CERTIFICATE

This is to certify that the project entitled “ECHO SERVER” is a bonafide work of the following students of Delhi College of Engineering

Madan Singh	2K1/COE/O28
Ravi Kumar	2K1/COE/042
Siddharth Sahai Mathur	2KI/COE/055

This project was completed under my direct supervision and guidance and forms a part of their Bachelor of Engineering(B.E.) course curriculum.

They have completed their work with utmost sincerity and diligence.

I wish them all the best for their future endeavors.

(Prof. D.R. Choudhary)
Department of Computer Engineering
Delhi College of Enginering,Delhi.

ACKNOWLEGMENT

We feel honored in expressing our profound sense of gratitude and indebtedness to Prof. D.R. Choudhary Department of Computer Engineering ,Delhi College of Engineering for giving us other opportunity to work on such a practical problem under his expert guidance. He constantly guided and helped us throughout the project. Words cannot express the support and motivation by him

Our sincere thanks to Prof. Goldie Gabrani and all the other faculty members for their cooperation and help during the project.

Madan Singh	Ravi Kumar	Siddharath Sahai Mathur
(2K1/COE/028)	(2K1/COE/042)	(2K1/COE/055)

ABSTRACT

Echo Service is a well known service running at port 7 for testing & debugging purposes. Our project was to develop an Echo Server. Echo server is a simple server which echoes sent by client back to it. Clients for these are also presented ,but with lesser focus. The application of the echo server suggests simplicity ,robustness & efficiency as prime requirements. Due Simplicity of application in case of echo server is synonymous with performance(simplicity reduces overhead) because of the simple nature of the service & the reasons namely testing for which it is used. Hence any extra overhead in the form of a complex algorithm or complex programming techniques like recursion, synchronization is avoided. Though for some special cases different approaches have been presented in brief. These are provided at the end in the section of suggestion for improvement. Four types of servers are there; we covered three(Connectionless-iterative ,Connection -oriented iterative ,connection oriented Concurrent(common))while the fourth Connectionless concurrent is rare. Except the initial servers, rest was designed for robustness/efficiency or both. Each type fulfills different requirements to different degrees & hence, for different versions is developed. Echo service is mostly exploited by Ackers/malicious clients & hence, their vulnerabilities are discussed with possible solutions are provided for those within our scope. The first two servers developed by us are Connection oriented Concurrent TCP server(general Process based) & Connectionless Iterative UDP server(generally UDP servers are iterative).The next Server was a robust multi-mode version of the Concurrent TCP server with input validation, error checks , redundancy, acceptable behavior at boundary conditions, etc).The fourth one is a single process concurrent server to minimize resources based on I/O multiplexing. Lastly, the best option for an echo server ;the threaded server was used to reduce process forking overhead. The platform chosen was UNIX(posix.1g)/Linux owing to the efficiency & simplicity of these systems & the C language was used.

CONTENTS

Ec Network Working Group J. Postel
Request for Comments: 862
ISI
May
1983

Echo Protocol

This RFC specifies a standard for the ARPA Internet community. Hosts on the ARPA Internet that choose to implement an Echo Protocol are expected to adopt and implement this standard.

A very useful debugging and measurement tool is an echo service. An echo service simply sends back to the originating source any data it receives.

TCP Based Echo Service

One echo service is defined as a connection based application on TCP. A server listens for TCP connections on TCP port 7. Once a connection is established any data received is sent back. This continues until the calling user terminates the connection.

UDP Based Echo Service

Another echo service is defined as a datagram based application on UDP. A server listens for UDP datagrams on UDP port 7. When a datagram is received, the data from it is sent back in an answering datagram.

!.2 APPLICATIONS OF ECHO SERVER

The Echo service returns whatever it receives. It is called through port 7. With TCP, it simply returns whatever data comes down the connection, whereas UDP returns an identical datagram (except for the source and destination addresses). The echoes continue until the port connection is broken or no datagrams are received.

The Echo service provides very good diagnostics about the proper functioning of the network and the protocols themselves. The reliability of transmissions can be tested this way, too. Turnaround time from sending to receiving the echo provides useful measurements of response times and latency within the network. Thus, the echo server has the following applications:

1) *Testing the functionality of Transport layer*.

2) *Diagnosis of Networks & protocols.*

3) *Provides visual & easily understandable broad feedback of Latency & response times within the network.*

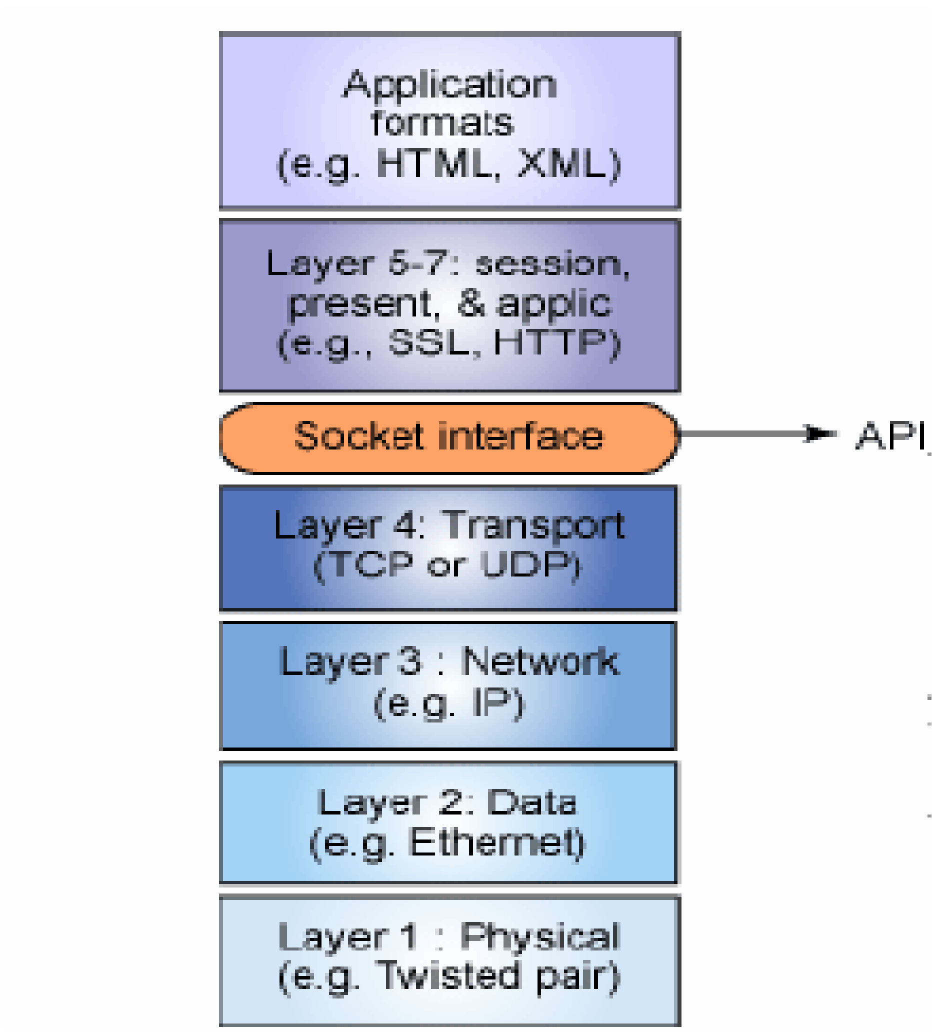
4) *Can be used for testing of an independently developed client Module independent of its corresponding server. By this method The client's responses to the server under various test cases fed To it by the driver can be monitored by sending them to the Client echo application which sends them to echo server which Sends it back to the acceptance testing program on the host.*

2.1 Understanding IP networks and network layers

What is a network?

(**APPLICATION PROGRAMMER'S VIEW**)

What we usually call a computer network is composed of a number of *network layers*,. Each of these network layers provides a different restriction and/or guarantee about the data at that layer. The protocols at each network layer generally have their own packet formats, headers, and layout. The seven traditional layers of a network are divided into two groups: upper layers and lower layers. The sockets interface provides a uniform
API to the lower layers of a network, and allows you to implement upper layers within your sockets application. Further, application data formats may themselves constitute further layers; for example, SOAP is built on top of XML, and ebXML may itself utilize SOAP.


Application formats (e.g. HTML, XML)
Layer 5-7: session, present, & applic (e.g., SSL, HTTP)
Socket interface
API
Layer 4: Transport (TCP or UDP)
Layer 3 : Network (e.g. IP)
Layer 2: Data (e.g. Ethernet)
Layer 1 : Physical (e.g. Twisted pair)

2.2 IP, TCP, and UDP

As the last panel indicated, when you program a sockets application, you have a choice to make between using TCP and using UDP. Each has its own benefits and disadvantages.

TCP is a stream protocol, while UDP is a datagram protocol. In other words, TCP establishes a continuous open connection between a client and a server, over which bytes may be written (and correct order guaranteed) for the life of the connection. However, bytes written over TCP have no built-in structure, so higher-level protocols are required to delimit any data records and fields within the transmitted bytestream.

UDP, on the other hand, does not require a connection to be established between client and server; it simply transmits a message between addresses. A nice feature of UDP is that its packets are self-delimiting; that is, each datagram indicates exactly where it begins and ends A possible disadvantage of UDP, however, is that it provides no guarantee that packets will arrive in order, or even at all. Higher-level protocols built on top of UDP may, of course, provide handshaking and acknowledgments.

A useful analogy for understanding the difference between TCP and UDP is the difference between a telephone call and posted letters. The telephone call is not active until the caller "rings" the receiver and the receiver picks up. The telephone channel remains alive as long as the parties stay on the call, but they are free to say as much or as little as they wish to during the call. All remarks from either party occur in temporal order. On the other hand, when you send a letter, the post office starts delivery without any assurance the recipient exists, nor any strong guarantee about how long delivery will take. The recipient may receive various letters in a different order than they were sent, and the sender may receive mail interspersed in time with those she sends. Unlike with the postal service (ideally, anyway), undeliverable mail always goes to the dead letter office, and is not returned to sender.

2.3 Peers, ports, names, and addresses

Beyond the protocol, TCP or UDP, there are two things a peer (a client or server) needs to know about the machine it communicates with: an IP address and a port. An IP address is a 32-bit data value, usually represented for humans in "dotted quad" notation, such as `64.41.64.172`. A port is a 16-bit data value, usually simply represented as a number less than 65536, most often one in the tens or hundreds range. An IP address gets a packet *to* a machine; a port lets the machine decide which process/service (if any) to direct it to. That is a slight simplification, but the idea is correct. Mostly a name is given instead of a number ;to find the particular host for that name Domain Name Server is queried , but sometimes local lookups are used first (often via the contents of `/etc/hosts`).

3. Client-Server Paradigm

The Client-Server paradigm is the most prevalent model for distributed computing protocols. It is the basis of all distributed computing paradigms at a higher level of abstraction. It is service-oriented, and employs a request-response protocol. A server process, running on a server host, provides access to a service. A client process, running on a client host, accesses the service via the server process. The interaction of the process proceeds according to a protocol.

3.1 Client-Server Design Alternatives

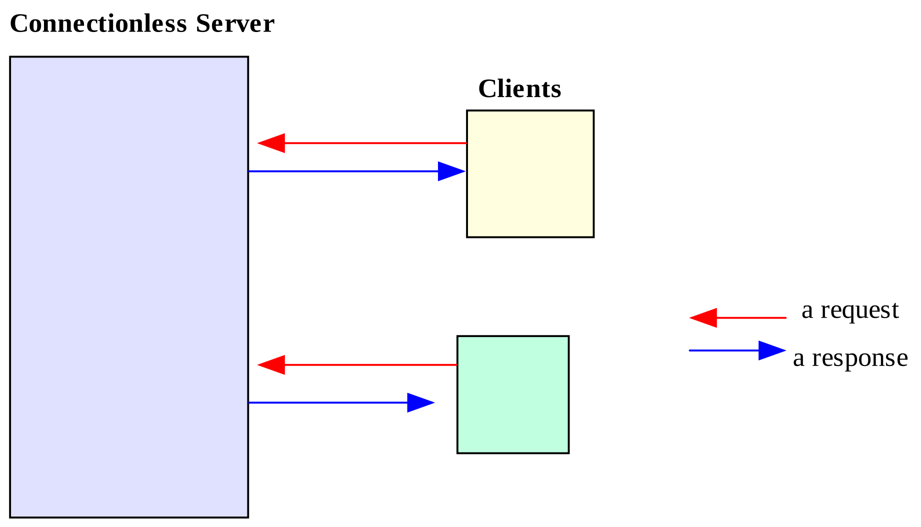
Types of Servers

A server can be:

	Iterative	Concurrent
Connectionless	iterative connectionless	concurrent Connectionless
Connection-Oriented	iterative connection -oriented	concurrent connection-oriented

3.2 Connectionless Server

A connectionless server accepts one request at a time from any client, processes the request, and sends the response to the requestor.

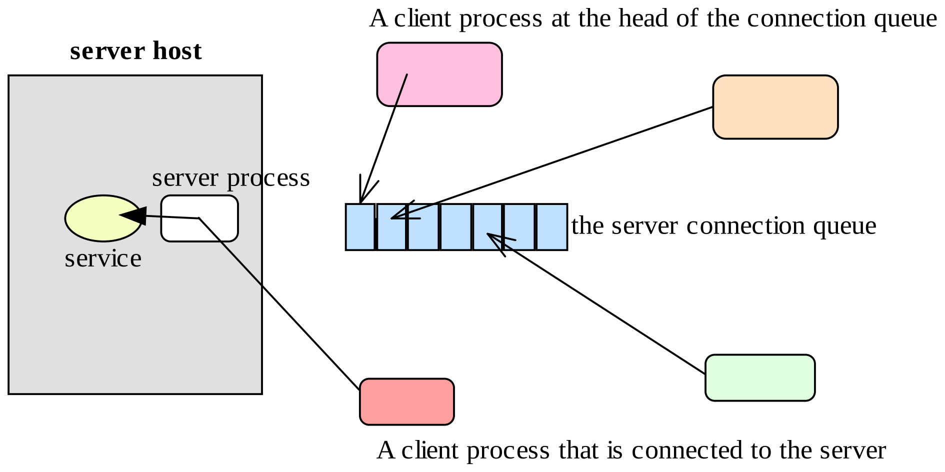


Generally it is iterative.

3.3 Connection-Oriented Client-Server applications

A client-server application can be either *connection-oriented* or *connectionless*.
In a connection-oriented client-server application:

The server is passive: it listens and waits for connection requests from clients, and accepts one connection at a time. A client issues a connection request, and waits for its connection to be accepted. Once a server accepts a connection, it waits for a request from the client.
When a client is connected to the server, it issues a request and waits for the response. When a server receives a request, it processes the request and sends a response, then wait for the next request, if any. The client receives the request and processes it. If there are further requests, the process repeats itself until the protocol is consummated.


