SOCKET OPTIONS

There are a number of options like – send buffer size, bypass table look up, get socket type, to name a few - that needs to be set. Also these needs to be got back also. Towards realizing this, there are ways to get and set socket options. These are:

- The ***getsockopt*** and ***setsockopt*** functions
- The ***fcntl*** functions and
- The ***ioctl*** functions

The options are different for IPv4, IPv6, TCP and for generic categories. These options are briefly discussed in this chapter.

Fcntl is the Posix way to set a socket for non blocking I/O, signal driven I/O and to set the owner of a socket. Ioctl is similar to fcntl but also has added functionalities that are not defined in the fcntl.

The syntax for the ***getsockopt*** and ***setsockopt*** is givenbelow:

#include <sys / socket.h>

int getsockopt(int sockfd, int level, int optname, void *optval, socklen_t, *optlen);

int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t, *optlen);

both return : 0 if OK –1 on error.

The sockfd refers to an open socket descriptor. The level specifies the code in the system to interpret the option. The general socket code, or some protocol specific (IPv4 or IPv6 or TCP etc) The optval is a pointer to a variable from which the new value of the option is fetched by setsockopt, or into which the current value of the option is stored by the getsockopt. The size of this variable is pecified by the final argument, as a value result for getsockopt.

The list of options that can be set and get by the socket options are listed in the fig 7.1 (page 179.) These options, in additions being classified based on the protocol, they are classified as either binary options (shown as flags) that enable or disable a certain features and other options that fetch and return specific values that we can either set or examine.

[Here are the functions for examining and modifying socket options. They are declared in `sys/socket.h`*.*

— *Function: int* ***getsockopt*** *(*int socket, int level, int optname, void *optval, socklen_t *optlen-ptr*)*

The `getsockopt` *function gets information about the value of option* optname *at level* level *for socket* socket*.*

The option value is stored in a buffer that optval *points to. Before the call, you should supply in* *optlen-ptr *the size of this buffer; on return, it contains the number of bytes of information actually stored in the buffer.*

Most options interpret the optval *buffer as a single* `int` *value.*

The actual return value of `getsockopt` *is* `0` *on success and* `-1` *on failure. The following* `errno` *error conditions are defined:*

`EBADF`
The socket *argument is not a valid file descriptor.*
`ENOTSOCK`
The descriptor socket *is not a socket.*
`ENOPROTOOPT`
The optname *doesn't make sense for the given* level.
Function: int ***setsockopt*** *(*int socket, int level, int optname, void *optval, socklen_t optlen*)*

This function is used to set the socket option optname at level level for socket socket. The value of the option is passed in the buffer optval of size optlen.]

The data type column shows the data types of what the ***optval*** pointer must point to for each option. The two braces notation {} is used to indicate a structure.

There are two basic types of options : binary options that enables or disables a certain features (flag), and options that fetch and return specific values that we can either set or examine values. The column labeled "flag" specifies if the option is a flag option. When calling ***getsockopt*** for these flag options, ***optval*** is an integer. The value returned in optval is zero if the option is disabled, or nonzero if the options is enabled. Similarly, ***setsocket*** requires a nonzero optval to turn the options on and a zero value to turn the option off. If the flag column does not contain a * then the options is to turn the options is used to pass a value of the specified datatype between user process and the system.

Level	Optname	Get	Set	Description	Flag	Datatype
SOL_SOCKET	SO_BROADCAST	*	*	Permit Sending Of Broadcst Datagrams	*	Int
	SO_DEBUG	*	*	Enable debug tracing	*	int
	SO_ERROR	*		Get Pending Error And Clear		int
	SO_TYPE	*		Get Socket Type		int
	SO_LINGER	*	*	Linger on close if data to send		Linger{}
	SO_RCVTIMEO	*	*	receive time out		Timevalue{}
IPPRTO_IP	IP_TOS	*	*	Type Of Service And Precedence		Int
	IP_MULTICAST_IF	*	*	Specify outgoing interface		Int_addre{}
IP_PROTOICMPV6	ICMPV6_FILTER	*	*	Specify ICMPv6 message types to pass		Icmp6_ffilter
IPPROTO_IPV6	IPV6ADDFORM	*	*	Change address format of socket		Int
IPPROTO_TCP	TCP_KEEPALIVE	*	*	Seconds Between Keep alive Probes		Int

The code in 7.2, 7.3 and 7.4 provides a method to to find out if the given option is supported and if so to print the default value.

GENERIC SOCKET OPTIONS:

These socket options are protocol independent meaning that the protocol independent code within the kernel handles these and not particular module of any protocol. Some options apply to only

certain types of sockets. For example, SO-BROADCAST socket options applied only to datagram socket although it is listed under GENERIC Sockets.

SO_BROADCAST Socket Option: This socket option enables or disables the ability of the process to send broadcast messages. Broadcasting is supported for only datagram sockets and only on net works such as ethernet, token ring etc but not point to point networks. This option controls whether datagrams may be broadcast from the socket. The value has type `int`; a nonzero value means "yes".

SO_DEBUG :The option is supported by TCP only. When enabled for a TCP socket, the kernal keeps track of all the packets sent or received by TCP for the socket.

SO_DONTROUTE Socket option: The option specifies that outgoing packets are to bypass the normal routing mechanism of the underlying ptotocol. With Ipv4, the packet is directed to the appropriate local interface, as specifried by the network and subnet protions of the destination address. If the local interface cannnot be determined from destination address, ENETUNREACH is returned.

SO_ERROR options: This option can be used with `getsockopt` only. It is used to reset the error status of the socket. When a errors occurs on a socket, the protocol module sets a variable named so_error for that socket to one of the standard unix values. The processs is immediately notified. The process can then obtain the values of so-error by fetching the SO_ERROR socket option. After the receipt, the so_error value is reset to 0 by the kernel.

SO_KEEPALIVE socket option: When the keep alsive socket option is set for a TCP socket, and if no data is exchanged across in either direction for two hours TCP automatically sends a keepalive probe to the peer. The probe is a
TCP segment to which the peer must respond. The possible three scenarios ae ;

- The peer responds with expected ACK. The application is no notified but the TCP sens another probe after 2 hours.
- The peer responds with RST which tells the local TCP that the peer host has crashed and rebooted.. The socket's pending error is set to ECONNRESET and the socket is closed.
- There is no response from the peer. TCP sends eight additional probes, 75 sec apart . If there is no response within 11 min a dn 15 se after first probe, the socket is sent with ETIMEOUT and the socke tis closed.

The purpose of this option is to detect if the peer host crashes. IF the peer host crashes, its TCP will send FIN across the connection which can easily detect with select.
SO_LINGER Socket Option :

The option specifies how the close function operates for a connection oriented protocol (TCP). BY default, close returns immediately, but if there is any data still remaining in the socket send buffer, the system will try to deliver the data top the peer.

The SO_LINGER socket option lest us change this default. This option requires the folowing structure to be passed between the user process and the kernel. It is is defined by including <sys/socket.h>

```
Struct linger {
        Int l_onoff;   /*0=off, nonzero = on */
        Int l_linger ;  /* liner time, posix 1g specifies uits as sec*/
        }
```

Calling setsockopt leads to one of the following three scenarios depending on the values of the two structure.

1. if l_onoff is 0, the option is turned off. The value of l_linger is ignored and the previously discussed TCP defaults applies. Close returns immediately.

2. IF l_onoff is nonzero and l_linger is 0, TCP aborts the connection when it is closed. That is TCP discards any data still remaining in the socket send buffer and an RST to the peer.
3. IF l_onoff is nonzero and l_linger is nonzero, the kernel will linger when the socket is closed. Thst is if ther is any data still remaining in the socket send buffer, the porcess is put to sleep until either a) all the data is send and acknowledged by the peer. B) the linger time expires.

Let us understand when the close on a socket returns, given the various scenarios that we have seen so far.

