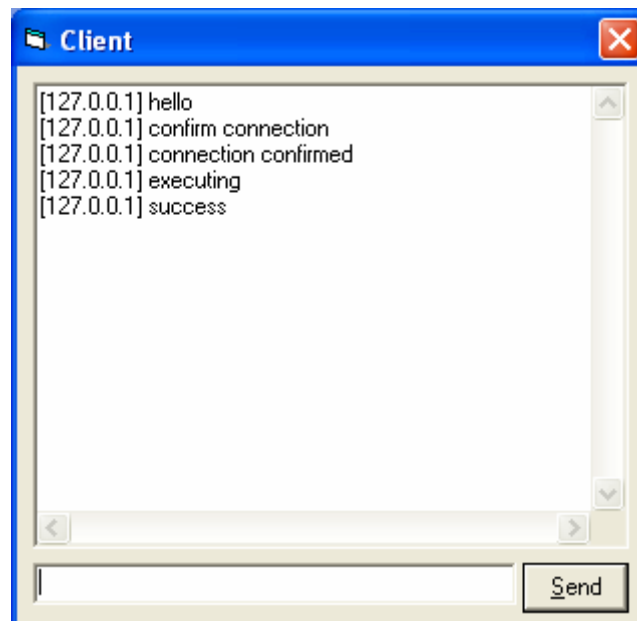
CLIENT CONNECTED



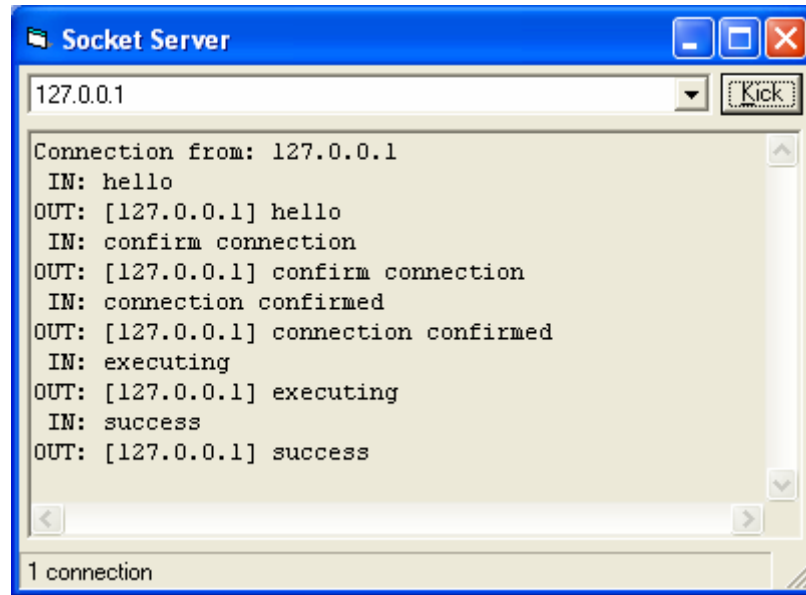
SERVER ACCEPTED THE CONNECTION



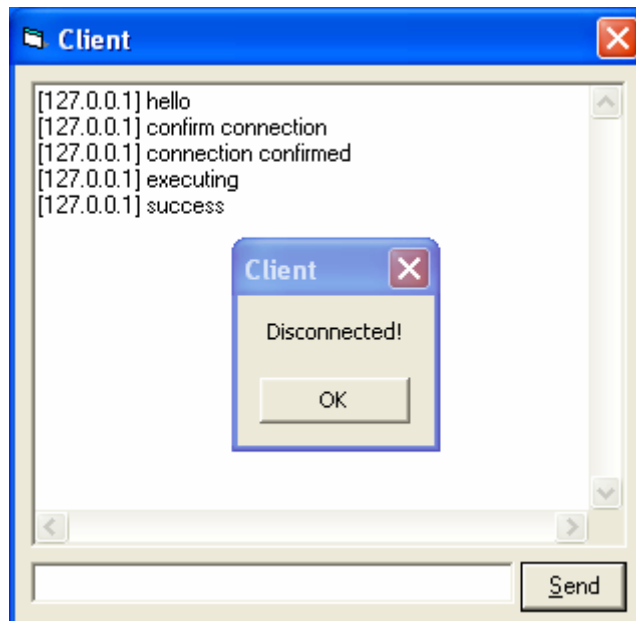
CLIENT COMMUNICATING WITH SERVER



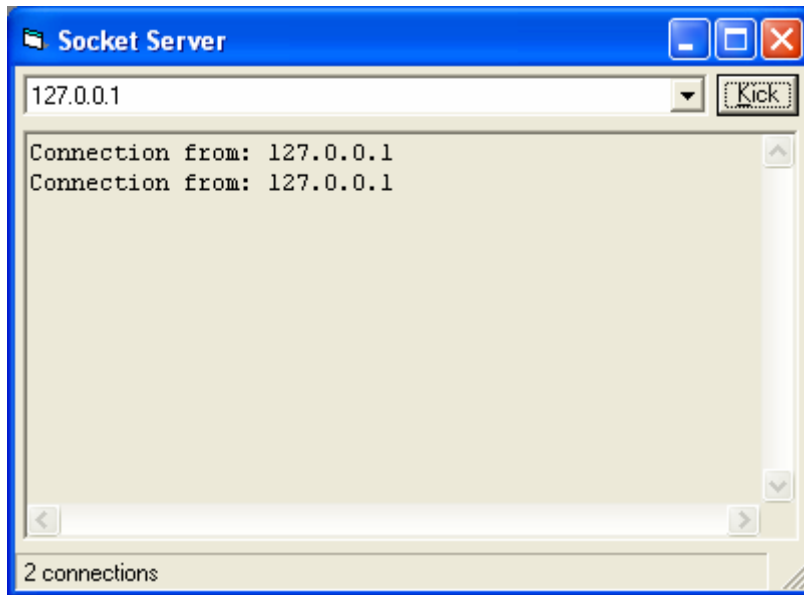
SERVER ECHOING BACK THE CLIENT DATA



CONNECTION TERMINATION



SERVER HANDLING TWO CLIENTS



SYSTEM CONFIGURATION

The following System Configuration is used

- 1) Pentium III 933MHZ
- 2) 256MBRAM
- 3) WINDOWS XP
- 4) RED HAT LINUX 9.0
- 5) VISUAL BASIC 6.0
- 6) JAVA (J2SDK 1.4)
- 7) WINSOCK LIBRARY FOR V.B.

*RESEARCH BASED
APPLICATION*

APPLICATION

We suggest the following research based application of the Echo Server.

Studying of the Impact of the Internet Audio Transmission using different parameters

Basic Theory :

The human ear is much more receptive to the audio signals as compared to video signals. With the arrival of different methods to transmit audio signals over networks it has become important to study their quality of service.

The quality of the audio in IP telephony depends upon and is influenced by various parameters like distance, delay , loss , errors. Hence the quality of audio transmitted significantly depends upon the above factors. We can conduct an experiment to measure these different parameters and hence their impact on the audio quality.

Experiment :

After studying the various parameters an experimental set up can be suggested. A setup can be designed to study the correlation of loss , and various RTT (round trip time) measurements , jitters and out of order packets.

Steps Involved :

- 1) The audio signal is encoded and sent over a channel to echo server.
- 2) The signal is time stamped and given a sequence number.
- 3) Various parameters like packets lost , time delay ,distance etc are noted and graphs are made between different parameters