A client process at the head of the connection queue
server host
server process
service
the server connection queue
A client process that is connected to the server

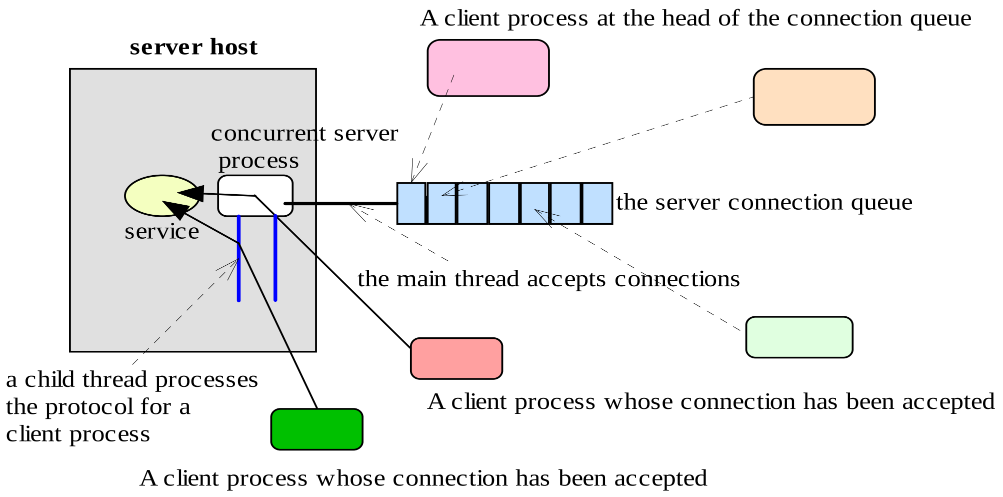
<u>3.4 Iterative (or sequential) Server</u>

An unthreaded connection-oriented server is said to be an *iterative server*. Handles one request at a time Client waits for all previous requests to be processed Unacceptable to user if long request blocks short request

3.5 Concurrent Server

A connection-oriented server can be threaded so that it can service multiple clients concurrently. Such a server is said to be a *concurrent server*. Can handle multiple requests at a time by creating new thread of control to handle each request.

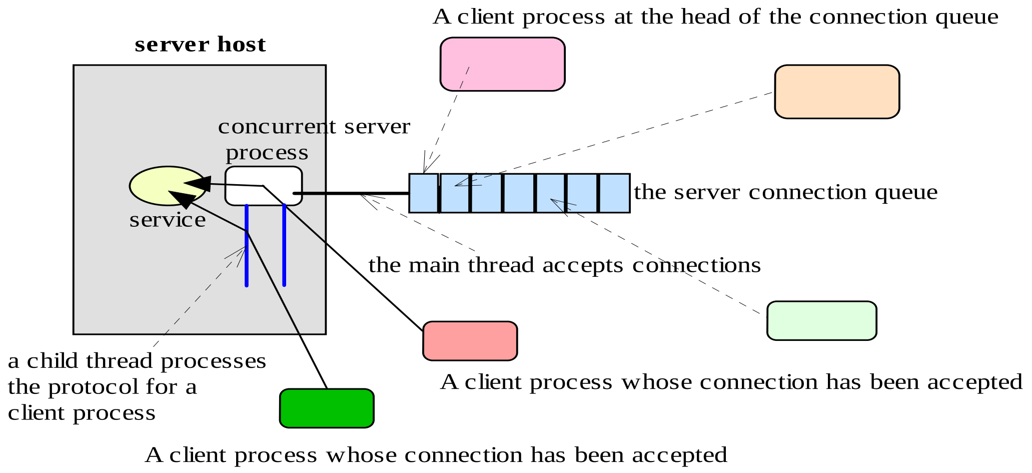
Introducing concurrency into a server arises from the need to provide faster response time to multiple clients. Concurrency improves response time, if forming a response requires significant I/Other processing time varies dramatically among requests, the server executes on a computer with multiple processors. Concurrency may have significant overhead cost associated with it.



A client process at the head of the connection queue
server host
concurrent server process
service
the server connection queue
the main thread accepts connections
a child thread processes the protocol for a client process
A client process whose connection has been accepted
A client process whose connection has been accepted

3.6 Concurrent, Connection-Oriented Server

A connection-oriented server services one client at a time.
If the duration of each client session is significant, then the latency or turnaround time of a client request becomes unacceptable if the number of concurrent client processes is large. To improve the latency, a server process spawns a child process or child thread to process the protocol for each client. Such a server is termed a ***concurrent*** server, compared to an ***iterative*** server. A concurrent server uses its main thread to accept connections, and spawns a child thread to process the protocol for each client. Clients queue for connection, then are served concurrently. The concurrency reduces latency significantly



A client process at the head of the connection queue
server host
concurrent server process
service
the server connection queue
the main thread accepts connections
a child thread processes the protocol for a client process
A client process whose connection has been accepted
A client process whose connection has been accepted

Connection-oriented servers use a *connection* as the basic paradigm for communication. They allow a client to establish a connection to a server, communicate over that connection, and then discard it after finishing.
In most cases, the connection between clients and a server handles more than a single request, thus:
Connection-oriented protocols implement concurrency among Connections rather than individual requests.

3.7 Iterative connection oriented

Here the server processes one connection after another &only a single process is used to serve clients

4 **SOCKET INTERFACE**

4.1 What do sockets do?

While the sockets interface theoretically allows access to *protocol families* other than IP, in practice, every network layer you use in your sockets application will use IP. For this tutorial we only look at IPv4; in the future IPv6 will become important also, but the principles are the same. At the transport layer, sockets support two specific protocols: TCP (transmission control protocol) and UDP (user datagram protocol).
Sockets cannot be used to access lower (or higher) network layers; for example, a socket application does not know whether it is running over Ethernet, token ring, or a dial-up connection. Nor does the socket's pseudo-layer know anything about higher-level protocols like NFS, HTTP, FTP, and the like (except in the sense that you might yourself write a sockets application that implements those higher-level protocols).
At times, the sockets interface is not your best choice for a network programming API.
Specifically, many excellent libraries exist (in various languages) to use higher-level protocols directly, without your having to worry about the details of sockets; the libraries handle those details for you. While there is nothing wrong with writing you own SSH client, for example, there is no need to do so simply to let an application transfer data securely. Lower-level layers than those addressed by sockets fall pretty much in the domain of device driver programming.

A socket is formally defined as an endpoint for communication between an application program, and the underlying network protocols. This odd collection of words simply means that the program reads information from a socket in order to read from the network, writes information to it in order to write to the network, and sets sockets options in order to control protocol options. From the programmer's point of view, the socket is identical to the network. Just like a file descriptor is the endpoint of disk operations.

In this section we begin with socket address structure; this structure can be passes in two directions:

1. From process to kernel.
2. and from kernel to process

We have used address conversation function. One problem with address conversation function is that they are dependant on the types of address being converted: IPv4 or IPv6

A socket server needs to be able to listen on a specific port, accept connections and read and write data from the socket. A high performance and scaleable socket server should use asynchronous socket IO and IO completion ports. Since we're using IO completion ports we need to maintain a pool of threads to service the IO completion packets

Types of sockets

In general, 3 types of sockets exist on most UNIX systems:

1. Stream sockets,

2. Datagram sockets and

3. Raw sockets.

Stream sockets are used for stream connections, i.e. connections that exist for a long duration. TCP connections use stream sockets.

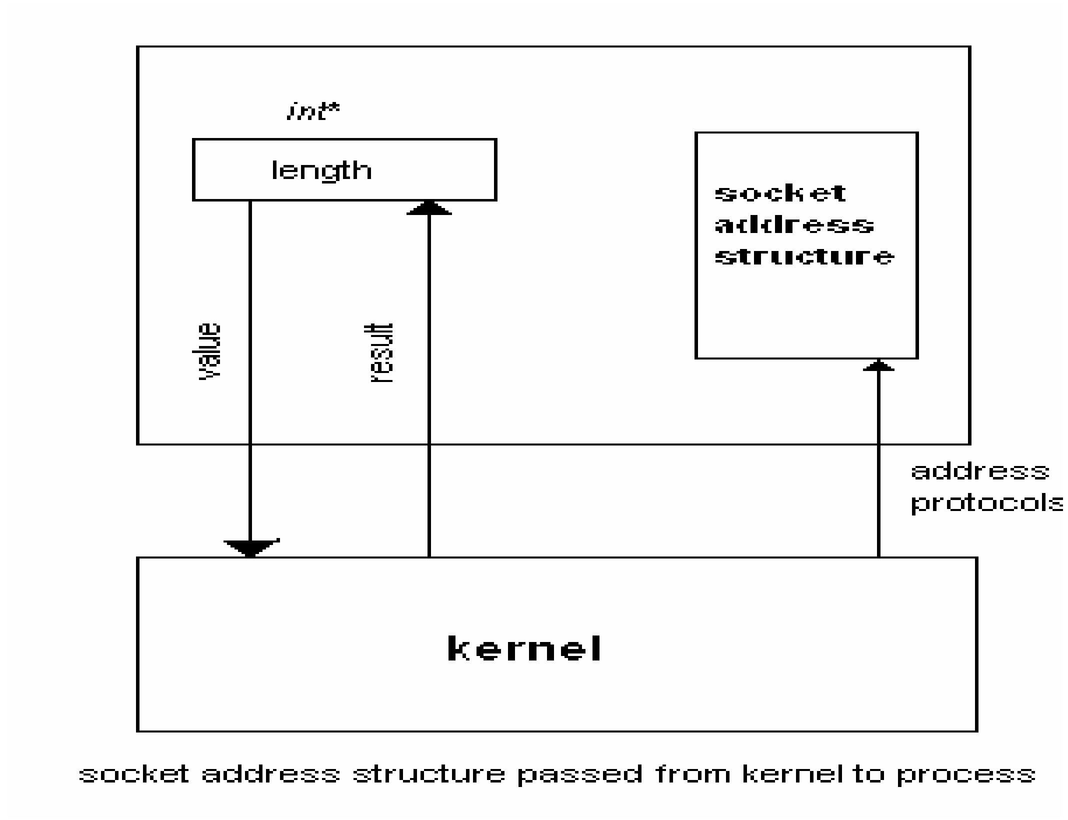
Datagram sockets are used for short-term connections that transfer a single packet across the network before terminating. The UDP protocol uses such sockets, due to its connection-less nature.

Raw sockets are used to access low-level protocols directly, bypassing the higher protocols. They are the means for a programmer to use the IP protocol, or the physical layer of the network, directly. Raw sockets can therefore be used to implement new protocols on top of the low-level protocols. Naturally, they are out of our scope.

4.2 Concept of Value results arguments:

When a socket address structure is passed to any socket function it is always passed by reference that is a pointer to structure is passed. The length of structure is also passed as an argument. But the way in which the length is passed depends on which direction the structure is being passed.

1. Three functions , *bind* , *connect* , and *sendto* , passes a socket address structure from the process of kernel. One argument of these three functions is the pointer to the socket address structure and another argument is the integer size of the structure.
2. Four functions *accept, recvfrom, getsockname,* and *getpeername,* pass a socket address structure from kernel to the process, the reverse direction from previous scenario. Two of the arguments to these four functions are the pointer to socket address structure along with a pointer to an integer containing the size of the structure



socket address structure passed from kernel to process

4.3 SOCKET API

4.3.1Creating sockets

Creation of sockets is done using the `socket()` system call. This system call is defined as follows:

```
  int socket(int address_family, int socket_type,
int proto_family);
```

As an example, suppose that we want to write a TCP application. This application needs at least one socket in order to communicate across the Internet, so it will contain a call such as this:

```
  int s;    /* descriptor of socket */

  /* Internet address family, Stream socket */
```

```
s = socket(AF_INET, SOCK_STREAM, 0);
if (s < 0) {
    perror("socket: allocation failed");
}
```

4.3.2 Associating a socket with a connection

After a socket is created, it still needs to be told between which two end points it will communicate. It needs to be bound to a connection. There are two steps to this binding. The first is binding the socket to a local address. The second is binding it to a remote (foreign) address.

Binding to a local address could be done either explicitly, using the `bind()` system call, or implicitly, when a connecting is established. Binding to the remote address is done only when a connection is established. To bind a socket to a local address, we use the `bind()` system call, which is defined as follows:

```
   int bind(int socket, struct sockaddr *address,
int addrlen);
```

Note the usage of a different type of structure, namely `struct sockaddr`, than the one we used earlier (`struct sockaddr_in`). Why is the sudden change? This is due to the generality of the socket interface: sockets could be used as endpoints for connections using different types of address families. Each address family needs different information, so they use different structures to form their addresses. Therefore, a generic socket address type, struct sockaddr, is defined in the system, and for each address family, a different variation of this structure is used. For those who know, this means that `struct sockaddr_in`, for example, is an overlay of struct sockaddr (i.e. it uses the same memory space, just divides it differently into fields).

the bind assign a local protocol aaddress to asocket with ainternet protocol
the protocol address is the combination of either a 32 bit IP v4 address or
128 bit IPv6 address along with 16 bit TCP or UDP port no.

#include<sys/socket.h>

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

retuerns o if ok -1 on error

the second argument is a pointer a protocol specific address and the third argument is the size of address structure. With TCP calling bind let us specify a port no. an IP address , both or neither.

There are 4 possible variations of address binding that might be used when binding a socket in the Internet address family.

The first is binding the socket to a specific address, i.e. a specific IP number and a specific port. This is done when we know exactly where we want to receive messages. Actually this form is not used in simple servers, since usually these servers wish to accept connections to the machine, no matter which IP interface it came from.

The second form is binding the socket to a specific IP number, but letting the system choose an unused port number. This could be done when we don't need to use a well-known port.

The third form is binding the socket to a wild-card address called INADDR_ANY (by assigning it to the `sockaddr_in` variable), and to a specific port number. This is used in servers that are supposed to accept packets sent to this port on the local host, regardless of through which physical network interface the packet has arrived (remember that a host might have more than one IP address).

The last form is letting the system bind the socket to any local IP address and to pick a port number by itself. This is done by not using the `bind()` system call on the socket. The system will make the local bind when a connection through the socket is established, i.e. along with the remote address binding. This form of binding is usually used by clients, which care only about the remote address (where they connect to) and don't need any specific local port or local IP address. However, there are exceptions here too.

4.3.3 Sending and receiving data over a socket

After a connection is established (We will explain that when talking about Client and Server writing), There are several ways to send information over the socket. We will only describe one method for reading and one for writing.

The `read()` system call

The most common way of reading data from a socket is using the `read()` system call, which is defined like this:

```
int read(int socket, char *buffer, int buflen);
```

- socket - The socket from which we want to read.
- buffer - The buffer into which the system will write the data bytes.
- buflen - Size of the buffer, in bytes (actually, how much data we want to read).

The read system call returns one of the following values:

- 0 - The connection was closed by the remote host.
- -1 - The read system call was interrupted, or failed for some reason.
- n - The read system call put 'n' bytes into the buffer we supplied it with.