Assue that the client writes data to the socket and then calls close as shown in the folowing figure. When the clidnt data arrives, the server is temporarily busy, so the data is added to the socket receive buffer by its TCP. Similarly the next segment, the client's FIN is also addded to the socket receive buffer. But by default, the client's close returns immediately.

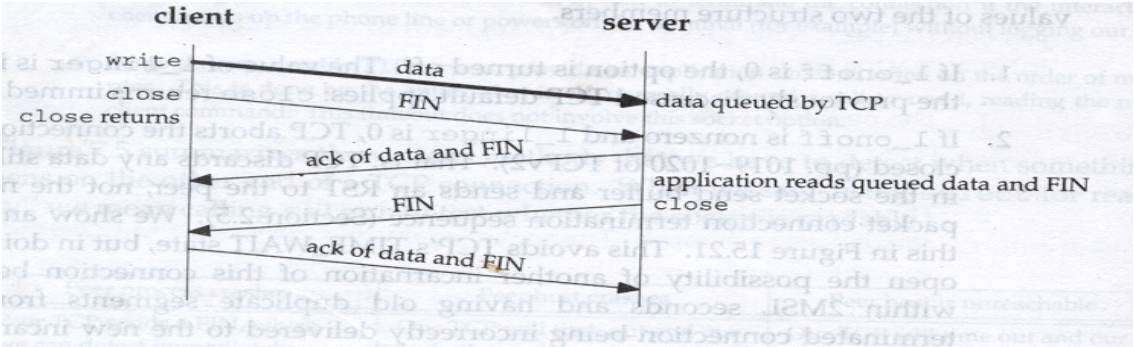


Figure 7.6 Default operation of `close`: it returns immediately.

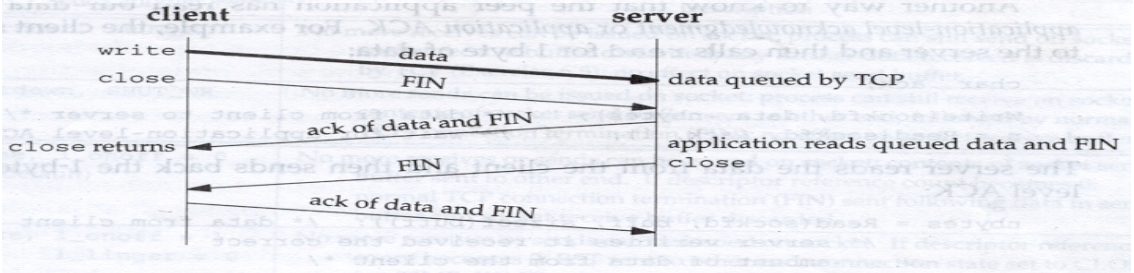


Figure 7.7 `close` with `SO_LINGER` socket option set and `l_linger` a positive value.

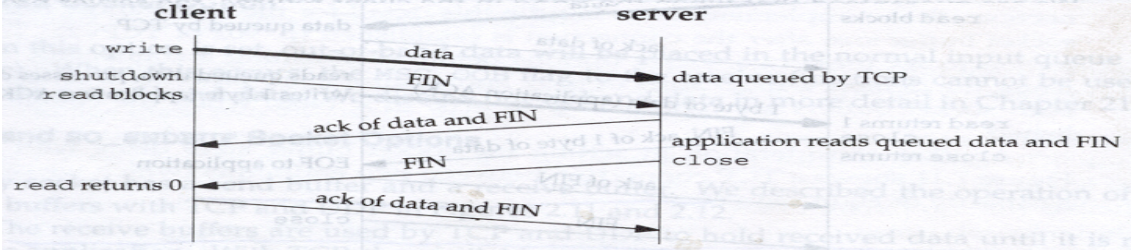


Figure 7.8 Using `shutdown` to know that peer has received our data.

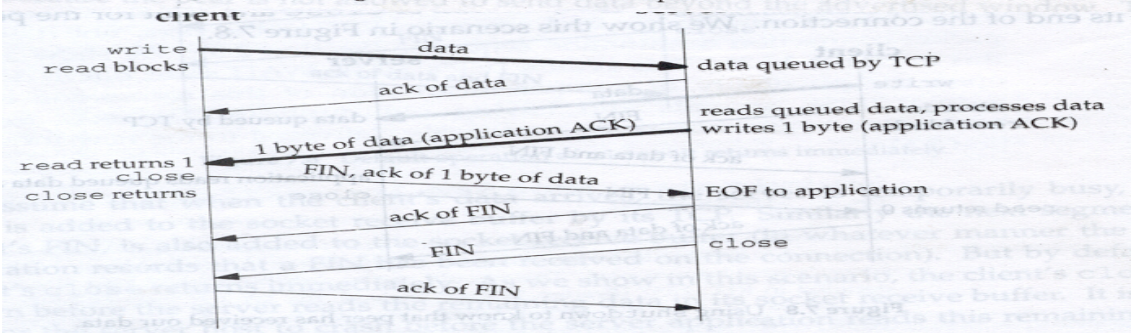


Figure 7.9 Application ACK.

SO-OOBINLINE socket option:

When This option is set, out of band data will be placed in the normal input queue. Whne this occurs, the MSG_OOB flag to the receive functions cannot be used to read the out of band data.

SO_RCVLOWAT and SO_SNDFLOWAT Socket Options:

Every socket has a receive low mark and a send low water mark. These are used in select function. These two socket opitons les us to change these options.

Receive low water mark is the amount of data that must be in the socket receive buffer for select function to be readable. If defalut to 1 for a TCP and UDP sockets. The send low water mark is the amount of available space that must exist in the socket send buffer for a select function to return writable. This low watermarl bnormally defaults to 2048 for TCP sockets.

SO_CVTIMEO AND SO_NDTIMO socket options: These two socket options allow us to place a timeout on socket receives and sends. Notice that the arguments to the two sockopt functionsis a pointer to a timeval structure, the same one used with select. This lets specify the timeout in seconds and microseconds.

The receive timeout afffects the five input functions : read, readv, recv, recvfrom and recvmsg. The send timeout affects the five output functions : write, writev,send, sendto, and sendmsg.

16.12.2 Socket-Level Options

— Constant: int **SOL_SOCKET**

Use this constant as the *level* argument to `getsockopt` or `setsockopt` to manipulate the socket-level options described in this section.

Here is a table of socket-level option names; all are defined in the header file `sys/socket.h`.

`SO_REUSEADDR`

This option controls whether `bind` (see Setting Address) should permit reuse of local addresses for this socket. If you enable this option, you can actually have two sockets with the same Internet port number; but the system won't allow you to use the two identically-named sockets in a way that would confuse the Internet. The reason for this option is that some higher-level Internet protocols, including FTP, require you to keep reusing the same port number.

The value has type `int`; a nonzero value means “yes”.

`SO_KEEPALIVE`

This option controls whether the underlying protocol should periodically transmit messages on a connected socket. If the peer fails to respond to these messages, the connection is considered broken. The value has type `int`; a nonzero value means “yes”.

`SO_DONTROUTE`

This option controls whether outgoing messages bypass the normal message routing facilities. If set, messages are sent directly to the network interface instead. The value has type `int`; a nonzero value means “yes”.

`SO_LINGER`

This option specifies what should happen when the socket of a type that promises reliable delivery still has untransmitted messages when it is closed; see Closing a Socket. The value has type `struct linger`.

— Data Type: **struct linger**

This structure type has the following members:

`int l_onoff`
This field is interpreted as a boolean. If nonzero, `close` blocks until the data are transmitted or the timeout period has expired.
`int l_linger`
This specifies the timeout period, in seconds.