- 4) The experiment is usually conducted over a long period of time so as to average the data. This increases the accuracy of the results.
- 5) Graphs are interpreted.

Results and Observations :

- 1) RTT increase is an indicator of increase in packet loss but this graph is not linear.
- 2) In addition to end to end hop time , the per hop time is also important to measure in order to have an accurate prediction of packet loss.
- 3) Application level and network level RTT detect significantly different conditions : for example at application level it reveals problem in receiver and at network level it reveals congestion.

Disadvantages

- 1) Using this experiment for TCP applications sometimes increases the delay because of the inherent connection oriented background of communication.
- 2) Hardware delays sometimes bring inaccuracy in the observations.

Future Works

- 1) Experimental set up can be suggested so as to average the inherent delay in TCP communication so as to increase the accuracy of the setup.
- 2) Set up can be suggested to measure the factors such as the asymmetry in the network.

BIBLIOGRAPHY

UNIX

1. **UNIX NETWORK PROGRAMMING** (VOL 1 2nd EDITION) BY W. RICHARD STEVENS
2. **.LINUX SOCKET PROGRAMMING** BY SEAN WALTON.
3. **ONLINE TUTORIALS** AT <http://www.csc.villanova.edu/>

JAVA

1. **CORE JAVA 2 VOLUME-2** BY CAY S. HORSTMANN AND GARY CORNELL.
2. **COMPLETE REFERENCE JAVA 2** BY HERBERT SCHILDT
3. **ONLINE RESOURCES** AT <http://java.sun.com/docs/books/tutorial/networking/sockets/>
4. **ONLINE TUTORIALS** AT <http://www.csc.vill.edu/~mdamian/Sockets/>

VISUAL BASIC

1. **LEARN VISUAL BASIC 6.0 IN 24 HOURS** : SAMS SERIES
2. **MICROSOFT'S ONLINE MSDN HELP** AT <http://msdn.microsoft.com/library/>
3. **WINSOCK PROGRAMMING TUTORIALS** AT <http://tangentsoft.net/wsfaq/>

PAPERS REFERRED

1. “ *ON STUDYING THE IMPACT OF INTERNET AUDIO TRANSMISSION* “ BY LOPAMUDRA ROYCHOUDHURI , EHAB AL- SHAER , HAZEM HAMED AND GREGORY B. BREWSTER