The `write()` system call

The most common way of writing data to a socket is using the `write()` system call, which is defined like this:

```
int write(int socket, char *buffer, int buflen);
```

- socket - The socket into which we want to write.
- buffer - The buffer from which the system will read the data bytes.
- buflen - Size of the buffer, in bytes (actually, how much data we want to write).

The write system call returns one of the following values:

- 0 - The connection was closed by the remote host.
- -1 - The write system call was interrupted, or failed for some reason.
- n - The write system call wrote 'n' bytes into the socket.

4.3.4 Closing a socket.

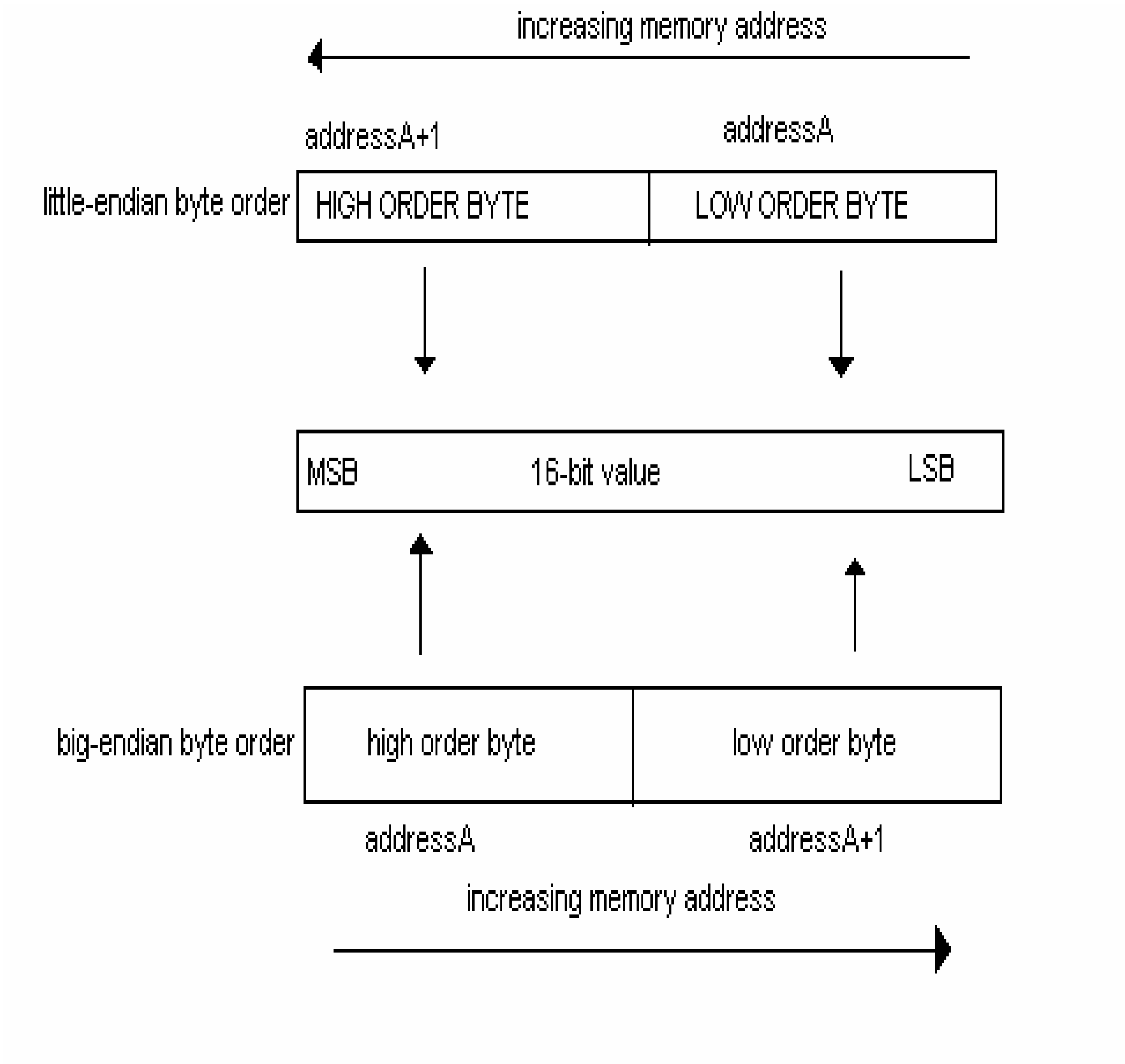
When we want to abort a connection, or to close a socket that is no longer needed, we can use the `close()` system call. it is defined simply as:

```
int close(int socket);
```

- socket - The socket that we wish to close. If it is associated with an open connection, the connection will be closed.

4.3.5 BYTE ORDENING FUNCTIONS

There are two ways to store the two bytes in memory: with the low order byte at the starting address known as *litlte- endian* byte order, or with the high-order byte at the starting address, known as big-endian byte order.



***inet_aton_ inet_addr*, and *inet _ntoa* FUNTIONS**

1.*inet_aton*, *inet_ntoa*, and *inet_addr* convert an IPv4 address from a dotted-decimal string (e.g., "206 .168 .112 . 96") to its 32-bit network byte ordered binary value.
2. the newer functions, inet_pton and innet_ntop, handle both IPv4 andIPv6 addresses.
#include<inet.h>
*int inet_aton(const char *strptr , stuct in_addr *addrptr);*

returns: 1 if string was valid, 0 on error
*in_addr_t inet_addr(const char *strpt);*
returns: 32-bit binary network byte ordered IPv4 address ;inaddr_none if error
*char *inet_ntoa(struct in_addr inaddr)*
returns:pointer to dotted-decimal string

EXPLAINATION: *inet_aton*, converts the c character string pointed to by *strptr* into its 32-bit binary network Byte ordered value , which is stored through the pointer *addrptr*. if successful, 1 is returned; other wise, 0 is returned.
Inet_addr does the same conversion, returning the 32-bit binary network byte ordered value as the return value.
The *inet_ntoa* function converts a 32-bit binary network byte ordered IPv4 address in to its corresponding dotted decimal string.

Inet_pton and *inet_ntop* function:

These two functions are new with IPv6 and work with both IPv4 IPv6.
#include<arpa/inet.h>
*int inet_poton(int family, const char *strptr, void *addrptr);*
return:1 if ok; input not valid presentation format., -1on error
*const char *inet_ntop(int family , const void *addrptr ,char *strptr,size_t len);*
returns:pointer to result if ok, null on error

Explanation:

the *inet_pton* function tries to tries to convert the string pointed to by *strptr*. Storing the binary result through the pointer *addrptr* . If successful , the return value is 1,if 0 is returned then not valid.
Inet_ntop does the reverse conversion ,from numeric to presentation.

Sock_ntop and related functions:

A basic problem with *inet_ntop* is that it requires the caller to pass a pointer to a binary address. this address is normally contain in a socket address

structure, repairing the caller to know the format of the structure and the address family .That is, to use it, we must write code of the form
Struct sockaddr_in addr;
Inet_ntop(AF_INET, &addr.sin_addr,str,sizeof(str));
For IPv4, or
Struct sockaddr_in6 addr6;
Inet_ntop(AF_INET6, &addr6.sin6_addr, str, sizeof(str));
For IPv6

4.3.6 **Preparing process for I/O**

To perform I/O, the first thing a process must do is call the socket function, specifying the type of communication protocol desired (TCP IPv4,UDP using IPv6,unix domain stream protocol)
The family specifies the protocol family and is one of the constants. The socket type is one of the constants . normally the protocol argument to the socket function is set to 0 except for raw circuit.

Family	Description
AF_INET	IPV4 PROTOCOLS
AF_INET6	IPV6 PROTOCOLS
AF_LOCAL	UNIX DOMAIN PROTOCOLS
AF_ROUTE	ROUTING SOCKETS
AF_KEY	KEY SOCKETS

1)Connect function

The connect function is used by tcp client to establish a connection with a tcp server

#include<sys/socket.h>

int connect (int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
return 0 if ok ,-1 on error

sockfd is a socket descriptor that was returned by the socket function.
The second and third arguments are pointer to socket address structure and its size.
The socket address structure must contain the IP address and port no. of the server
The client does not have to call bind before calling connect
TCP socket ,the connect function initiates TCP three way handshake . the function retunes only when the connection is established or an error occurs.

2)*bind*: As already described in a previous section.

3)listen function

listen function is called only by the TCP server and it performs two actions
1. when asocket is created by socket function, it is assumed to be an active socket, that is ,a client socket that will issue a connect.
The listen function converts an unconnected socket into a passive socket, indicating that a kernel should accept an incoming connection requests direct to the socket.

2.the second argument to these function specifies the maximum no. of connection that kernel should queue for the socket

```
#include<sys/socket.h>
int listen( int sockfd, int backlog);
```

returns 0 if ok ,-1 on error
these function is normally called after both the socket and the bind function and must be called before calling the accept function
accept function

4)accept function is called by TCP server to return the next complete connection from the front of the completed connection queue.

If the completed connection queue is empty the process is put to sleep

#include<sys/socket.h>

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

returns negative descriptor if ok ,-1 on error

the cliaddr and addrlen argument are used to return the protocol address of connected peer process. Addrlen is a value-result argument before the call, we set the integer pointed by *addrlen to the size of socket address pointed to by cliaddr and on return these integer value contains the actual no. of bytes stored by kernel in the socket address structure.

5)*fork and exec* functions
this function is only way in UNIX to create a new process.

#include<unistd.h>
pid_t fork(void);

fork is that it is called once but it returns twice. It returns in calling process with a return that is the process ID of the newly created.
It also return s once in child, with return value of 0 hence the return value tells the process whether it is parent or child.

5. A Concurrent TCP echo server

5.1 ECHO SERVER

A socket server is more complicated than a client, mostly because a server usually needs to be able to handle multiple client requests. Basically, there are two aspects to a server: handling each established connection, and listening for connections to establish.
In our example, and in most cases, you can split the handling of a particular connection into support function, which looks quite a bit like how a TCP client application does. We name that function `HandleClient()`.
Listening for new connections is a bit different from client code. The trick is that the socket you initially create and bind to an address and port is not the actually connected socket. This initial socket acts more like a socket factory, producing new connected sockets as needed. This arrangement has an advantage in enabling fork'd, threaded, or asynchronously dispatched handlers (using `select()`);Our echo server starts out with pretty much the same few `#include`s as the client did, and defines some constants and an error-handling function:
The `BUFFSIZE` constant limits the data sent per loop. The `MAXPENDING` constant limits the number of connections that will be queued at a time (only one will be *serviced* at a time in our simple server). The `Die()` function is the same as in our client.

5.1.1 Application setup

```
#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
```

```
#define MAXPENDING 5 /* Max connection requests */
#define BUFFSIZE 32
void Die(char *mess) { perror(mess); exit(1); }


void HandleClient(int sock) {
char buffer[BUFFSIZE];
int received = -1;
/* Receive message */
if ((received = recv(sock, buffer, BUFFSIZE, 0)) < 0) {
Die("Failed to receive initial bytes from client");
}
/* Send bytes and check for more incoming data in loop */
while (received > 0) {
/* Send back received data */
if (send(sock, buffer, received, 0) != received) {
Die("Failed to send bytes to client");
}
/* Check for more data */
if ((received = recv(sock, buffer, BUFFSIZE, 0)) < 0) {
Die("Failed to receive additional bytes from client");
}
}
close(sock);
}

int main(int argc, char *argv[]) {
int serversock, clientsock;
struct sockaddr_in echoserver, echoclient;
if (argc != 2) {
fprintf(stderr, "USAGE: echoserver <port>\n");
exit(1);
}
/* Create the TCP socket */
if ((serversock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) <
0) {
Die("Failed to create socket");
}
/* Construct the server sockaddr_in structure */
memset(&echoserver, 0, sizeof(echoserver)); /* Clear struct */
echoserver.sin_family = AF_INET; /* Internet/IP */
echoserver.sin_addr.s_addr = htonl(INADDR_ANY); /* Incoming addr */
echoserver.sin_port = htons(atoi(argv[1])); /* server port */

/* Bind the server socket */
if (bind(serversock, (struct sockaddr *) &echoserver,
sizeof(echoserver)) < 0) {
Die("Failed to bind the server socket");
}
/* Listen on the server socket */
if (listen(serversock, MAXPENDING) < 0) {
Die("Failed to listen on server socket");
}

/* Run until cancelled */
while (1) {
unsigned int clientlen = sizeof(echoclient);
/* Wait for client connection */
if ((clientsock =
accept(serversock, (struct sockaddr *) &echoclient,
&clientlen)) < 0) {
Die("Failed to accept client connection");
}
fprintf(stdout, "Client connected: %s\n",
inet_ntoa(echoclient.sin_addr));
HandleClient(clientsock);

}
}
```

5.1.2 Connection handler

The handler for echo connections is straightforward. All it does is receive any initial bytes
available, then cycles through sending back data and receiving more data. For short echo strings (particularly if less than `BUFFSIZE`) and typical connections, only one pass through the `while` loop will occur. But the underlying sockets interface (and TCP/IP) does not make any guarantees about how the bytestream will be split between

calls to `recv()`.The socket that is passed in to the handler function is one that already connected to the requesting client. Once we are done with echoing all the data, we should close this socket; the parent server socket stays around to spawn new children, like the one just closed.

5.1.3 Configuring the server socket

As outlined before, creating a socket has a bit different purpose for a server than for a client. Creating the socket has the same syntax it did in the client, but the structure `echo server` is set up with information about the server itself, rather than about the peer it wants to connect to. You usually want to use the special constant `INADDR_ANY` to enable receipt of client requests on any IP address the server supplies; in principle, such as in a multi-hosting server, you could specify a particular IP address instead.

Notice that both IP address and port are converted to network byte order for the `sockaddr_in` structure. The reverse functions to return to native byte order are `ntohs()` and `ntohl()`. These functions are no-ops on some platforms, but it is still wise to use them for cross-platform compatibility.

5.1.4 Binding and Listening

Whereas the client application `connect()`'d to a server's IP address and port, the server `bind()`s to its own address and port: Once the server socket is bound, it is ready to `listen()`. As with most socket functions, both `bind()` and `listen()` return -1 if they have a problem. Once a server socket is listening, it is ready to `accept()` client connections, acting as a factory for sockets on each connection.

5.1.5 **Socket factory**

Creating new sockets for client connections is the crux of a server. The function `accept()` does two important things: it returns a socket pointer for the new socket; and it populates the `sockaddr_in` structure pointed to, in our case, by `echoclient`. We can see the populated structure in `echoclient` with the `fprintf()` call that accesses the client IP address. The client socket pointer is passed to `HandleClient()`.

5.2 ECHO CLIENT

5.2.1 A TCP echo client (client setup)

```
#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#define BUFFSIZE 32
void Die(char *mess) { perror(mess); exit(1); }
```

A particular buffer size is allocated, which limits the amount of data echoed at each pass (but we loop through multiple passes, if needed). A small error function is also defined.