`SO_BROADCAST`
This option controls whether datagrams may be broadcast from the socket. The value has type `int`; a nonzero value means “yes”.
`SO_OOBINLINE`
If this option is set, out-of-band data received on the socket is placed in the normal input queue. This permits it to be read using `read` or `recv` without specifying the `MSG_OOB` flag. See Out-of-Band Data. The value has type `int`; a nonzero value means “yes”.
`SO_SNDBUF`
This option gets or sets the size of the output buffer. The value is a `size_t`, which is the size in bytes.
`SO_RCVBUF`
This option gets or sets the size of the input buffer. The value is a `size_t`, which is the size in bytes.
`SO_STYLE`
`SO_TYPE`
This option can be used with `getsockopt` only. It is used to get the socket's communication style. `SO_TYPE` is the historical name, and `SO_STYLE` is the preferred name in GNU. The value has type `int` and its value designates a communication style; see Communication Styles.
`SO_ERROR`

This option can be used with `getsockopt` only. It is used to reset the error status of the socket. The value is an `int`, which represents the previous error status.

http://www.unet.univie.ac.at/aix/aixprggd/progcomc/toc.htm

SO_RCVBUF /SO_SNDBUF
• Integer values options - change the receive and send buffer sizes.
• Can be used with STREAM and DGRAM sockets.
• With TCP, this option effects the window size used for flow control – must be established before connection is made.
`SO_REUSEADDR`
Boolean option: enables binding to an address (port) that is already in use.
• Used by servers that are transient - allows binding a passive socket to a port currently in use (with active sockets) by other processes.
Can be used to establish separate servers for the same service on different interfaces (or different IP addresses on the same interface).
• Virtual Web Servers can work this way.
KEEPALIVE socket Option: The purpose of this option is to detect if the peer host crashes. This option is usually used by servers, although client can also use this option. Server uses this option because they spend most of their time blocked wiating for the input across the TCP connection, that is waiting for the client request. But if the client host crashes, the server process will never know about it and it will wiat continuosly for input data that can never arrriver. This is called half open connection. The keep alive option will detect these half open connections and terminates them.

3. If there is no response from the peer to the keepalive probe, TCP sends eight additional probes, 75 sec apart trying to elicit response. TCP will give up if ther is no response within 11 minutes and 15 seconds after sending the first probe. If there is no response at all to TCPs keepalive probes, the socket's pending error is sent to ETIMEDOUT and the socket is closed. But if the socket receives an ICMP response to one of the keepalive probes, the socket corresponding error is returned instead such as EHOSTUNREACH error.

SO_LINGER Socket option: This option specifies how the close function operates for a connection oriented protocol (that is for TCP) By default, close function returns immediately with ack, but if there is any data still remaining the in the socket sedn buffer, the system will try to deliver the data to the peer.

The SO_LINGER socket option les us change this default. This option requires the folowing structure to be passed between the user process and the kernel. It is defined as

```
Struct linger {
        Int l_onoff; /* 0 for Off and Nonzero for ON*/
        Int l_linger;  /*Linger time Posix 1g specification as seconds*/
        }
```

Calling setsocket leads to one of the folowing threee scenarios depending on the values of the two structure memebrs.

1. If l_onoff is 0, the option is turned off. The value of l_linger is ignored and the preveiously discussed TCP default applies: close returns immediately.
2. If l_onoff is non zero and l_linger is 0, TCP aborts the connection when it is closed. That is TCP discards any data still remaining in the socket send buffer amd sends an RST to the peer. (Not the four step termination sequence) This avoids the TCP's TIME_WAIT state.
3. If l_onoff is nonzero and l_linger is non zero, then the kernel will linger when the socket is closed . That is, if ther is any bdata stilol remaining in the socket send buffer, the process is put to sleep until either i) all the data is sent and acknolodged by the peer TCP ii) the linger time expires

   When using this feature of the SO-LINGER option, it is important for the application to check the return value from close because if the linger time expires before the remianing data is sent and acknowledged, claose returns EWOULDBLOCK and any remaining data in the send buffer is discarded.

SO_RCVBUF and SO_SNDBUF Socket Option:

The receive buffer are used by the TCP and UDP to hold received data until it is read by the application. With TCP, the available room in the socket receive buffer is the window that TCP avertises to the other end. Hence the the peer sends only that amount of data and any data beyond that limit is discarded. In case of UDP, the bufffer size is not advertised hence, any data that do not fit ito the buffer, are dropped However, the abovementioned socket options allows one to change the default sizes. The default values for the TCP and UDP differ for different implementation. It is normally 4096 for TCP and send buffer for UDP is 9000 and 40000 bytes for receive buffer.

SO_REUSEADDR and **SO_REUSEPORT:**

The SO_REUSEADDR serves four purposes :

This option allows a listening server to start and bind its well known port even if previously established connections exists that use this port as their local port. As the server is in listening state,

when connection request comes from a client, a child process is spawned to handle that client. Wqith this listening server terminates. Once again when the listening is restarted buy calling socket, bind and listen, the call to bind fails because the listening server is trying to bind a port that is part of exisitng connection. But if the srver is sets the SO_REUSEADDR socket option between the calls to socket and bind, the latter function will succeed.

This allows multiple instances of the same service to be started on the same port as long as each instance binds a different local IP address. It is common of a a site to host multiple http servers using the IP alias techniques. If the primary address is 198.69.10.2 and it has two aliases as 198.69.10.129 and 198.69.10.128. Three HTTP servers are started. Whne the first connection request comes, the server binds the call with 198.69,10.128 and a local port of 80. The second request is connected to 198.69.10.129 provided the SO_REUSEADDR is set before the call to bind.
Simlarly for the final http server also.

It also allows a single process to bind the same port to multiple socket as slong as each bind specifies a different loacl IP address It also allows completely duplicate binding: a bind of an IP address and port, when the same IP address and port are already bound to another socket. This happens with the protocol that support multicasting (UDP)

SO_ TYPE socket Option: This option returs the socket ytpe. The integer value returned is a value such as SOCK _ STREAM or SOCK_DGRAM.

SO_USELOOPBACK Socket Option: When this option is set, the socket receives a copy of everything send on the socket.

IPv4 Socket options: The level of this options are IPPROTO_IP.

IP_HDRINC Socket Option : If this socket is set for a raw socket, we must buid our own IP header for all datagrams that we send on the raw socket. Normally kernel builds the headers for datagrams sent on raw socket. But for some applications, build there own IP address to override that IP would place into certain header fields. (Traceroute).

IP_OPTIONS Socket Options: Setting this option allows us to set the IP option in the Ipv4 header. This requires intimate knlowledge of the format of the IP options in the IP header.

IP_RECVDSTADDR Socket Options :
This socket options causes the destination IP address of a received UDP datqgram to be returned as ancillary data by recvmsg.

IP_RECVIF Socket Options :
This socket option causes the index of the interface on which a UDP datagram is received to be returned as an ancillary data by recvmsg.

IP_TOS Socket Option :
This options lets us set the type of field service in the IP header for a t
CP or UDP socket. The different value to which this o[tions can be set are given below:

Constant	description

IPTOS_LOWDELAY	minimize delay.
IPTOS_THROUGHPUT	maximize throughput
IPTOSRELIABILITY	maximize reliability.
IPTOS_LOWCOST	Minimize cost.

For telnet login shoul;d specify IPTOS_LOWDEAY while the data portion of an FTP transfer should specify IPTOS_ THROUGHPUT

IP_TTL Socket Option :

With this option we can set and fetch the default TTL (time to live field) that the system will use for a given socket (*64 for TCP and 255 for UDP)

ICMPv6 Socket Option : The level is of **IPPROTO_ICMPV6**

ICMP6_FILTER: This option lets us fetch and set an icmp6_filter structure that specifies which of the 255 possible ICMPV6 message types are passed to the process on a raw socket.

IPv6 Socket Options:\

These are processed by IPv6 and have a level of IPPROTO_IPV6.

IPV6_ADDRFORM Socket Option:

This option allows a socket to be converted from IPv4 to IPv6 or vice versa.