5.2.2 Creating the socket

The arguments to the `socket()` call decide the type of socket: `PF_INET` just means it uses IP (which you always will); `SOCK_STREAM` and `IPPROTO_TCP` go together for a TCP socket.

```
int main(int argc, char *argv[]) {
int sock;
struct sockaddr_in echoserver;
char buffer[BUFFSIZE];
unsigned int echolen;
```

```
int received = 0;
if (argc != 4) {
fprintf(stderr, "USAGE: TCPecho <server_ip> <word>
<port>\n");
exit(1);
}
/* Create the TCP socket */
if ((sock = socket(PF_INET, SOCK_STREAM,
IPPROTO_TCP)) < 0) {
Die("Failed to create socket");
}
```

The value returned is a socket handle, which is similar to a file handle; specifically, if the
socket creation fails, it will return -1 rather than a positive-numbered handle.

5.2.3 Establish connection

Now that we have created a socket handle, we need to establish a connection with the server.
A connection requires a `sockaddr` structure that describes the server. Specifically, we need to
specify the server and port to connect to using
`echoserver.sin_addr.s_addr` and
`echoserver.sin_port`. The fact that we are using an IP address is specified with
`echoserver.sin_family`, but this will always be set to `AF_INET`.

```
/* Construct the server sockaddr_in structure */
memset(&echoserver, 0, sizeof(echoserver)); /*
Clear struct */
echoserver.sin_family = AF_INET; /* Internet/IP */
echoserver.sin_addr.s_addr = inet_addr(argv[1]); /*
IP address */
echoserver.sin_port = htons(atoi(argv[3])); /*
server port */
/* Establish connection */
if (connect(sock,
(struct sockaddr *) &echoserver,
sizeof(echoserver)) < 0) {
Die("Failed to connect with server");
}
```

As with creating the socket, the attempt to establish a connection will return -1 if the attempt
fails. Otherwise, the socket is now ready to accept sending and receiving data.

5.2.4 Send/Receive data

Now that the connection is established, we are ready to send and receive data. A call to
`send()` takes as arguments the socket handle itself, the string to send, the length of the sent
string (for verification), and a flag argument. Normally the flag is the default value 0. The return
value of the `send()` call is the number of bytes successfully sent.

```
/* Send the word to the server */
echolen = strlen(argv[2]);
if (send(sock, argv[2], echolen, 0) != echolen) {
Die("Mismatch in number of sent bytes");
}
/* Receive the word back from the server */
fprintf(stdout, "Received: ");
while (received < echolen) {
int bytes = 0;
if ((bytes = recv(sock, buffer, BUFFSIZE-1, 0)) <
1) {
Die("Failed to receive bytes from server");
}
received += bytes;
buffer[bytes] = '\0'; /* Assure null terminated
string */
fprintf(stdout, buffer);
}
```

The `rcv()` call is not guaranteed to get everything in-transit on a particular call; it simply blocks until it gets *something*. Therefore, we loop until we have gotten back as many bytes as were sent, writing each partial string as we get it. Obviously, a different protocol might decide when to terminate receiving bytes in a different manner (perhaps a delimiter within the bytestream).

5.2.5 Security Issues with TCP Ports and Services

TCP attacks differ in that TCP is not a stateless protocol and requires a TWHS (three way hand shake) before initiating service. This does not make TCP ports immune to DoS attacks. In fact the TWHS is itself a major target of cr/hacker DoS attack attempts.

A SYN-Flood and the ACK-Flood DoS takes advantage of the TWHS to perform a DoS on a host. The normal process of SYN followed by RST or ACK is interrupted and the victim is left with an open port awaiting communication that never materializes. The process is repeated until the total number of simultaneous sessions is open (in theory 1024) and the system is hung. "... in order to completely deny services to a given port on your computer until the next system reboot, the attacker need only send 1024 packets to your computer with the SYN bit set ... One second of packets results in a system reboot - that's a big advantage for the attacker ... (but) ... many systems run out of internal space to store the incomplete connections before the second passes and crash on their own."

The SYN-Flood can easily be turned into a DDoS by using distributed hosts to bounce off packets so that the forensic log examination points to these hosts. A compromised web server can also be used so that infected systems participate after visiting the site, and so continue the attack.

Other TCP attacks include attacks against TCP services such as TELNET using combinations of TCP and ICMP forgery to create a "man-in-the-middle" situation that allows a cr/hacker to see (and route) TELNET packets.

Versions of UDP attacks also exist for TCP but are more difficult to initiate (because of the TWHS). However, once initiated, they can be very effective. For example, "on a TCP connection ... (to ‘chargen' TCP port#19) ... it will spit out a continual stream of garbage characters until the connection is closed. As we know the syn packet is used in the 3-way handshake. The syn flooding attack is based on an incomplete handshake. That is the attacker host will send a flood of syn packet but will not respond with an ACK packet. The TCP/IP stack will wait a certain amount of time before dropping the connection, a syn flooding attack will therefore keep the syn_received connection queue of the target machine filled.

5.3 An ITERATIVE UDP echo server in C

5.3.1Server setup

Even more than with TCP applications, UDP clients and servers are quite similar to each other.
In essence, each consists mainly of some `sendto()` and `recvfrom()` calls mixed together.

The main difference for a server is simply that it usually puts its main body in an indefinite loop to keep serving. Let's start out with the usual includes and error function: Declarations and usage message Again, not much is new in the UDP echo server's declarations and usage message. We need a socket structure for the server and client, a few variables that will be used to verify transmission sizes, and, of course, the buffer to read and write the message.

```
int main(int argc, char *argv[]) {
int sock;
struct sockaddr_in echoserver;
struct sockaddr_in echoclient;
char buffer[BUFFSIZE];
unsigned int echolen, clientlen, serverlen;
int received = 0;
if (argc != 2) {
fprintf(stderr, "USAGE: %s <port>\n", argv[0]);
exit(1);
}
```

5.3.2 Create, configure, and bind the server socket

The first real difference between UDP client and server comes in the need to bind the socket on the server side. We saw this already with the Python example, and the situation is the same here. The server socket is not the actual socket the message is transmitted over; rather, it acts as a factory for an *ad hoc* socket that is configured in the `recvfrom()` call we will see soon.

```
/* Create the UDP socket */
if ((sock = socket(PF_INET, SOCK_DGRAM,
IPPROTO_UDP)) < 0) {
Die("Failed to create socket");
}
```

```
/* Construct the server sockaddr_in structure */
memset(&echoserver, 0, sizeof(echoserver)); /*
Clear struct */
echoserver.sin_family = AF_INET; /* Internet/IP */
echoserver.sin_addr.s_addr = htonl(INADDR_ANY); /*
Any IP address */
echoserver.sin_port = htons(atoi(argv[1])); /*
server port */
/* Bind the socket */
serverlen = sizeof(echoserver);
if (bind(sock, (struct sockaddr *) &echoserver,
serverlen) < 0) {
Die("Failed to bind server socket");
}
```

The `echoserver` structure is configured a bit differently. In order to allow connection on any IP address the server hosts, we use the special constant
`INADDR_ANY` for the member `.s_addr`.

5.3.3 The receive/send loop

The heavy lifting -- such as it is -- in the UDP sever is its main loop. Basically, we perpetually
wait to receive a message in a `recvfrom()` call. When this happens, the `echoclient`
structure is populated with relevant members for the connecting socket. We then use that
structure in the subsequent `sendto()` call.

```
/* Run until cancelled */
while (1) {
/* Receive a message from the client */
clientlen = sizeof(echoclient);
if ((received = recvfrom(sock, buffer, BUFFSIZE, 0,
(struct sockaddr *) &echoclient,
&clientlen)) < 0) {
Die("Failed to receive message");
}
fprintf(stderr,
"Client connected: %s\n",
inet_ntoa(echoclient.sin_addr));


  if(htonl(cliaddr.sin_addr)==inet_addr(“127.0.0.1”
  ))
  {printf(“rogue client:shutting down client”);
  exit(1);}

/* Send the message back to client */

if (sendto(sock, buffer, received, 0,
(struct sockaddr *) &echoclient,
sizeof(echoclient)) != received) {
Die("Mismatch in number of echo'd bytes");
}
}
}
```

This arrangement does only one thing at a time, which might be a problem for a server handling many clients (probably not for this simple echo server, but something more complicated might introduce poor latencies).

5.3.4 A UDP echo client in C

Client setup

The first few lines of our UDP client are identical to those for the TCP client. Mostly we just use some includes for socket functions, or other basic I/O functions.

```
#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#define BUFFSIZE 255
void Die(char *mess) { perror(mess); exit(1); }
```

There is not too much to the setup. It is worth noticing that the buffer size we allocate is much larger than it was in the TCP version (but still finite in size). TCP can loop through the pending data, sending a bit more over an open socket on each loop. For this UDP version, we want a buffer that is large enough to hold the entire message, which we send in a single datagram (it can be smaller than 255, but not any larger). A small error function is also defined.

Declarations and usage message

At the very start of the `main()` function, we allocate two `sockaddr_in` structures, a few integers to hold string sizes, another int for the socket handle, and a buffer to hold the returned string. After that, we check that the command-line arguments look mostly correct.

```
int main(int argc, char *argv[]) {
int sock;
struct sockaddr_in echoserver;
struct sockaddr_in echoclient;
char buffer[BUFFSIZE];
unsigned int echolen, clientlen;
int received = 0;
if (argc != 4) {
fprintf(stderr, "USAGE: %s <server_ip> <word>
<port>\n", argv[0]);
exit(1);
}
```

For this C client, you *must* use a dotted-quad IP address. If you wanted to do a lookup in the C client, you would need to program a DNS function.In fact, it would not be a terrible idea to check that the IP address passed in as the server IP address really looks like a dotted-quad address. If you forgetfully pass in a named address, you will probably receive the somewhat misleading error: "Mismatch in number of sent bytes: No route to host". Any named address amounts to the same thing as an unused or reserved IP address (which a simple pattern check could not rule out, of course).

Create the socket and configure the server structure

The arguments to the `socket()` call decide the type of socket: `PF_INET` just means it uses IP (which you always will); `SOCK_DGRAM` and `IPPROTO_UDP` go together for a UDP socket. In preparation for sending the message to echo, we populate the intended server's structure using command-line arguments.

```
/* Create the UDP socket */
if ((sock = socket(PF_INET, SOCK_DGRAM,
IPPROTO_UDP)) < 0) {
Die("Failed to create socket");
}
/* Construct the server sockaddr_in structure */
memset(&echoserver, 0, sizeof(echoserver)); /*
Clear struct */
echoserver.sin_family = AF_INET; /* Internet/IP */
echoserver.sin_addr.s_addr = inet_addr(argv[1]); /*
IP address */
echoserver.sin_port = htons(atoi(argv[3])); /*
server port */
```

The value returned in the call to `socket()` is a socket handle and is similar to a file handle; specifically, if the socket creation fails, it will return -1 rather than a positive numbered handle. Support functions `inet_addr()` and `htons()` (and `atoi()`) are used to convert the string arguments into appropriate data structures.

Send the message to the server
.

```
/* Send the word to the server */
echolen = strlen(argv[2]);
if (sendto(sock, argv[2], echolen, 0,
(struct sockaddr *) &echoserver,
sizeof(echoserver)) != echolen) {
Die("Mismatch in number of sent bytes");
}
```

The error checking in this call usually establishes that a route to the server exists. This is the message raised if a named address is used by mistake, but it also occurs for valid-looking but unreachable IP addresses.
Receive the message back from the server
Receiving the data back works pretty much the same way as it did in the TCP echo client. The only real change is a substitute call to `recvfrom()` for the TCP call to `recv()`.

```
/* Receive the word back from the server */
fprintf(stdout, "Received: ");
clientlen = sizeof(echoclient);
if ((received = recvfrom(sock, buffer, BUFFSIZE, 0,
(struct sockaddr *) &echoclient,
&clientlen)) != echolen) {
Die("Mismatch in number of received bytes");
}
/* Check that client and server are using same
socket */
if (echoserver.sin_addr.s_addr !=
echoclient.sin_addr.s_addr) {
Die("Received a packet from an unexpected server");
}
buffer[received] = '\0'; /* Assure null-terminated
string */
fprintf(stdout, buffer);
fprintf(stdout, "\n");
close(sock);
exit(0);
}
```

The structure `echoserver` had been configured with an *ad hoc* port during the call to
`sendto()`; in turn, the `echoclient` structure gets similarly filled in with the call to

`recvfrom()`. This lets us compare the two addresses -- if some other server or port sends a datagram while we are waiting to receive the echo. We guard at least minimally against stray datagrams that do not interest us (we might have checked the `.sin_port` members also, to be completely certain).At the end of the process, we print out the datagram that came back, and close the socket.

5.3.5 Security Issues with UDP Ports and Services

Any port can be attacked as a DoS by simply sending a packet to that port. If there is no service attached to that port, then the packet is ignored and the DoS attack fails. If there is a service attached to that port, then the service must deal with the packet, even if it is malformed or incorrect. The service will deal with the incoming packet as a high priority (interrupt) event. The success of the DoS attack is dependent on how effectively the service deals with the inbound packet.

As a rule, any UDP port that sends a response to a packet is subject to a DoS attack (and therefore to a DDoS attack). Since the UDP service is a stateless response, it can simply be flooded with packets, forcing a DoS as the system struggles to keep up with these high priority service interrupts.

Echo - UDP port #7 is a typical example of a DoS and DDoS attack point. "UDP Port #7 is normally the echo service. The function of this service is to transmit whatever data was sent to it back to the source. The echo port is typically available as a service since many networks (and firewalls) use echo response for system management and performance monitoring. As well, "the Harvest Web server sometimes used port #7 to determine whether or not to update a cached web file. This means that any server that provides Web caching has to make UDP port #7 available for this service to work properly.

A simple attack would be to forge a packet from system A, port #7 to system B, port#7. B would process the packet and send it back to A, who would return it to B. The two machines would engage in a high priority packet passing 'ping pong' game, using resources normally assigned to user processing. This can also serve to " ... dominate lower speed communications media, denying communications. But, if we want to be more certain of this, we might add something else to the packet. For example, if we set the 'type of service' field to 'Network Control, Low delay, High Throughput, High Reliability' by setting the value to all 1's, we will force these packets to override other packets in the path between the two victims." [18] If the packet was to be sent between 2 systems configured as a cluster, the short communications channel between the 2 devices would serve to disable the entire cluster (which was set up as a cluster in the first place to ensure high reliability in the event of node failure).

A more sophisticated version of this attack is known as a "fraggle" attack (this is similar to a smurf -- discussed later under ICMP ping issues). These attacks are named after the hacker script (available for download on the Internet) that demonstrates them as attacks. A "fraggle" is an attack by originated by a broadcast message and takes adavantage of the 'echo' and 'chargen' UDP services. A forged packet is broadcast to the 'chargen' port (UDP port#19) of all hosts receiving the broadcast. These hosts see the spoofed return IP address of the victim; everyone responds with a packet of random data, flooding the victim. A "fraggle amplifier" is any host that has the echo service available. The forged message is sent to this service, which then acts to broadcast it to all hosts on their net, increasing the 'range' of the attack. Since many web servers sit outside of firewalls (in order to securely process requests) and since many have the echo service enabled, this attack is particularly effective.

In his article on UDP viruses,[19] Dr. Frederick Cohen suggests several "other UDP services that are likely to provide environments for protocol viruses ... " including 'systat' (UDP port #11), 'quote of the day' (UDP port #17), 'chargen' (UDP port#19), 'time' (UDP port#37), 'whois' (UDP port#43) and 'who' (UDP port#513).

Remember that any DoS attack at a UDP port can become a DDoS attack. This must be true if the definition of DDoS is multiple distributed attackers, because distributed hosts can be spoofed into participating by various vectors -- from a programmed virus through to infection from 'bad' html after visiting an infected web site.

```
It is simple to get UDP services (echo, time,
daytime, chargen) to
loop, due to trivial IP-spoofing.
that causes the network to become useless. In the
example the header
```

claim that the packet came from 127.0.0.1 (loopback) and the target
is the echo port at system.we.attack. As far as system.we.attack knows
is 127.0.0.1 system.we.attack and the loop has been establish.
Ex:

```
        from-IP=127.0.0.1
        to-IP=system.we.attack
        Packet type:UDP
        from UDP port 7
        to UDP port 7
```

Note that the name system.we.attack looks like a DNS-name, but the
target should always be represented by the IP-number.A great deal of systems don't put loopback on the wire, and simply emulate it. Therefore, this attack will only affect that machine
in some cases. It's much better to use the address of a different machine on the same network. Again, the default services should be disabled in inetd.conf. **Other than some hacks for mainframe IP stacks that don't support ICMP, the echo service isn't used by many legitimate programs, and TCP echo should be used instead of UDP**

Hence, loopback should be avoided by incorporating Fault Tolerance without redundancy ,i.e., *robustness* by including checks for the client IP address.

5.4 Robustness Principles

Client

- **Nothing user does/types should make program crash**
- **Must perform complete checking for user errors**

Server

- **Nothing a client does should cause server to malfunction**
- **Possibly malicious clients**

Things to Worry About

- **Error return codes by system calls**
- **String overflows**
- **Malformed messages**
- **Memory/resource leaks**
- **Especially for server**

6. A CONCURRENT TCP ECHO SERVER

Our project is on TCP echo server that performs the following function:

1. The client reads a line of text form its standard input and writes the line to the server.
2. The server reads the line from its network input and echoes the line back to the client.
3. The client reads the echoed line and prints it on its standard output.

While we will develop our own implementation of an echo server, most TCP/IP implementations provide such a server, using both TCP and UDP. We will also use this server with our own client.
Besides running our client and server in their normal mode (type in a line at watch it echo), we examine lots of our boundary conditions for these example:
1. What happens when the client and server are started?
2. What happened when client terminates normally?
3. What happens to client if the server process terminates before the client is done?
4. What happens to client if the server host crashes?
and so on by looking at all these scenarios and under standing what happens at the network level, and how this appears to the sockets API, we will under stand more about what goes on at this levels and how to code our applications to handle theses scenarios.

TCP ECHO SERVER

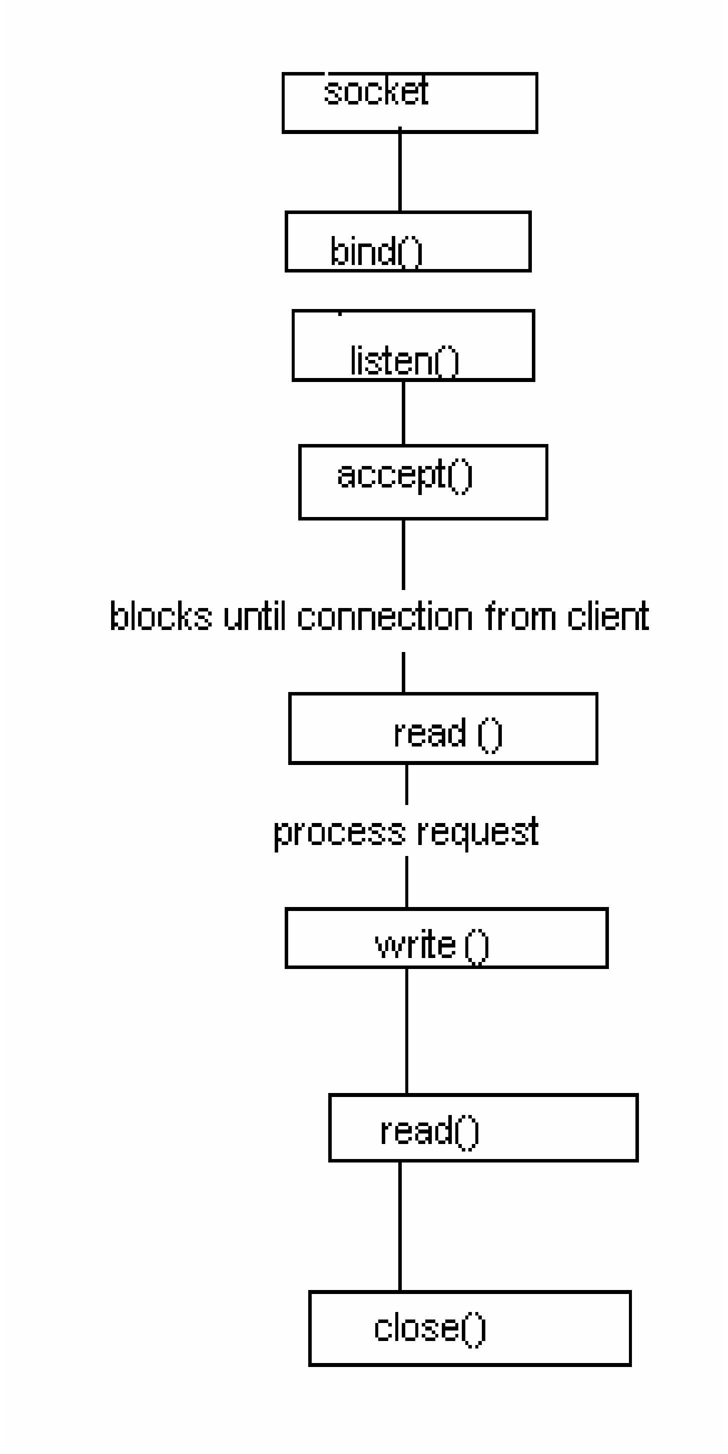


fig.1

6.1 CODE for ECHO SERVER

```
#include<dcecoe.h>
#include<varargs.h>
int errsys(const char * format , . . .)
{va_list ap;
va_start(ap,fmt);
errdo{0,LOG_err,fmt,ap);
va_end(ap);
exit(1);
}

static void
errdo(int errnoflag,int level,const char * fmt,va_list ap)
{int errnosav,n;
char buf[MAXLINE];
errnosav=errno; //value to be printed to caller

#ifdef HAVE_VSNPRINTF
        vsnprintf(buf,fmt,ap);//safe
#else
       vsprintf(buf,fmt,ap);
#endif

n=strlen(buf);
if(errnoflag)
snprintf(buf+n,sizeof(buf)-n,":%s,strerror(errnosav));

strcat(buf,"\n");
fflush(stdout);
fputs(buf,stderr);
fflush(stderr);
return;
}

int errsys(const char * format , . . .)
{va_list ap;
```

```
va_start(ap,fmt);
errdo{1,LOG_err,fmt,ap);
va_end(ap);
exit(1);
}


 int  main(int argc,char **argv)
 {printf("enter mode:a or b");
char mode;
int flag;
do{
getc(mode);
flag=mode;
};
while((flag!=32)||(flag!=33))
int limit=0 ;
 if( flag==33)
{do
{scanf("%d",&limit);};
while(n<=0)

int listenfd, connfd;
 pid_t childpid;
 socklen_t clilen;
 struct sockaddr_in cliaddr, servaddr;

void sid_chld(int);

listenfd = socket (AF_INET, SOCK_STREAM, 0) ;

if(listenfd<0)
errsys("can't create socket:%s\n,strerr(errno));

bzero(&servaddr, sizeof(servaddr)) ;
 servaddr .sin_family =AF_INET;
 servaddr .sin_addr.s_addr =htonl (INADDR_ANY);
 servaddr .sin_port = htons (SERV_PORT) ;

if(bind (listenfd, (SA *) &servaddr, sizeof(servaddr) )<0);
```

```
errsys("bind error);

 if(listenten(listenfd, LISTENQ)<0)
errsys("listen error");

signal(SIGCHLD,sid_chld);

 for ( ; ;) {
 clilen = sizeof (cliaddr);
if( connfd = acccept(listenfd, (SA *) & cliaddr, &clilen)<0)
{if(errno=EINTER)
continue;
else
errsys("accept error");
}
if(htonl(cliaddr.sin_addr)==inet_addr("127.0.0.1"))
{printf("rogue client:shutting down client");
exit(1);}

 if ( (childpid = Fork () ) == 0){                    /* child process*/
 close(listenfed) ;                             /* close listening socket*/
 str_echo (connfd,flag,limit);                        /*  process the request */
 exit (0);
 }

  close(connfd);                    /*  parent closes connected socket*/
   }
}

void  sid_chld(int signo)
{
pid_t pid;
int stat;

while( (pid=waitpid(-1,&stat,WNOHANG) )>0)
printf("child%d terminated\n",pid;

return;
}
```

6.1.1 MAIN FUNCTION

TCP follow the flow of function that we diagrammed in fig.1.we have shown the server program in code segment.

1. Create socket, bind server's well known port
TCP socket is created .An internet socket address structure is filled in with the wildcard address (*inaddr_any*) and servers well known port(*serv_port*). Binding the wild card address tells the system that we will accepts the connection destined for any local interface in case system is multihomed.

2. Wait for client connection to complete
Servers blocks in the call to accept, waiting for a client connection to complete.

3. Concurrent server
For each client, *fork* spawns a child, and then the child handles the new client. The child closes the listen socket and parent closes the connection socket. Then child calls str_echo to handle the client.

6.1.2 str_echo FUNCTION:

The function str_echo, shown in code, performs the server processing for each client: it reads data from the client and echoes it back to the client.

CODE FOR str_echo FUNCTION:

```
#include"dcecoe.h"
void
str_echo(int sockfd,int flag)
{
ssize_t n;
char buf[MAXLINE];
if(flag==32)
{if((n=readline(sockfd,buf,MAXLINE,flag))==0)
return;

if(writen(sockfd,buf,n)<0))
errsys(“write error”);
}
else
if(flag==33)
{char* junkbuf=malloc(MAXLINE);
char c;char * ptr=junkbuf;
int ctr=1;
ssize_t n,rc;
for(n=1,n<maxlen,n++){
again:
                if((rc=read(fd,&c,1))==1)
                           {*ptr++=c;
                              if(c==’\n’)
                             break;
                           }
               else if(rc==0)
                    {if(n==1)
                        return(0);
                    else break;
                    }
                  else
                  {if(errno==EINTER)
                     goto again;
                   return(-1);
                  }
}
if(n==maxlen)
if(read(fd,&c,1)>0)
{if(ctr==limit)
{return;}
ctr++;
ptr= realloc(maxlen*ctr);
*ptr=c;
n=1;goto again;}
}

else
```

```
{
printf("quiting:memory corrupted");
exit(1);}

}

ssize_t
written (int fd, const void * vptr, size_t  n)
{
size_ t nleft;
 ssize_t nwritten;
const char *ptr;
ptr = vptr;
nleft=n;



while(nleft>0)
{
if( (nwritten=write(fd,ptr,nleft))<=0)
       if (errno==EINTER)
               nwritten=0;//call write() agian
        else
              return(-1);//error

nleft-=nwritten;
 ptr+=nwritten;
 }

  return(n);
}
```

Since one call to write is does not guarantee all of n bytes due to the uncertainty associated with tcp layer bufers which have to adjust the flow rate according to the needs of the client we have to call write again(each time noting down the number of bytes written to tcp buffers) & again till all data to be sent back to client is written to tcp layers buffers.

6.1.3 READLINE FUCTIONS

```
 ssize_t readline(int fd,void *vptr,size_t maxlen)
{
  int n,rcnt;
  char c,*ptr;
ptr=vptr;
char junkbuf[MAXLINE];
for(n=1;n<maxlen;n++)
 {
    if(rc=readbuf(fd,&c)==1)//if flag
            {
               *ptr++=c;
               if(c=='\n')
               break;
             }
             else if(rc==0)
             {if(n==1)
             return(0);
             else {*ptr=0;return(n);}
             }
               else
            return(-1);
 }
while( read(fd, junkbuf,sizeof(junkbuf)) );

 *ptr=0;
 return(n);
 }


 static ssize_t
 readbuf(int fd,char *ptr)
 {
            static int readcnt=0;
            static char *readptr,readbuf[MAXLINE];
            if(readcnt<=0)
            {
                        for(;;)
                        {
```

```
          if( (readcnt=read(fd,readbuf,sizeof(readbuf)))<0)
            {
                if(errno==EINTER)
                   continue;
                return(-1);
            }
           else
               { if(readcnt==0)
                return(0);

                readptr=readbuf;
                }

  readcnt--;
  *ptr=*readptr++;
  return(1);
}


}
```

readbuf() buffers the result of read operation in a static buffer & fetches one character each time it is called by readline() to fetch a character.
In this way the costly read system call does not have to be used for each character improving efficiency. readline() must fetch one character at a time to see if the next character signifies an end of line.

<u>6.1.3 HOW IT WORKS:</u>

1.READ A BUFFER AND ECHO THE BUFFER(*LINE8-9*):
Read reads data from the socket and the line is echoed back to the client by *written.* if client closes the connection , the receipt of the client's FIN causes child's *read to return 0*. This causes *str_echo* function to return ,which terminates the child.

2.create socket, fill in internet socket address structure(*line9-13*)

A TCP socket is created and an internet socket address structure is filled in with the server's IP address and port number. We take the server's IP address from the command line argument and server's well known port(*serv_port*)is from our *unp.h* header.

3.connect to server
connect establishes the connection with the server. The function *str_cli* handles the rest of client process.

<u>6.1.4. FEATURES</u>

Overview: The program works in two modes which the user enters when the server starts as a character 'a' or 'b'.
In mode 'a', the client can send message of only MAXLINE bytes. Message greater than that will be truncated & only MAXLINE bytes will be sent back.
In mode 'b' user ,the server person starting the server is asked to enter the length of the largest message acceptable by client from server as a multiple of MAXLINE. Messages of greater length(which may be due to malicious nature of client) are discarded & connection closed.