IPV6_CHECKSUM Socket Option: This socket option specifies the byte offset into the user data of where the checksum field is located. If this value is nonnegative, the kernel computes and store the checksum for all outgoing pockets. And verify the received checksum on input, discarding packets with invalid checksum. If the value is set to -1, the kernel will calculate and store the checksum for outgoing packets.

IPV6_DSTOPTS Options : This options lets any received IPv6 destination options are to be returned as ancillary data by ***recvmsg.***

IPV6_HOPOPTS: Setting this option specifies that the received IPv6 hop by hop options are to be returned as ancillary data by recvmsg.

IPV6_HOPLIMIT : Setting this option specifies that the received hop limit field be returned as ancillary data by **recvmsg.**

IPV6_NEXTMSG : This is a not a socket option but the type of an ancillary data object that can be specified to ***sendmsg.*** This opjet specifies the next hop address for a datagram as a socketaddress structure.

IPV6 _PKTINFO: Setting this option specifies that the following two pieces of information about a received IPv^ datagram ae to be returned as ancillary data by ***recvmsg.***

IPV6_PKTOPTIONS: Most of the IPv6 socket options assume a UDP socket with the information being passed between the kernel and the application using ancillary data with ***recvmsg and sendmsg.*** A TCP socket fetches and stores these values using the IPV6_PLTOPTIONS socket options.

IPV6_RTHDR: Setting this options specifies that a received IPV6 routing header is to be returned as ancillary data by ***recvmsg.***

IPV6_UNICAST_HOPS : This IPv6 option is similar to the IPv4 IP_TTL socket option. Setting the socket option specifies the default hop limit for outgoing datagram sent on the socket, whcile fetching the socket options returns the value for the hop limit that the kernel use fo the socket. To obtain actual hop limit field, IPV^_HOPLIMIT socket option is used.

TCP SOCKET OPTIONS: The leve is IPPROTO_TCP
TCP_KEEPALIVE socket options : It specifies the idle time in seconds for the connections before TCP starts sending keepalive probes. Default value is 7200 sec (2 hours) This is effective when SO_ ***KEEP ALIVE option is enabled.***

TCP_MAXRT Socket Options : It specifies the amount of time in seconds before a connections is broken once TCP starts transmitting data. A value of 0 means touse the system default. And a value of –I means to rettransmit forever. If a positive value is specified, it may be rounded up to the implementations next transmission time.

TCP_MAXSIZE Socket Option: This socket option allows us to fetch or set the maximum segment size for a TCP connection. Often it is the MSS value announced by the peer process.

TCP_NODELAY Socket Option :
If this option is set, TCP's Nagle's algorithm is disabled.
Nagle's Algorithm: The algorithm states that if a given connection has outstanding data (that is, data that our TCP has sent and for which it is currently awaiting an ack), then no small packet will be sent on the connection until the existing data is acknowledged. Small packet is any size less than MSS. The purpose is to prevent a connection from having multiple packets outstanding at any time. This situation becomes more prominenet in a WAN with Telnet.

Consider the example where in we type Hello to Telnet client. Let this take 250 ms between each letter (as shown below) The RTT to the server and back is 600 ms and the server immediately sends the echo of the character. We assume that ACK of the client character is sent back to the client along with the character echo and we ignore the ACKs that the client sends for the server echo. Assuming Nagle's algorithm disabled, we have the following figure:

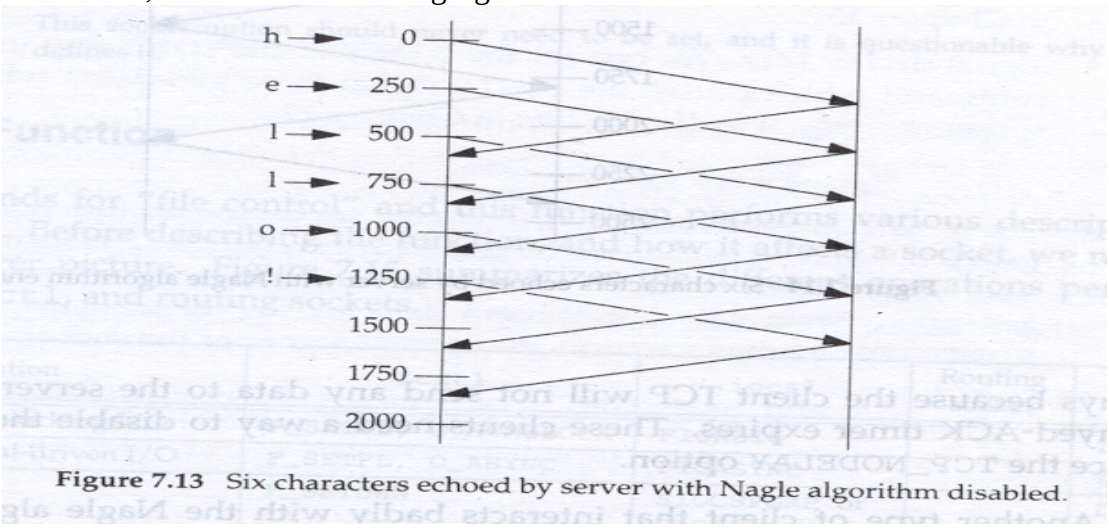


Figure 7.13 Six characters echoed by server with Nagle algorithm disabled.

IN this each packet is sent in a packet itself.

But if the Nagle's algorithm is enabled (default), we have the six packets as shown in the following figure:

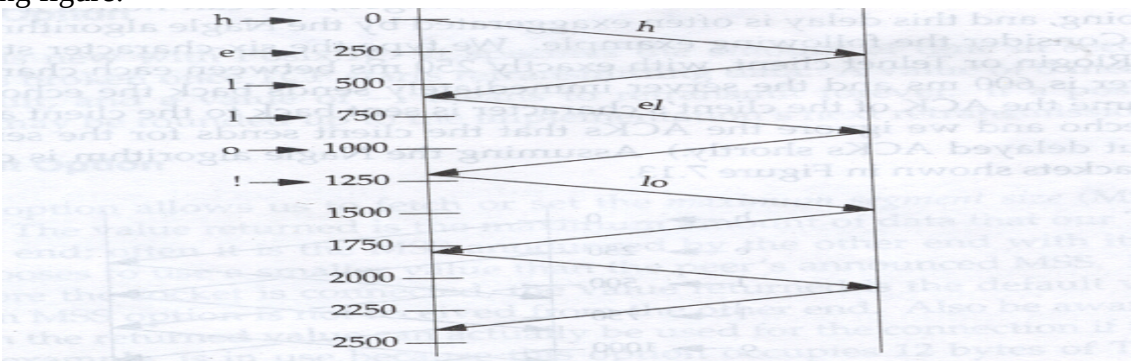


Figure 7.14 Six characters echoed by server with Nagle algorithm enabled.

The first packet is sent as a packet by itself., but the next two characters are not sent, since the connection has small packet outstanding. At time 600 ms when the ACK of the first packet is received, along with the echo of the first character, theset two characters are sent. Until this packet ACKed at time 1200, no more small packets are sent.

When the Nagles algorithm oftern interacts with another TCP algorithm called 'delayed ACK' algorithm. This algorithm causes the TCP to not send an ACK immediately whne it receives data; instead TCP will wait some small amount of time and only then send ACK. The hope is that in this small time (50 – 200 ms) there will be more data to be sent back to the peer and the ACK can piggy back on the data saving the TCP segment. This is the normally case with Telnet. Therefore, they wait for the echoed data to piggy back.

In all these cases, the TCP_NODELAY socket option is set.

TCP_STDURG socket option: If this is set, then the the urgent pointer will point to the data byte sent with the MSG_OOB flag

fcntrl Function : This stands for file control. This control performs various deascriptors control operations. Following table summarised some of the key file operations. These are preferred way under Posix 1g.