1)<u>**Robustness**</u>
<u>I)</u> to "*outside attack*" in which a client sends a packet with self looping address as its address to engage the server in sending this message to itself ;which is quickly returned to itself(because its not put on the "wire" thereby wasting resources & makes server less responsive to clients.

ii)By including validation checks for entering of mode(must be a or b only) & limit from server administrator.
iii)Can accept input of any length in mode ’b’ because it uses malloc to allocate the buffer which is readjusted the time new previous allocation of maxlen bytes are insufficient by calling realloc function & its size is increased by maxlen bytes.

2)Interrupted system calls are handled by examining the return status from these calls; if negative ,the global variable errno is checked to see the cause. If errno is EINTER ,it signifies an interrupted system call & the call is re-initiated
This is the incorporation of Fault tolerance in the form of *TEMPORAL REDUNDANCY*.

3)The main parent process of the server closes the connected socket & the child closes the listening socket inherited by it; to prevent the case of an unnecessary connection lingering on.
4)**Use of signal handler for children**. Signal SIG_CHLD is sent to the parent. If no action is taken child will enter zombie state & waste CPU resource. Hence, a signal handler sid_chld(defined in the server) handles the signal by executing a non-blocking waitpid command in a loop so that no SIG_CHLD signals are missed if they arrive simultaneously due to the problem of signals not being queued in UNIX. After parent has executed the waitpid status of process is examine by the parent after which the child “truly” ends
5)Use of wait pid() instead of wait() ;so that no zombies get left behind unattended.

7)Provides valuable feedback to user in case of error or invalid inputs.
8)The server reads only the marline no of arguments & sends back the same to client ;ignoring the rest.Thus, eliminating any chance of a buffer overflow.
9)No printf/snprintf function is used which takes client data(external input) into its format string thereby removing any possibility of a format string attack from the user end.
10)If somehow mode variable flag is overwritten by malicious hacker somehow; the server can detect it very easily & exit stating the corruption of memory as the reason.
The probability of discovering error on flag if it occurs is 0.999969

6.2 Echo Client

6.2.1 open_clientfd

This function opens a connection from the client to the server at host name: port:

```
int open_clientfd(char *hostname, int port)
{
  int clientfd;
  struct hostent *hp;
  struct sockaddr_in serveraddr;

  if ((clientfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    return -1; /* check errno for cause of error */

  /* Fill in the server's IP address and port */
  if ((hp = gethostbyname(hostname)) == NULL)
    return -2; /* check h_errno for cause of error */
  bzero((char *) &serveraddr, sizeof(serveraddr));
  serveraddr.sin_family = AF_INET;
  bcopy((char *)hp->h_addr,(char *)&serveraddr.sin_addr.s_addr,
                                         hp->h_length);
  serveraddr.sin_port = htons(port);

  /* Establish a connection with the server */
  if (connect(clientfd, (SA *)&serveraddr, sizeof(serveraddr))<0)
    return -1;
  return clientfd;
}


int  main(int argc, char ** argv)
{if (argc!=3)
   errquit(usage:cliecho <IPaddress> <port>”);

 int sockfd= open_clientfd(argv[1],argv[2]);
 str_cli(stdin,sockfd);
exit(0);
}
```

//CODE for dcecoe.h

```
#ifndef__dcecoe_h
#define __dce_h
#include “config.h”

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <time.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#include <fcntl.h>
#include <netdb.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/uio.h>
#include <unistd.h>
```

```
#include <sys/wait.h>
#include <sys/un.h>
ifdef HAVE_SYS_SELECT_H
#include  <sys/select.h>
#endif
#ifdef   HAVE_POLL_H
#include <poll.h>
#endif
#ifdef  HAVE_STRINGS_H
#include <strings.h>
#endif
#ifdef  HAVE_SYS_IOCTL_H
#include <sys/ioctl.h>
#endif
#ifdef  HAVE_SYS_FILIO_H
#include <sys/filio.h>
#endif
#ifdef  HAVE_SYS_SOCKIO_H
#include <sys/sockio.h>
#endif
#ifdef  HAVE_PTHREAD_H
#include <pthread.h>
#endif
#ifdef  INADDR_NONE
#define INADDR_NONE 0xffffffff
#ifndef SHUT_RD
#define SHUT_RD
#define SHUT_RD  0
#define SHUT_WR 1
#define SHUT_RDWR   2
#endif
#ifndef  INET_ADDRSTRLEN
#define  INET_ADDRSTRLEN 16
#endif
#ifndef  INET6_ADDRSTRLEN

#define  INET6_ADDRSTRLEN
#endif
```

Config.h is generated by auto conf tool according to features provided by the system e.g support for multicasting,presence of len field in sock structures etc.

7. Single Process Concurrent TCP server

If single-Client Servers were a rather simple case, the multi-Client ones are a tougher nut. There are two main approaches to designing such servers.
The first approach is using one process that awaits new connections, and one more process (or thread) for each Client already connected. This approach makes design quite easy, cause then the main process does not need to differ between servers, and the sub-processes are each a single-Client server process, hence, easier to implement.
However, this approach wastes too many system resources (if child processes are used), and complicates inter-Client communication: If one Client wants to send a message to another through the server, this will require communication between two processes on the server, or locking mechanisms, if using multiple threads.
The second approach is using a single process for all tasks: waiting for new connections and accepting them, while handling open connections and messages that arrive through them. This approach uses less system resources, and simplifies inter-Client communication, although making the server process more complex.
Luckily, the Unix system provides a system call that makes these tasks much easier to handle: the `select()` system call.

The `select()` system call puts the process to sleep until any of a given list of file descriptors (including sockets) is ready for reading, writing or is in an exceptional condition. When one of these things happen, the call returns, and notifies the process which file descriptors are waiting for service.
The select system call is defined as follows:

```
int select(int numfds,
           fd_set *rfd,
           fd_set *wfd,
           fd_set *efd,
           struct timeval *timeout);
```

- numfds - highest number of file descriptor to check.
- rfd - set of file descriptors to check for reading availability.
- wfd - set of file descriptors to check for writing availability.
- efd - set of file descriptors to check for exceptional condition.
- timeout - how long to wait before terminating the call in case no file descriptor is ready.

`select()` returns the number of file descriptors that are ready, or -1 if some error occurred.
We give `select()` 3 sets of file descriptors to check upon. The sockets in the rfd set will be checked whether they sent data that can be read. The file descriptors in the wfd set will be checked to see whether we can write into any of them. The file descriptors in the efd set will be checked for exceptional conditions (you may safely ignore this set for now, since it requires a better understanding of the Internet protocols in order to be useful). Note that if we don't want to check one of the sets, we send a `NULL` pointer instead.
We also give `select()` a timeout value - if this amount of time passes before any of the file descriptors is ready, the call will terminate, returning 0 (no file descriptors are ready).
NOTE - We could use the `select()` system call to modify the Client so it could also accept user input, Simply by telling it to `select()` on a set comprised of two descriptors: the standard input descriptor (descriptor number 0) and the communication socket (the one we allocated using the `socket()` system call). When the `select()` call returns, we will check which descriptor is ready: standard input, or our socket, and this way will know which of them needs service.
There are three more things we need to know in order to be able to use select. One - how do we know the highest number of a file descriptor a process may use on our system? Two - how do we prepare those sets? Three - when select returns, how do we know which descriptors are ready - and what they are ready for?
As for the first issue, we could use the `getdtablesize()` system call. It is defined as follows:

```
int getdtablesize();
```

This system call takes no arguments, and returns the number of the largest file descriptor a process may haves for the second issue, the system provides us with several macros to manipulate fd_set type variables.

```
FD_ZERO(fd_set *xfd)
```

Clear out the set pointed to by 'xfd'.

```
FD_SET(fd, fd_set *xfd)
```

Add file descriptor 'fd' to the set pointed to by 'xfd'.

```
FD_CLR(fd, fd_set *xfd)
```

Remove file descriptor 'fd' from the set pointed to by 'xfd'.

`FD_ISSET(fd, fd_set *xfd)`

check whether file descriptor 'fd' is part of the set pointed to by 'xfd'.

An important thing to note is that `select()` actually modifies the sets passed to it as parameters, to reflect the state of the file descriptors. This means we need to pass a copy of the original sets to `select()`, and manipulate the original sets according to the results of `select()`. In our example program, variable 'rfd' will contain the original set of sockets, and 'c_rfd' will contain the copy passed to `select()`.
Here is the source code ~~of a Multi-Client echo Server. This~~ Server accepts connection from several Clients simultaneously, and echoes back at each Client any byte it will send to the Server. This is a service similar to the one give by the Internet Echo service, that accepts incoming connections on the well-known port 7.

```
#include <stdio.h>              /* Basic I/O routines          */
#include <sys/types.h>     /* standard system types       */
#include <netinet/in.h>    /* Internet address structures */
#include <sys/socket.h>    /* socket interface functions  */
#include <netdb.h>         /* host to IP resolution       */
#include <sys/time.h>      /* for timeout values          */
#include <unistd.h>        /* for table size calculations */

#define  PORT    5060            /* port of our echo server */
#define  BUFLEN  1024            /* buffer length           */

void main()
{
   int              i;           /* index counter for loop operations  */
   int              rc;          /* system calls return value storage  */
   int              s;           /* socket descriptor                  */
   int              cs;          /* new connection's socket descriptor */
   char             buf[BUFLEN+1]; /* buffer for incoming data          */
   struct sockaddr_in sa;          /* Internet address struct          */
   struct sockaddr_in csa;         /* client's address struct          */
   int              size_csa;    /* size of client's address struct    */
   fd_set             rfd;         /* set of open sockets                */
   fd_set             c_rfd;       /* set of sockets waiting to be read  */
```

```
int             dsize;       /* size of file descriptors table     */

/* initiate machine's Internet address structure */
/* first clear out the struct, to avoid garbage  */
memset(&sa, 0, sizeof(sa));
/* Using Internet address family */
sa.sin_family = AF_INET;
/* copy port number in network byte order */
sa.sin_port = htons(PORT);
/* we will accept connections coming through any IP */
/* address that belongs to our host, using the      */
/* INADDR_ANY wild-card.                            */
sa.sin_addr.s_addr = INADDR_ANY;

/* allocate a free socket */
/* Internet address family, Stream socket */
s = socket(AF_INET, SOCK_STREAM, 0);
if (s < 0) {
   perror("socket: allocation failed");
}

/* bind the socket to the newly formed address */
rc = bind(s, (struct sockaddr *)&sa, sizeof(sa));

/* check there was no error */
if (rc) {
   perror("bind");
}

/* ask the system to listen for incoming connections   */
/* to the address we just bound. specify that up to    */
/* 5 pending connection requests will be queued by the */
/* system, if we are not directly awaiting them using  */
/* the accept() system call, when they arrive.         */
rc = listen(s, 5);

/* check there was no error */
if (rc) {
   perror("listen");
}
```

```
/* remember size for later usage */
size_csa = sizeof(csa);

/* calculate size of file descriptors table */
dsize = getdtablesize();

/* close all file descriptors, except our communication socket */
/* this is done to avoid blocking on tty operations and such.  */
for (i=0; i<dsize; i++)
    if (i != s)
        close(i);

/* we initially have only one socket open,   */
/* to receive new incoming connections.      */
FD_ZERO(&rfd);
FD_SET(s, &rfd);
/* enter an accept-write-close infinite loop */
while (1) {
    /* the select() system call waits until any of  */
    /* the file descriptors specified in the read,  */
    /* write and exception sets given to it, is     */
    /* ready to give data, send data, or is in an   */
    /* exceptional state, in respect. the call will */
    /* wait for a given time before returning. in   */
    /* this case, the value is NULL, so it will     */
    /* not timeout. dsize specifies the size of the */
    /* file descriptor table.                       */
    c_rfd = rfd;
    rc = select(dsize, &c_rfd, NULL, NULL, NULL);

    /* if the 's' socket is ready for reading, it   */
    /* means that a new connection request arrived. */
    if (FD_ISSET(s, &c_rfd)) {
        /* accept the incoming connection */
        cs = accept(s, (struct sockaddr *)&csa, &size_csa);

        /* check for errors. if any, ignore new connection */
        if (cs < 0)
            continue;
        /* add the new socket to the set of open sockets */
        FD_SET(cs, &rfd);

        /* and loop again */
        continue;
    }

    /* check which sockets are ready for reading, */
    /* and handle them with care.                 */
    for (i=0; i<dsize; i++) {
        if (i != s && FD_ISSET(i, &c_rfd)) {
            /* read from the socket */
            rc = read(i, buf, BUFLEN);

            /* if client closed the connection... */
            if (rc == 0) {
                /* close the socket */
                close(i);
                FD_CLR(i, &rfd);
            }
            /* if there was data to read */
            else {
                /* echo it back to the client       */
                /* NOTE: we SHOULD have checked that */
                /* indeed all data was written...    */
                write(i, buf, rc);
            }
        }
    }
}
```

8. ECHO SERVER USING THREADS

Memory is copied from the parent to the child, use a technique copy on write which avoids a copy of the parent's data space to the child need its own copy.Interprocess communication(IPC) is required to pass information between parent and child after the fork.
Threads are called lightweight processes since a thread is "lighter weight" than a process. Threads creation can be 10-100 faster than the process creation. All threads within a process share the same global memory. This makes the sharing of information easy between the threads, but with the problem of synchronization.
All process within threads share:

- Process instructions,
- Most data,
- Open files(e.g. descriptors),
- Current working directory, and,
- User and group Ids

Each thread has its own :

- Thread ID,
- Set of registers ,including program counter and a stack pointer,
- Stack(for local variables and return addresses)
- Errno
- Priority

<u>8.1 BASIC THREAD FUNCTIONS</u>

Phthead_create function

```
#include<phthread.h>
int phhead_create(phthread_t  *tid, const phthread_attr_t *attr,
                  void * (*func)(void  *), void *arg);
                      return :0 if ok,positive Exxx value on error
```

each thread within a process is identified by a thread ID whose data type os pthread_t. on successful creation of a new thread , its ID is returned through the pointer *tid.W*hen a thread is created we can specify these attributes by initializing a pthread_attr_t variable that overrides the default.

pthread_join function

We can wait for a given thread to terminate by calling to terminate by calling pthread_join. Comparing threads to UNIX processes, pthread_create is similar to fork ,and pthread_join is similar to waitpid

```
#include <pthread.h>
int pthread_join(pthread_t tid, void ** status);
                    return: 0 if ok, positive Exxx value on error
```

pthread_self function

Each thread has an ID that identifies it within a given process. The thread ID is returned by pthread_create and we saw it was used by pthread_join. A thread fetches this value for itself using pthread_self.

```
#include <pthread.h>
pthread_t pthread_self(void);
                         return: thread ID of calling thread
```

pthread_detach function

A thread is either is either joinable (the default) or dethached. When a joinable threat terminates, its thread ID and exit status are retained until another thread calls pthread_join . but a dethached thread is like a daeomon process: when it terminates all its resources are released and we cannot wait for it to terminate. If one thread needs to know when another thread terminates, it is best to leave the thread as joinable.