Opeations	Fcntl
Set socket for non blocking I/O	F-SETFL, O_NONBLOCK
Set socket for signal driven I/O	F_SETFL, O_ASYNC
Set socket owner	F-SETOWN
Get socket owner	F-GETOWN

These features are provided in the following way.

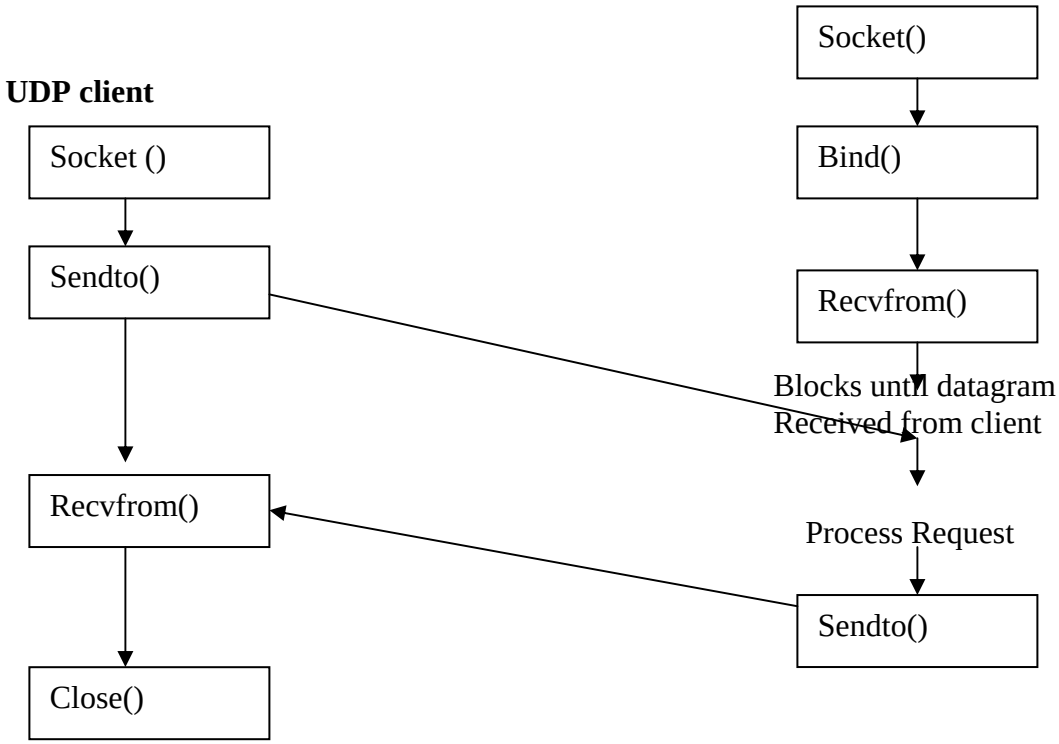
- Non blocking I/O: We can set the O_NONBLOCK file status flag using the F_SETFL command to set a socket nonblocking.
- Signal Driven I/O : We can set the O_SYNC file status by using F-SETFL command which caused the SIGIO signal to be generated when the status of a socket changes.
- The F_SETDOWN command lets us set the socket owner to receive the SIGIO and SIGURG signals. SIGIO is generated when the signal driven I/O is enabled for a socket and the latter is generated with new out of band data arrives for a socket. The F-GETOWN command returns the current owner of the the socket.

#include <fcntl.h>

int fcntl (int fd, int cmd, …) returns: depends on cmd if OK, -1 on error.

ELEMENTS OF UDP SOCKET

The significant difference between TCP and UDP applications are in the Transport layer. UDP is connectionless, unreliable, datagram protocol. However, there are needs for such requirements in applications such as DNS, NFS and SNMP. Typical functions calls of UDP client server are shown below:



IN this, client does not establishes connection with server, rather it sends datagram using send to function along with destination address. Similarly, the server does not accept connection from a client, instead the server just calls recvfrom function which waits until data arrives from some client. Recvfrom returns the protocol address of the client along with the datagram so the server can send a response to the correct client.

Recvfrom and sendto functions :

#include <sys/socket .h>

ssize_t recvfrom (int sockfd, void *buff, size_t nbytes, int flags, struct sockaddr *from, socklen_t *addrlen)

ssize_t sendto (int sockfd, const void *buff, size_t nbytes, int flags, struct const sockaddr *to, socklen_t *addrlen)

sockfd, buff and nbytes are identical to the first three arguments for read and write: descriptors, pointer to buffer to read into or write from, and number of bytes to read or write.

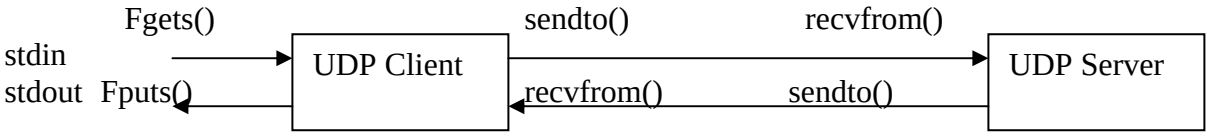
The flags are meant are normally used with recvmsg and sendmsg functions. IN this function they are defaulted to the value of 0. The to argument for the sendto is a socket address structure containing protocol address (IP and port) of where data is to be sent. The recvfrom functions fills in the socket address structure pointed to by from with the protocol address of who sent the datagram. The number of bytes stored in this socket address structure is also returned to the caller in the integer pointed to by the addlen.

Final argument to sendto is an integer value, while the final argument to recvfrom is a pointer to an integer value (a value result - argument.)

The final two argument of recvfrom is similar to the two argument to accept : the contents of the socket address structure upon return tell us who sent the datagram (in case of UDP)or who initiated the connection in the case of TCP. Both function return the length of the data that was read or written as the value of the funtion.

In the case of writing a datagram of length 0 is OK. IN UDP, this means, 20 byte length of IP header of IPV4, 8 byte UDP header and no data. It is accepted unlike TCP where in a 0 is consider as EOF.

UDP Echo Server : Main function: The function call of UDP client and server is shown in the following figure.



The main server program is shown below:

```
#include    "unp.h"

int
main(int argc, char **argv)
{
      int                           sockfd;
      struct sockaddr_in      servaddr, cliaddr;

      sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

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

      Bind(sockfd, (SA *) &servaddr, sizeof(servaddr));

      dg_echo(sockfd, (SA *) &cliaddr, sizeof(cliaddr));
}
```

IN this UDP socket is created by giving SOCK_DGRAM . The address for the bind is given as INADDR_ANY for multihomed server. And the const SERVER_PORT is the well known port.

UDP Echo Server : dg_echo function: The programme is given below:

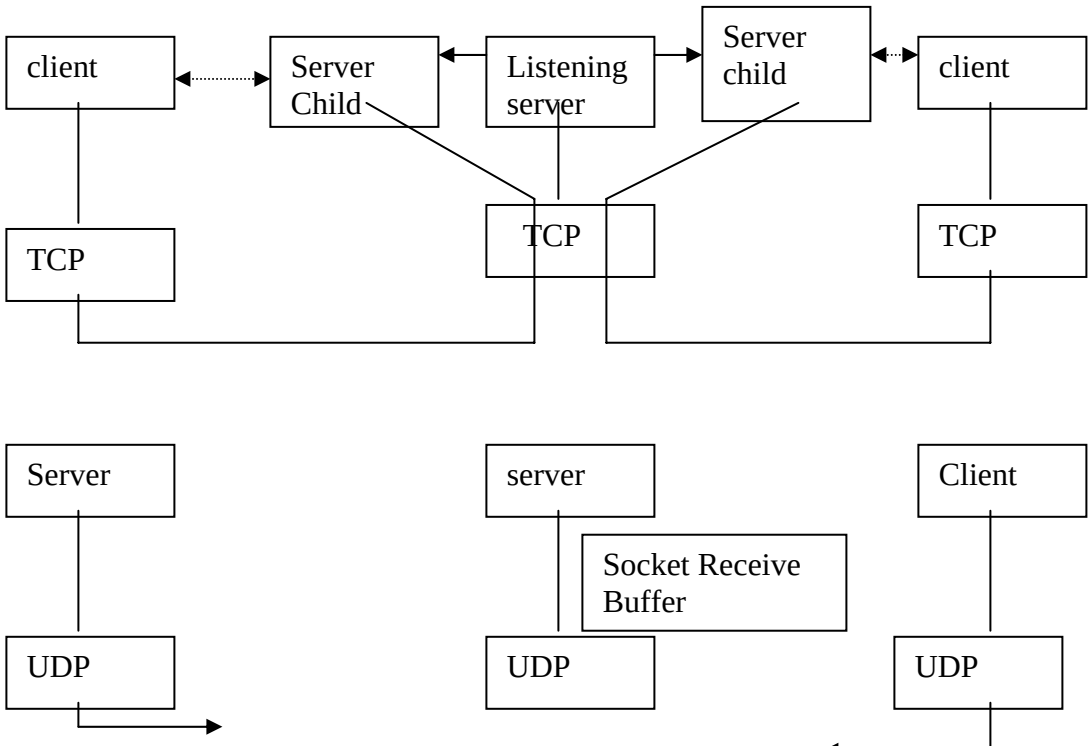
```
 void
dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
{
      int n;
      socklen_t   len;
      char        mesg[MAXLINE];

            for ( ; ; ) {
            len = clilen;
            n = recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);
            sendto (sockfd, mesg,n,0,pcliaddra, len);

      }
}
```

The important details to consider are:

1. The function never terminates (exit (0) is not called) as it is connection less protocol
2. The main function is iterative server not concurrent. There is no call tofokr, so a single server process handles any and all clients.
3. There is implied queuing takes place in the UDP layer for the socket. Each datagram that is arrived is received in a buffer from where the recvfrom receives the next datagram in FIFO manner. The size of the buffer may be changed by changing the value of SO_RCVBUF



The main function in figure is the protocol dependent (it creates a socket of protocol AF_INET) and allocates and initializes an IPv4 socket address structure. But the dg_echo function is protocol independent as it is provided with the socket address structure and its length by the main function. It is the recvfrom that fills in this structure with the IP address and prot number of the client and as the same pointer is passed to sendto as the destination address.

UDP Echo Client :

udpcliserv/ udpcli01.c Page No: 216

```
#include        "unp.h"

int
main(int argc, char **argv)
{
        int                                     sockfd;
        struct sockaddr_in      servaddr;
        if (argc != 2)
                err_quit("usage: udpcli <IPaddress>");
        bzero(&servaddr, sizeof(servaddr));
        servaddr.sin_family = AF_INET;
        servaddr.sin_port = htons(SERV_PORT);
        Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
        sockfd = Socket(AF_INET, SOCK_DGRAM, 0);
        dg_cli(stdin, sockfd, (SA *) &servaddr, sizeof(servaddr));
        exit(0);
}
```

An IPv4 socket address structure is filled in with the IP address and port numbers of the server. This structure is passed on to dg_cli.
A UDP socket is is created and the function calls dg_cli.

Dg_cli Function

```
#include        "unp.h"

void dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
{
        int                 n;
        char    sendline[MAXLINE], recvline[MAXLINE + 1];

        while (Fgets(sendline, MAXLINE, fp) != NULL) {

                sendto (sockfd, sendline, strlen(sendline, 0, pservaddr, servlen);

                n = recvfrom(sockfd, recvline, MAXLINE,0,NULL,NULL);

                recvline[n] = 0;        /* null terminate */
                Fputs(recvline, stdout);
        }
}
```

There are four steps in the client processing loop. Read a line from the standard input using fgets, send the line to the server using sendto, read back the server's echo using recvfrom and print the echoed line to standard output using fputs

With a UDP socket, when the first time the call to sendto() is given, the port is bound to it. Also similar to server, client can also call for bind() to tie up its port.

In the recvfrom(), a null pointer is specified in the fifth and sixth argument. Which means that we are not interest in knowing who has send the datagram. This may result in error in reception of datagram, as the same host or any other host may send datagram which is likely to be mistaken.

Similar to dg_echo(), dg_cli() is also protocol independent. In this the main function allocates and initializes a socket address structure of some protocol type and then passes pointer to this structure along with its size.

Lost Datagrams:

Above client server example is not reliable. If the client datagram is lost, (may be dropped at any router), the client will block its recvfrom() forever waiting for a server reply that will never arrive. Similarly if the client datagram arrives at the server but the server's reply is lost, the client will once again block the recvfrom(). A time out and a appended number is normally sent with the datagram which is sent back as acknowledgment with reply. This will assure whether the datagram is received or not. This facility is provided many UDP servers.

Verifying the Received Response : It is seen from the above example that any process that knows the clients ephemeral port number could datagram to our client and these will be interpreted with the normal server replies. It can be avoided by comparing the IP address and port number of the recvfrom() (reply received from server), with that of the IP address and port number to which it was sent earlier. This requires creating a socket address structure in which the return address structure is returned and compared with the sent address structure. This is shown in the following example

udpcliserv/dgcliaddr.c Page No : 219

```
dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
{
        int                     n;
        char                    sendline[MAXLINE], recvline[MAXLINE + 1];
        socklen_t               len;
        struct sockaddr         *preply_addr;
        prpeply_addr = malloc(servlen);
        while (Fgets(sendline, MAXLINE, fp) != NULL) {
        Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
        Len = servlen;
        n = Recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
        if (len != servlen|| memcmp (pservaddr, preply_addr, len)!= 0)
        {printf (reply from %s (ignored)\n“),sock_ntop (preply_addr, len);
          continue;
        }
        Recvline[n] = 0;        /* null terminate */
        Fputs(recvline, stdout);
        }
}
```

First the, client main function is changed to use the standard echo server. **Servaddr.sin_port = htons(SERV_PORT)** changed to **servaddr.sin_port = htons (7).** Malloc() function allocates the byte size to the structure. With this arrangement, the received address captured in preply_addr is compared with the one that was sent in sendto(). First a comparison is made of its length and then memcopy compares then socket address itself is compared. This will assure if the server address is same.

However, even this may not work if the server is multi homed where in it is likely to allocate different IP address as the server has not bound an IP address to its socket and it chooses any IP Address. Hence this may also fail.

One solution is for the client to verify the responding host's domain name instead of its IP address by looking up the server's name in the DNS, given the IP address returned by recvfrom().

Another solution is for the server to create one socket for every IP address that is configured on the host , bind that IP address to the socket, use select across all these sockets (waiting for one to be readable) and then reply from the socket that is readable. Since the socket used for the reply was bound to the IP address that was the destination address of the client request, this guarantees that the source address for the reply is the same as the destination address of the request.

Server not Running :

If the client sends a datagram to a server that is not yet started, what happens to it? The recvfrom() will be blocked for ever waiting for a reply from the server. When the call to sendto() is given by passing the massage to it, the function returns OK (size of message to application) which means that datagram is put in the output interface queue for further transmission. Where as the datagram traveling to server that is not switched on, is responded with an ICMP port unreachable error. As this error takes more time to return (equal to RTT), this error is known as asynchronous error - not synchronous with the error causing source, which in this case is sendto().

This error is not returned to UDP socket at the client. Hence, the recvfrom() is left open. The asynchronous error can be returned only when the socket have been connected which requires a call to connect() by the client.

connect() with UDP:

As it was seen that asynchronous errors are not returned to UDP sockets unless the socket has been connected. It is possible to give a call to connect() in UDP also. But it does involved few changes that are listed below:

1. When connect() is called, the destination IP address and port number of the server host is provided. Hence there is no requirement to pass these values in sendto(). Hence, the functions sendto() is replaced with write() or send() functions. This assumes that anything written is automatically send to the server address specified in the connect function.