The pthread_detach function changes the specified thread so that it is dethached.

```
#include <pthread.h>
int pthread_detach(pthread_t tid);
```

<u>pthread_exit function</u>

one way for a thread to terminate is to call pthread_exit.

```
#include <pthread>
void pthread_exit (void *status);
```

If the thread is not detached, its thread ID and exit status are retained for a later pthread_join by some other thread in the calling process.
The pointer status must not point to an object that is local to the calling thread, since that object disappears when the thread terminates.

There are wo other ways for a thread to terminate.
The function that started the thread (the third argument to pthread_create) can return. Since this function must be declared as returning a void pointer,that return value is the exit status of the thread.
If the main function of the process returns or if any thread calls exit, the process terminates,including any threads.

str_cli function using threads

```
#include "unpthread.h"
void  *copyto(void *);
static int sockfd;
static FILE *fp;
void
str_cli(FILE *fp_arg, int sockfd_arg)
{
        char  recvline[MAXLINE];
        pthread_t tid;
        sockfd = sockfd_arg;
        fp = fp_arg;
        Pthread_create(&tid, NULL, copyto, NULL);
        while (Readline(sockfd, recvline, MAXLINE) > 0)
            Fputs(recvline,stdout);
}
```

```
void *
copyto(void *arg)
{
            char  sendline[MAXLINE];
            while  (Fgets(sendline,MAXLINE,  fp) != NULL)
                Written(sockfd,sendline,strline,strlen(sendline));
            shutdown(sockfd, SHUT_WR);
            return (NULL);
}
```

8.2 ECHO SERVER WITH THREADS

```
#include     "unpthread.h"
static void *doit(void *);
int
main(int argc, char ** argv)
{
    int listenfd,*iptr;
    socklen_t addrlen, len;
    struct sockaddr *cliaddr;
    if (argc == 2)
    listenfd = Tcp_listen(NULL, argv[1],&addrlen);
    else if (argc == 3)
    listenfd =Tcp_listen(arg[1],argv[2],&addrlen);
    else
                                                                  err_quit("usage:
tcpserv01 [ <host> ] <service or port>");
    cliaddr = Malloc(addrlen);

    for ( ; ; )  {
    len = addrlen;
    iptr = Malloc(sizeof(int));
    *iptr = Accept(listenfd,cliaddr,&len);

    Phthread_create(NULL, NULL, &doit, (void *) connfd);
    }
}
```

```
static void *
doit(void * arg)
{
     int connfd;
     connfd = *((int*) arg);
     free(arg);

     Phtread_detach(phthread_self());
     str_echo(connfd);
     close(connfd);
     return (NULL);
}
```

<u>unpthread.h header</u>

It includes our normal unp.h header, followed by the Posix.1<pthread.h> header, and then defines the function prototypes for our wrapper versions of the pthread_XXX functions which all begin with Pthread_.

Save arguments in externals

The thread that we are about to create needs the values of the two arguments to str_cli:fp, the standard I/O FILE pointer for the input file and sockfd,the TCP socket that is connected to the server .
An alternative technique is to put the two values into a structure and then pass a pointer to the structure as the argument to the thread that we are about to create.

Create new thread

The thread is created and the new thread ID is saved in tid. The function that is executed by the new thread is copyto. No arguments are passed to the thread.

MAIN THREAD LOOP: COPY SOCKET TO STANDARD OUTPUT

The main thread calls readline and fputs , copying from the socket to the standard output

TERMINATE

When the str_cli function returns, the main function terminates by calling exit.
When this happens ,all threads in the process are terminated. In the normal scenario the other thread has already terminated when it read an end of file on standard input. But in case the server terminates prematurely, calling exit terminates the other thread, which is what we want.

Copy to thread

This thread just copies standard input to the w socket. When it reads an end of file on standard input to the socket. When it reads an end-of-file on standard input, a FIN is sent across the socket by shutdown and the thread returns. The return from this function terminates the thread.

Create a thread

When accept returns, we call pthread_create instead of fork. The first argument is a null pointer, since we don’t care about the thread ID. The single argument that we pass to the doit function is the connected socket descriptor,connfd.

Thread function

Doit is the function executed by the thread. The thread detaches itself, since there is no reason for the main thread to wait for each thread that it creates. The function str_echo does not change .when this function returns ,wemust close the connected socket, since the thread shares aall descriptors with the main thread. With fork, the child did not need to close the connected socket because the child then terminates and all open descriptors are closed on process termination.

PASSING ARGUMENTS TO NEW THREADS

We cast the integer variable connfd to be a void pointer, but this is not guranteed to work on all systems. To handle this correctly requires additional work.

__//tcp_listen function

```
#include     "unp.h"
int
tcp_listen(const char *host, const char *serv, socklen_T *addrlenp)
{
     int    listenfd, n;
     const int on = 1;
     struct addrinfo hints, *res,  *ressave;

     bzero (&hints, sizeof(struct addrinfo));
     hints.ai_flags = AI_PASSIVE;
     hints.ai_family = AF_UNSPEC;
    hinta.ai_SOCKTYPE = SOCK_STREAM;

     if ( n = getaddrinfo(host, serv, &hints, &res)) != 0)

     errquit("tcp_listen error for %s,%s: %s",
         host,serv, gai_strerror(n));
     resave = res;

     do {
     listenfd = socket(res->ai_family, res->ai_socktype, res-
                                                        >ai_protocol);
     if (bind(listen < 0)
     continue;
     Setsockopt(listenfd,SOL_SOCKET, SO_REUSEADDR, &on,
                                                                      sizeof(on));
     if (bind(listenfd, res->ai_addr, res-.ai_addrlen) ==0)
     break;
     Close(listenfd);
     } while  ((res = res->ai_next) !=NULL);
     if(res == NULL)

     errsys("tcp_listen errorfor %s, %s", host,serv);
     Listen(listenfd,LISTENQ);
     if (addrlenp)
     *addrlenp = res->ai_addrlen;

     freeaddrinfo(resave);

     return(listenfd);
}
```

8.3 THREAD –SPECIFIC DATA

There is a common problem when converting existing functions to run in a threads environment. Solutions are:

Use thread-specific data. This is nontrivial and then converts the function into one that works only on systems with threads support. The advantage to this approach is that the calling sequence does not change and all the changes go into the library function and not the applications that call the function. We show a version of readline that is thread-safe by using thread-specific data.

Change the calling sequence so that the caller packages all the arguments into a structure, and also store in that structure the static variables.

Thread-specific data is a common technique for making an existing function thread-safe. Before describing the Pthread functions that work with thread specific data, we describe the concept and possible implementation, because the functions appear more complicated than they really are.

readline function using thread-specific data

Without changing the calling sequence ,converting the optimized version of our readline function to be thread safe
the pthread_key_t variable, the pthread_once_t variable,the readline_destructor function the readline_once function and our Rline structure that contains all the information we must maintain on a per-thread basis

destructor

our destructor function just frees the memory that was allocated for this thread.

One-time function

One-time function is called by pthread_once, and it just creates the key that is used by readline.

Rline structure

Our Rline structure contins the three variables that caused the problem by being declared static .one of these structures will be dynamically allocated per thread, and then realeased by our destructor function.

my_read function

The first argument to the function is now a pointer to the Rline structure that was allocated for this thread.

allocate thread-specific data

We first call pthread_once so that the first thread that calls readline in this process calls readline_once to create the thread-specific data key.

Fetch thread-specific data pointer

Pthread_getspecific returns the pointer to the Rline structure for this thread. But if this is the first time this thread has called readline, the return value is a null pointer. In this case e allocate space for an Rline structure and the rl_cnt member is initialized to 0 by calloc. We then store this pointer for this thread by calling pthread_setspecific. The next time this thread calls readline pthread_getspecific will return this pointer that was just stored.

getaddrinfo function

The getaddrinfo function hides all of the protocol dependencies in the library function, which is where they belong. The application deals only with the socket address structure that are filled in by getaddrinfo.

```
#include<netdb.h>
int getaddrinfo (const char  *hostname,  const char * service,
              const struct addrinfo *hints, struct addrinfo ** result);
```

return : 0 if ok,nonzero on error

this function returns, through the result pointer, a pointer to alinked list of addrinfo structures,which is defined by including<netdb.h>:

```
struct addrinfo {

int    ai_flags;
int    ai_family;
int    ai_socktype;
int    ai_protocol;
size_t aiaddrlen;
char   *ai_canonname;
struct sockaddr   *ai_addr;
struct  addrinfo   *ai_next;

};
```

gai_strerror function

the non zero error returns values from getaddrinfo have the names and meanings .the function gai_strerror takes one of these values as an argument and returns a pointer to the corresponding error string.

```
#includen<netdb.h>
char  *gai_strerror(int error);
```

freeaddrinfo function

All of the storage returned by getaddrinfo, the addrinfo structures, the ai_addr structures, and the ai_canonname string are obtained dynamically from malloc.this srorage is returned by calling freeaddrinfo.

```
#include<netdb.h>
void freeaddrinfo(struct addrinfo *ai);
```

ai should point to the first of the addrinfo structures returned by getaddrinfo. All the structures in the linked list are freed, along with any dynamic storage pointed to by those structures

tcp_listen function

tcp_listen ,performs the normal TCP server steps: create a tcp socket, bind the server's well known portend allow incoming connection requests to be accepted

#include "unp.h"
int tcp_listen(const char *hostname, const char *service, socklen_t *lenptr);
returns: connected socket descriptor if ok, no return on error

call getaddrinfo

We intialize an addrinfo structure with our hints:AI_PASSIVE, since this function is for serverAF_UNSPEC for the address family,and SOCK_STREAM.

create socket and bind address

The socket and bind functions are called. If earlier call fails we just ignore this addrinfo structure and move on to the next one. we always set the SO_READDUSER socket option for a TCP server.

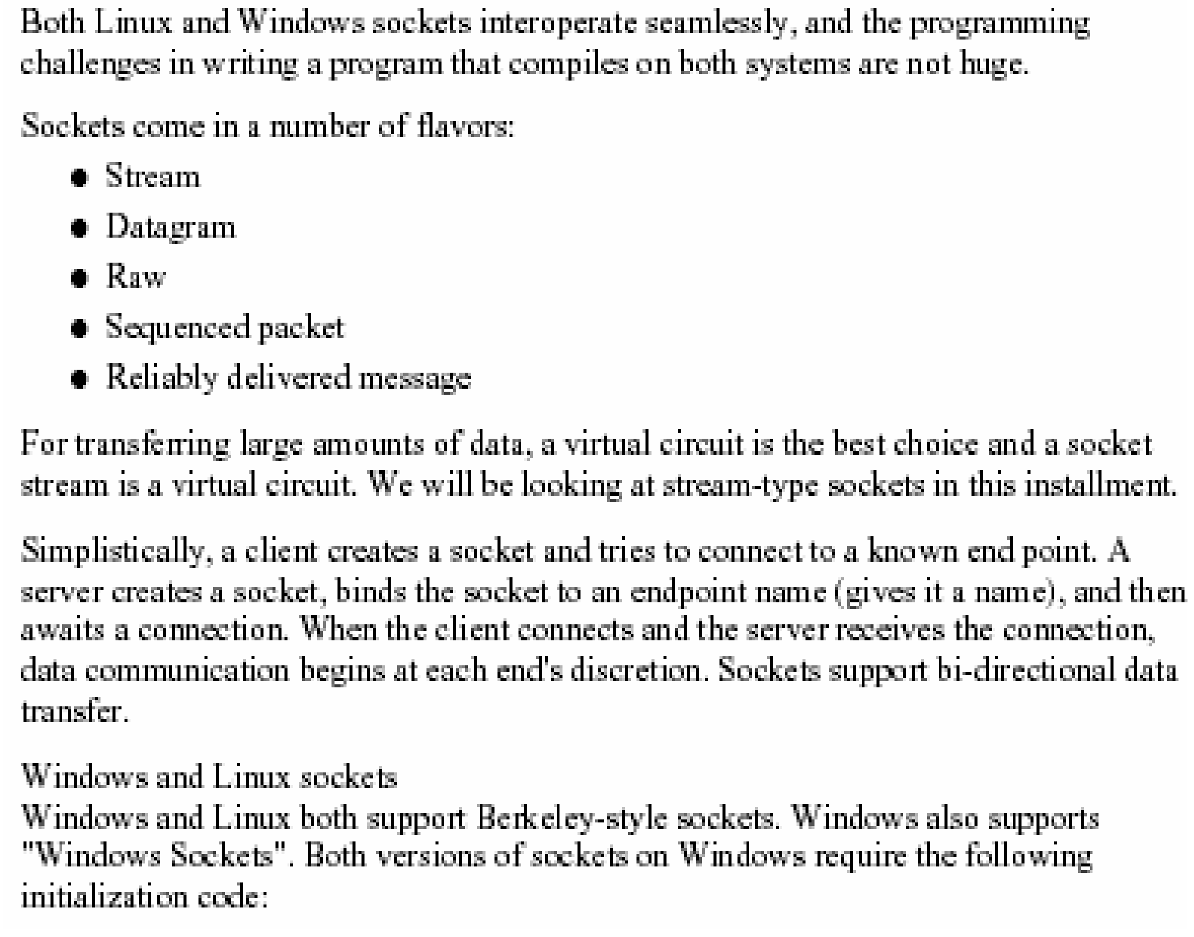
Check for failure

If all the calls to the socket and bind failed, we print an error and terminate .as with our tcp_connect function .
This socket is turned into a listening socket by listen.

Choice of OS:

Comparison of Linux & Windows' sockets

We chose Linux/Unix platforms over windows because of the information gathered below which repeatedly pointing out the fact that Linux sockets were better than windows sockets. Apart from this windows other ipc mechanism like pipes are also inefficient for small block transfers as compared with those of Linux.
Here is an excerpt from the report: that suggested that Linux sockets were faster than windows sockets which were very cumbersome to program.



Both Linux and Windows sockets interoperate seamlessly, and the programming challenges in writing a program that compiles on both systems are not huge.

Sockets come in a number of flavors:

- Stream
- Datagram
- Raw
- Sequenced packet
- Reliably delivered message

For transferring large amounts of data, a virtual circuit is the best choice and a socket stream is a virtual circuit. We will be looking at stream-type sockets in this installment.

Simplistically, a client creates a socket and tries to connect to a known end point. A server creates a socket, binds the socket to an endpoint name (gives it a name), and then awaits a connection. When the client connects and the server receives the connection, data communication begins at each end's discretion. Sockets support bi-directional data transfer.

Windows and Linux sockets
Windows and Linux both support Berkeley-style sockets. Windows also supports "Windows Sockets". Both versions of sockets on Windows require the following initialization code:

WSAStartup

```
#ifdef _WIN32
```

```
WSADATA wsadata;
rc = WSAStartup(2, &wsadata);
if(rc) {
printf("WSAStartup FAILED: err=%d\n",
GetLastError());
return 1;
}
#endif
```

Here, `2` is a version. Using any non-zero number as the first argument works. (The first argument is an unsigned short.)