2. Similarly, recvfrom() is also not used instead recv() or read() functions are used. The only datagram returned by the kernel for an input operation on a connected UDP socket are those arriving from protocol address specified in the connect. Datagram destined to the connected UDP sockets local protocol address but arriving from a protocol address other than the one to which the socket was connected, are not passed to the connected socket. This limits the connected UDP socket to exchanging datagram with one and only peer.

3 Asynchronous errors are returned to the process for a connected UDP socket. Also an unconnected UDP sockets does not receive any asynchronous error.

It can be said that UDP client or server can call connect() only if that process uses UDP socket to communicate with exactly one peer.

Calling connect () Multiple Times for a UDP socket:

A process with a connected UDP socket can call connect() function again for that socket to either

- Specify a new IP address and p ort or to,
- Unconnect the socket.

Unlike in the case of TCP, where in the connect() is called only once, in the case of UDP, it can be called again.

To unconnect, the connect is called by setting the family members of the socket address structure (sin_family for IPv4 and sin6_family for IPv6) to AF_UNSPEC. This might return EAFNOSUPPORT error but it is accepted.

Performance:

When an application calls sendto() on an unconnected UDP socket, Berkeley derived kernels temporarily connect the socket, and datagram and then un connect the socket . Calling sendto for two datagram on unconnected UDP socket then involves the following six steps:

- Connect the socket.
- Output the first datagram.
- Unconnect the socket.
- Connect the socket.
- Output the second datagram
- Unconnect the socket.

When the application knows that it will be sending multiple datagram to the same peer, it is more efficient to connect the socket explicitly by calling connect() function and then call write() function as many times. As shown below:

- Connect the socket/
- Output the first datagram and
- Output the second datagram.

IN this case the kernel copies only the socket address structure containing the destination IP address and port one time, versus two times when sendto is called twice.

The dg_cli function with call to connect function is given below:

```
#include        "unp.h"

void
dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
{       int                 n;
        char    sendline[MAXLINE], recvline[MAXLINE + 1];
        Connect(sockfd, (SA *) pservaddr, servlen);
        while (Fgets(sendline, MAXLINE, fp) != NULL) {
                Write(sockfd, sendline, strlen(sendline));
```

```
            n = Read(sockfd, recvline, MAXLINE);
            recvline[n] = 0;        /* null terminate */
            Fputs(recvline, stdout);
        } }
```

The changes are the new call to connect and replacing the calls to sendto() and recvfrom() with calls to write() and read(). This functions are still protocol independent.

TCP and UDP Echo Server using Select ():

Following example combines the concurrent TCP echo server with iterative UDP echo server into a single server using select function to multiplex the TCP and UDP socket.

```
/* include udpservselect01 */
#include    "unp.h"
int
main(int argc, char **argv)
{
    int                 listenfd, connfd, udpfd, nready, maxfdp1;
    char                mesg[MAXLINE];
    pid_t               childpid;
    fd_set              rset;
    ssize_t             n;
    socklen_t           len;
    const int           on = 1;
    struct sockaddr_in  cliaddr, servaddr;
    void                sig_chld(int);
    /* 4create listening TCP socket */
    listenfd = Socket(AF_INET, SOCK_STREAM, 0);
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family      = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port        = htons(SERV_PORT);
    Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
    Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
    Listen(listenfd, LISTENQ);
        /* 4create UDP socket */
    udpfd = Socket(AF_INET, SOCK_DGRAM, 0);
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family      = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port        = htons(SERV_PORT);
    Bind(udpfd, (SA *) &servaddr, sizeof(servaddr));
/* end udpservselect01 */
/* include udpservselect02 */
    Signal(SIGCHLD, sig_chld); /* must call waitpid() */
    FD_ZERO(&rset);
    maxfdp1 = max(listenfd, udpfd) + 1;
    for ( ; ; ) {
        FD_SET(listenfd, &rset);
        FD_SET(udpfd, &rset);
        if ( (nready = select(maxfdp1, &rset, NULL, NULL, NULL)) < 0) {
            if (errno == EINTR)
```

```
                        continue;               /* back to for() */
                else
                        err_sys("select error");
        }
        if (FD_ISSET(listenfd, &rset)) {
                len = sizeof(cliaddr);
                connfd = Accept(listenfd, (SA *) &cliaddr, &len);
                        if ( (childpid = Fork()) == 0) {        /* child process */
                        Close(listenfd);        /* close listening socket */
                        str_echo(connfd);       /* process the request */
                        exit(0);
                }
                Close(connfd);                  /* parent closes connected socket */
        }
        if (FD_ISSET(udpfd, &rset)) {
                len = sizeof(cliaddr);
                n = Recvfrom(udpfd, mesg, MAXLINE, 0, (SA *) &cliaddr, &len);
                Sendto(udpfd, mesg, n, 0, (SA *) &cliaddr, len);
        } }}
/* end udpservselect02 */
```

Create listening TCP socket

A listening TCP socket is created that is bound to the server's well known port. We set the SO_REUSEADDR socket option in case of connections exist on this port.

Create a UDP socket

A UDP socket is also created and bound to the same port. Even though the same port is used for the TCP and UDP sockets, there is no need to set the SO_REUSEADDR socket option before this call to bind because TCP ports are independent of UDP ports.

Establish a signal handler for SIGCHLD:

Estabilised the signal handler SIGCHLD because TCP connections will be handled by a child process.

Prepare for Select:

A descriptor set is initialized for select and maximum of two descriptors for which the select waits.

Call select:

We call select waiting only for readability on the listening TCP socket or readability on the UDP socket. Since our sig_chld handler can interrupt our call to select, we handle an error of EINTR

Hane the new client:

We accept a new client connection when the listening TCP socket is readable, fork a child and call our str_echo in the child.

Handle arrival of datagram.

If the UDP socket is readable, a datagram has arrived. We read it with recvfrom and send it back to the client with sendto().

Summary:

Converting echo-client server to use UDP instead of TCP was simple. But the features provided by TCP are missing: detecting lost packet and retransmitting, verifying responses and so on.

UDP socket can generate asynchronous errors that is errors that are reported some time after the packet was sent. IN TCP, these error are always reported to application but not in UDP

UDP has no flow control. But this is not a big restriction as the UDP requirement are built for request – response application.

gethostname () :

I n the application, as it is easy to input human readable DNS name instead of IP address, there is a need to convert the host names into IP address format. This is achieved by the function **gethostname ().** When called, if successful, it returns a pointer to a **hostent structure** that contains all the **IPv4** address or all **IPv6** address for the host.

The syntax is given below:

#include <netdb.h>

struct hostent *gethostbyname (const char * hostname);