Windows sockets seems to support other transport protocols. However, with the uniform acceptance of the Internet and
its TCP/IP protocols, I don't understand the value of the added complexity of Windows Sockets.

With one exception, I have written the program in this article using the Berkeley-style interfaces.

Socket creation

Sockets are created with the `socket()` API supported on Linux and Windows:

socket() API

```
SOCKET socket(
int af,
int type,
int protocol
);
```

`af` is address family, and I use `AF_INET`. `type` is either a `SOCK_STREAM` or `SOCK_DGRAM`, and here I focus only
on `SOCK_STREAM`. `protocol` is a number selected from the /etc/protocols file on Linux and the
\winnt\system32\drivers\etc\protocol file on Windows. I'm sticking with 0, which is the IP protocol on both systems.

Windows also has the Microsoft proprietary interface called Windows Sockets

Windows structures for the proprietary sockets have a large number of protocol related fields leading to complexity. The complexity issue is a real one. Programs are written using existing documentation, and programmers expect the documentation to be correct. If a simple API like the `sockets()` API is complicated with additional parameterization and there is little likelihood that the additional parameter space is fully tested, a

compelling reason should exist before using the larger parameter space. Programmers are well advised to stick to the main roads.
Microsoft and Linux won't be making mistakes in simple socket open/connect/send-recv/close situations. However, the darker corners of a complex API are much less likely to be fully tested. Enormous amounts of time can be wasted trying to get obscure features of an API to perform as documented. In almost all cases a little more work on the part of the programmer would allow him or her to avoid the untested paths of complex API()s.
With this thought in mind, we computed the size of the additional parameter space of the WSASocket() API(proprietary) . The complexity discovered here is in addition to the still present parameterizations of the Berkeley Sockets interface. From
the above possibilities, there appears to be
32 * 3 * 2 * 524,288 = 100,663,296
combinations of calls possible to present to the Windows operating system. This number does not include any parameterization on the first three parameters of the `WSASocket()` API call. It is hard for me to understand how even a substantial subset of these parameterizations can be tested or verified. I confess that I don't understand the use or even meaning of some of these parameters. For the purpose of this column, unless a reader can show how a program might be improved by the use of the more obscure interfaces of WSA sockets, we will avoid them.
We used the `WSASocket()` API but with a `NULL WSAProtocol_Info` structure and only using the `WSA_FLAG_OVERLAPPED` bit. The `WSA_FLAG_OVERLAPPED` is documented to be meaningful only when the parameterizations of `WSASend`, `WSASendTo`, etc. reflect an overlapped IO request. I don't use them either. Thus, the parameterizations I use with

Windows Sockets (the WSA interfaces) are identical to the ones I use with the standard Berkeley-style interface. Those of you who have a better understanding of the WSA interfaces and overlapped IO issues might want to try to see if you can improve the performance numbers presented here.

Once we get past the preliminaries, socket programming on Windows and Linux is quite similar. Here is the client
code that performs the connection to a listener:
Connecting
```
if(connect(sock2, (struct sockaddr *)&addr1,
sizeof(addr1))) {
```

```
printf("connect FAILED: err=%d\n", errno);
return 1;
}
```
There are no conditional definitions needed. The parent creates a sockets and awaits a connection with this code.

Accepting a connection
```
rc = listen(sock1,1);
if(rc) {
printf("Listen FAILED: err=%d\n", errno);
return 1;
}
sock3 = accept(sock1, (struct sockaddr *)&addr2,
(socklen_t *)&addr2len);
if(sock3 == BADSOCK) {
printf("Accept FAILED: err=%d\n", errno);
return 1;
}
```
Once again, no conditional definitions. Finally, the transmission and reception of data.

Sending and receiving data

Code to send data is:

Socket send() operation
```
rc = send(sock3, (char *)&wtoken[0], 1, 0);
if(rc == SOCKERR) {
printf("send (1) FAILED: err=%d\n", errno);
return 1;
}
```
and

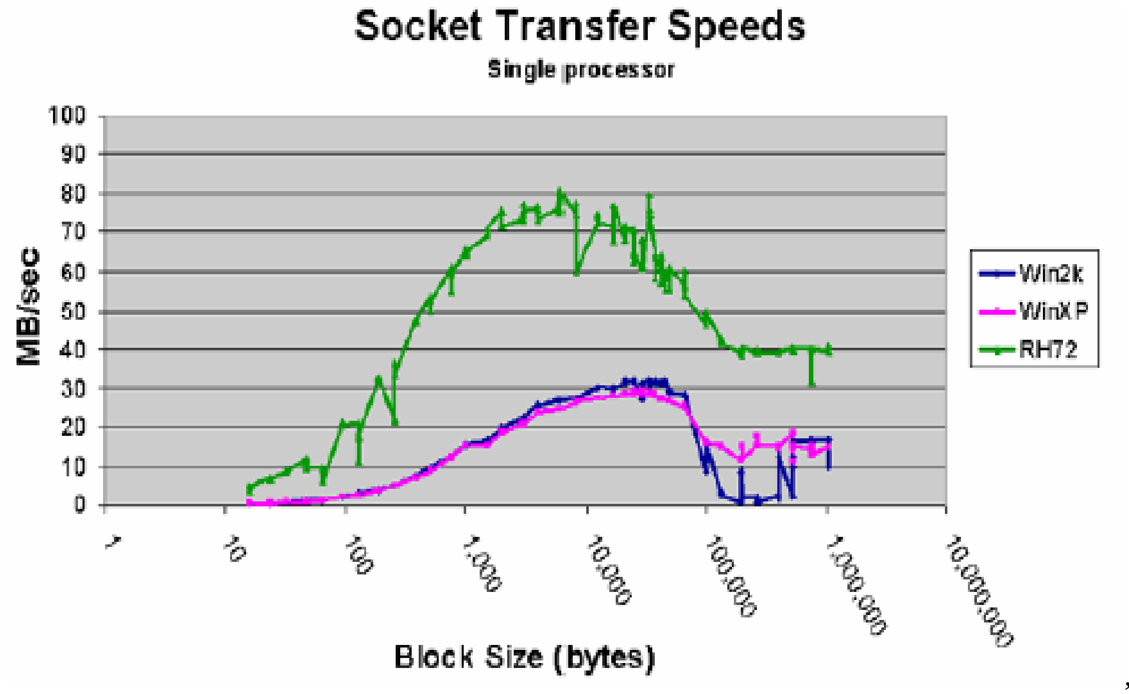
Socket recv() operation
```
rc = recv(sock2, (char *)&rtoken[0], 1, 0);
if(rc == SOCKERR) {
printf("recv (1) FAILED: err=%d\n", errno);
return 1;
}
```
Sockspeed6(A program) was used to look at the data transfer rates from one process to another process on the same machine. The values generated should give us a good idea of potentially how fast the underlying networking code can transfer data independent of the media speed. I investigated transfer sizes from 16 bytes to 1 megabyte. I did not investigate the no-delay option (turning off the Nagel algorithm), the non-blocking option, or

changing the receive or transmit buffer sizes. Those investigations await another column.

Results

All tests were run on an IBM ThinkPad 600X with 576 MB of memory and an 18-GB disk. The system boots all three operating systems. Figure 1 shows the results for Windows 2000 Advanced Server, Windows XP and Red Hat 7.2 (Linux 2.4.2).

Professional



,

During the development and measurements presented here, I noticed that Windows 2000 seemed to perform better when it used the local host IP address, 127.0.0.1. I re-ran the test for all three platforms using the local host IP address. Figure 2 shows the results. For the 127.0.0.1 tests, I just plotted skinny lines with markers on top of Figure 1. It appears that Windows 2000 does indeed transfer data faster over the 127.0.0.1 address. However, it also shows that Windows XP seems to have removed this feature.

<u>10. Suggestion for improvement & other alternatives</u>:-

1)The child processes waste their time slice waiting for I/O;instead they should poll their connections & if no request is pending they should relieve their time slot. This can be done by embedding assembly code which uses the int 0x80H interrupt for relieving time slice from a process.
2)Alternative use of single process server by I/O multiplexing based on select primitive.
& totally avoid process/thread overhead. Such a server would need timeout I/O to prevent a rogue client from disrupting service by sending one byte & then sitting idle causing server to block waiting for the input.
3)We have developed many versions of the same echo server. Though this may seem unnecessary but all the versions can be incorporated into one <u>big **Fault Tolerant Echo serve**r</u>. In all the versions if some error other than interrupt occurred the program exited. These conditions can be seen as failure in one acceptance test out of the given set of acceptance tests(not needing to be coded by the user; actually done by tcp layer in the kernel).Upon such failure we can do two things: first, wait for some time entering into a “freeze state” so that the external conditions(network related temporary problems) stabilize & then re-launch the server as a child process of the original server (with the provision that parent will kill itself after forking) or launching another version after some time (based on load checking or some random sequence)& in the second option
we see if the error is not of temporary nature then we launch the new version automatically. The <u>*launching of new version*</u> can be preceded by <u>*load analysis*</u> & the <u>*analysis of the cause of failure*</u> & <u>*choosing an appropriate version which minimizes such error & fits the current load requirements best.*</u>
E.g. A single process server even though it may be faster& requires less resources than multi process server but since number of descriptors available to a single process is limited & if load is big or ‘Syn’ Flooding takes place; the server may run out of the number of descriptors needed so we choose the multiprocess version. Also the user can choose appropriate server, specific to his system requirements less memory choose single process ; more resources threaded/process.
Small load expected :iterative lest concurrent & so on & so forth.

It can also be seen that only one version of UDP server was developed. This is because UDP servers are more vulnerable than TCP ones & in case of attack which is the usual cause of failure it is better to shut it off so that no more havoc is wreaked on the network.

<u>*Another innovation*</u> in this design can be the use of "**<u>restore points"</u>** to enable restart of new service. A dump of the stack of the main server process can be created after certain intervals of time in a pre denied bufffer.In case of failure the original Server can *revert back to the previous dump* in *which connection descriptors, parameters of the malicious client* which may have caused failure or those of the new set of clients part of the new set of received connections whose large magnitude could not be managed by the server *are absent.* Then the other(or same version) can be launched (or re-launched) which provides almost *uninterrupted* service to its older clients(when the original server was stable).How it is uninterrupted stems from the fact that new version is handed over(inherits) the previous descriptors by original version(which is explained in next to next Para.)

The time interval of setting restore points can be calculated like that of the retransmission timer of TCP or based on the dynamics of client requests. For this each clients session length should be evaluated simultaneously & used to calculate the average session length of the client. The restore should occur in less than this time to be effective .How much less is determined by the client queues length & the server's & system's kernel tables, buffer, qeue's length etc..Larger queue/memory space means more ability to store restore data & frequent restores. also large buffers mean more clients can be handled by newly started version so more data is stored in the dump.

Restart of new service can be done by forking a child which also in "freeze state"(sleeping) & then causing to load another version using exec system call))*which suits the current conditions which are observed by the failed version. Thus the new version inherits the old descriptors(sockets) from parent to provide uninterrupted service.*
The new set of descriptors start from 0,1,and 2.
This "suitability" can be determined according to some pre- defined criteria stored in a file(which will be big :storing solutions for all cases) for each of the conditions
Or

generating some metric for each version basic on certain attributes enlisted for each of the criteria. These attributes are essentially performance measures(speed at load of x queued incomplete/complete connections) & can be set for a particular system after rigorous set of experiments. These metrics allow us a way to measure the suitability of a version on more than one type of apparently non-comparable factors e.g probability of failure at that particular load & expected response times at a particular load.
*One such simple approach to calculate attributes can be enlisted as a linear interpolation of the experimental results(obtained by carrying out least mean squares approximation) ;e.g. queued connections per unit load attribute is k(at avg of that at all loads) allows us to calculate the" expected queue length metric " at load y as (k*y). Other approaches for non-linear interpolations can be dealt at master's level in the presence of ample time.*

4)The buffer to read from sockets could be allocated in chunks of maxline size as per need
so as to allow client to send as much data as it wants in one go & so that precious memory space of server is not wasted if client sends only few characters per line.
Of course an upper limit could be sent even for the dynamically allocated memory crossing which the connection is closed. The client may be marked as a "possible rogue" & this information stored in a file which will be consulted while accepting connections & connection can be rejected if some "pre- defined apology" packet is not received from client. the apology packet will have some pre defined massage.

5)The Echo server can be made network aware; able to identify location of client whether it is on same LAN or not; if not then in case resources are scarce(in case of failure of previous version);the server can just process its local clients, i.e. only connections from local clients are accepted for sessions & those from outside neglected in times of trouble.
A simple approach is to store the list of possible clients on local LAN in a file of proper format preferably using indexed B tree/B*tree organization(record size one bit: present on LAN or not) entries of clients; load that file into a buffer & search it
While accepting connection when load is very high. Obviously, this approach requires load monitoring sub-system meaning overhead which means poorer diagnosis & testing capability.

6)Another idea if for server to periodically poll the sockets & if no socket is ready for
I/O then it can go to sleep & redpoll again if request is there it awakens & processes it.
Thus it has two states: sleeping-polling mode & normal client processing mode like that of any other echo server. This version can be used if echo service is used occasionally & can relieve system of time slice & recourses needed by echo server when it blocks on its I/O . Obviously, it can relieve its time slice to other processes in its sleep mode if it finds no request after polling.
7)At the end we propose a *radical approach* to the design of the server in the optimized for operation on LAN networks suffering from severe congestion ,based on broadcast strategy. In this approach the server blocks on a call to select & after returning processes each clients line & for each of the 26 alphabets marks for each of the client how many times & at what locations the character appears. After compiling such data for all ready clients it creates a multicast packet for each of the alphabets(one after another maintaining this state info & sharing it with the client) to all the ready clients in which it mentions the offsets from the beginning of the line where the letter appears in the message(timeout I/O is used so that no rogue client may block the server on its call to read ; a list of no. of timeouts out of read call to each client is maintained & if it exceeds a maximum limit the client is removed from the list of ready clients & marked as rogue if is removed from ready list at more than certain number of times(the list of which is maintained & cleared periodically in a dynamic cache)) . So for n ready clients n-records for each client is sent as multicast .So after 26 such multicasts all ready clients receive their full complete message & the redundant information does not clog the network. After this the server may go back to select call. Also the client is modified to send only the Huffman coded message which is treated as a ASCII byte stream message by the server & echoed back using the multicast/broadcast method the client receives the Huffman coded message along with the decoding information it encapsulated at the beginning of the frame it sent to the server. In this way large messages could be sent to the server & received in 26 multicast without the server having to bear with process creation, switching overhead.
8) Having two ports per child processes/threads , one for sending data & the other one for receiving data with stdout ‘s access controlled by a semaphore.
9)Code the echo server in assembly for better efficiency.

Bibliography& References :

1) Douglas Comer, David Stevens, “Internetworking with TCP/IP: Client-Server Programming and Applications”, Volume III, Prentice Hall.
2)Stevens w.,”Unix Network Programming, Volume I, Prentice Hall.

Internet sites
www.cis.temple.edu
www.ibm.com