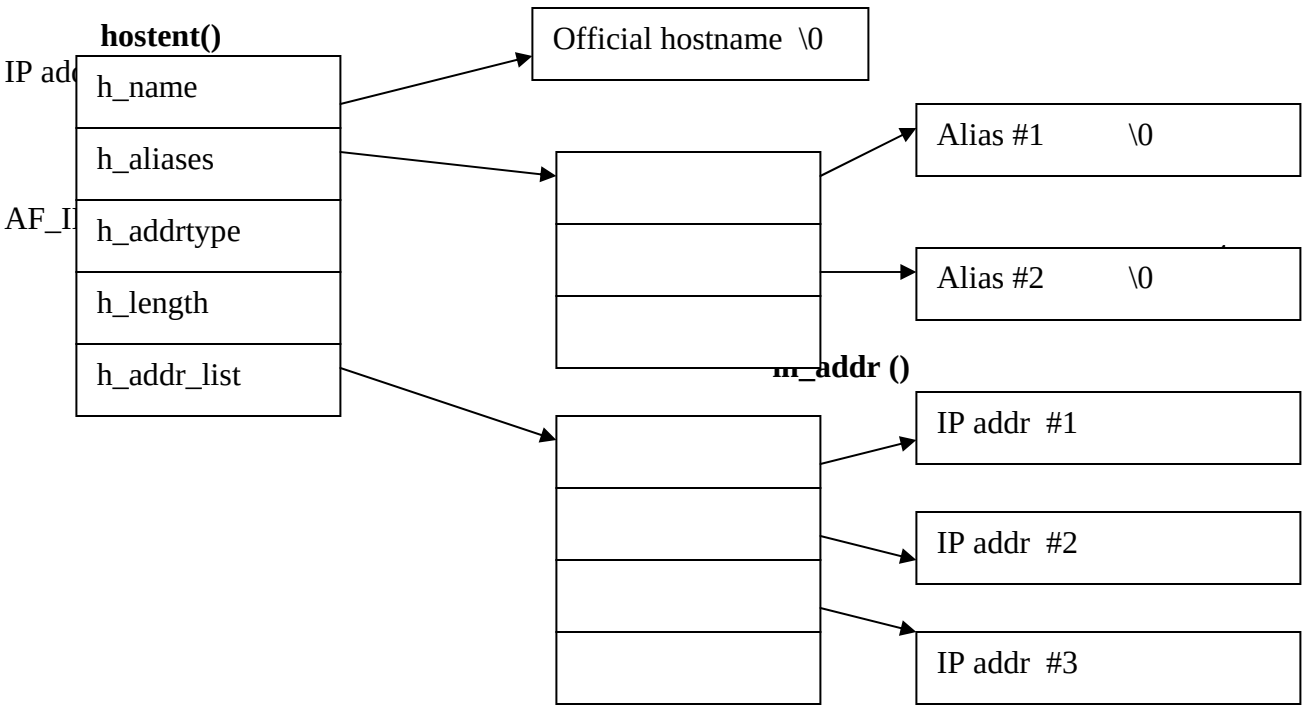
returns nonnull pointer if OK , NULL on error with h_errorno set.

The non null pointer returned by this function points to the following **hostent** structure:

```
Struct hostent {
        char *h_name;  /* official canonical) name of host*/
        char  **h_aliases ;     /* pointer to array of pointers to alias names*/
        int       h_addrtype ;  /* host address type: AF_INET or AF_INET6 */
        int       h_length ;    /* length of address : 4  or 16*/
        char ** h_addr_list     /* ptr to array of ptrs with IPv4 or IPv6 addrs*/
        }
#define h_addr  h_addr_list[0]          /* first address in list*/
```

In terms of DNS, gethostbyname() performs a querry for an A record or for a AAAA record. This function can return either IPv4 or IPv6 address.

Arrangement of hostent structure is shown below:



Similar structure is ther6 for IPv6 wherein h_addrtype is AF_INET6 and h_length is equal 6.

Returned h_name is called canonical name of the host. When error occurs, the functions sets the global integer h_errono to one of the following constants defined by including <netdb.h>.

- HOST_NOT_FOUND
- TRY_AGAIN
- NO_RECOVERY
- NO_DATA (same as N0_ADDRESS)

NO_DATA error is valid and it indicates that the hostent has only MX record

Example:

```
#include        "unp.h"
int  main(int argc, char **argv)
{
        char                    *ptr, **pptr;
        char                    str[INET6_ADDRSTRLEN]; /* handles longest IPv6 address */
        struct hostent   *hptr;

        while (--argc > 0) {
                ptr = *++argv;
                if ( (hptr = gethostbyname(ptr)) == NULL) {
                        err_msg("gethostbyname error for host: %s: %s",
                                        ptr, hstrerror(h_errno));
                        continue;
                }
                printf("official hostname: %s\n", hptr->h_name);

                for (pptr = hptr->h_aliases; *pptr != NULL; pptr++)
                        printf("\talias: %s\n", *pptr);

                switch (hptr->h_addrtype) {
                case AF_INET:
#ifdef  AF_INET6
                case AF_INET6:
#endif
                        pptr = hptr->h_addr_list;
                        for ( ; *pptr != NULL; pptr++)
                                printf("\taddress: %s\n",
                                        Inet_ntop(hptr->h_addrtype, *pptr, str, sizeof(str)));
                        break;
                default:
                        err_ret("unknown address type");
                        break;
                }
        }
```

```
        exit(0);
}
```

IN this gethostname() is called for each command line argument. The official hostname output followed by the list of alias names. This program supports both IPv4 and IPv6 address type.

RES_USE_INET6 Resolver Option:

This option is used to tell the resolver that we want IPv6 addresses returned by gethostname() instead of IPv4 addresses.

- A application can set this option itself by first calling the resolver's re_init function and than enabling the option as shown below:
  - #include <resolv.h>
    res_init();
    res.options |=*RES*USE_INET6;

  This must be done before the first call to gethostbyname() or gethostbyaddr(). The effect of this option is only on the application that sets the option,.

- IF the environment variable RES_OPTION contains the string inet6, the option is enabled. It is set in the file **.profile** file with the **export attribute as in**

  export RES_OPTIONS = inet6;

  This setting affects every program that we run from out login shell. But if it set in the command line, then if affects only that command.
- The resolver configuration file – normally **/etc/resolv.conf** – can contain the line

  options inet6

  Setting this option ion the resolver configuration file affects all applications on the host that call the resolver functions. Hence this should not be used until all applications in the host are capable of handling IPv6 addresses returned in a hostent structure.

The first method sets the option on a per application basis, the second method on a per user basis and the third method on a per system basis.

gethostbyname2() function and IPv6 support:

gethostbyname2() allows us to specify the address family.

```
#include  <netdb.h>
struct hostent    *gethostbyname2(cost char *hostname, int family);
  returns : non null pointer if OK,  NULL pointer on error with h_errono set
```

The return value is the same as with gethostbyname, a pointer to a hostent structure and this structure remains the same. The logic function depends on the family argument and on the RES_USE_INET6 option.

Following table summarizes the operation of the gethostbyname6 with regard to the new RES_USEINET6 option.

getbostbyname and gethostbyname2 with resolver RES_USE_INET6 options

The logic works on

- whether the RES_USE_INET6 options is **on or off**
- whether the second argument to gethostbyname2() is **AF_INET** OR **AF_INET6**

- whether the resolver searches for A records or for AAAA records and
. whether the returned addresses are of length 4 or 16.

	RES_USE_INET6 option	
	off	On
Gethostbyname (host)	Search for A records. IF found, return IPv4 address (h_length=4) Else error This provides backward compatibility for all existing IPv4 applications	Search for AAAA records, If found, return IPv6 addresses (h_length = 16). Else search for A records. If found, return IPv4 mapped IPv6 addresses (h_length=16). Else error.
Gethostbyname2 (host, AF_INET)	Search for A record. IF found return IPv4 addresses (h_length = 4). Else error	Search for A record. IF found return IPv4 mapped IPv6 addresses (h_length = 16). Else error
Gethostbyname2 (host, AF_INET6)	Search for AAAA record. IF found return IPv6 addresses (h_length = 16). Else error	Search for AAAA record. IF found return IPv6 addresses (h_length = 16). Else error

The operation of gethostbyname 2 is as follows:

- If the family argument is AF_INET, a query is made for the A records. IF unsuccessful, the function returns a null pointer. IF successful, the type and assize of the returned addresses depends on the new RES_USE_INET6 resolver option : if the option is not set, IPv4 address are returned and the h_length members of the hostent structure will be 4. if the option is set, IPv4 mapped IPv6 address are returned and the h_length member of the hostent structure will be 16.
- If the family argument is AF_INET6, a query is made for AAAA records. IF successful, IPv6 addresses are returned and the h_length member of the hostent structure will be 16. otherwise the function returns a null pointer.

The source code that is given below describe the action of gethostbyname and RES_USE_INET^ options.

```
Strcut  hostent * gethostbyname (const char *name)  {
        Struct hostent

}
```