UNIT -3

UDP and Socket Options

Elementary UDP socket

Introduction :

There are some fundamental differences between applications written using TCP versus those that use UDP. These are because of the differences in the two transport layers: UDP is a connectionless, unreliable, datagram protocol, quite unlike the connection-oriented, reliable byte stream provided by TCP. Nevertheless, there are instances when it makes sense to use UDP instead of TCP, and we will go over this design choice in Some popular applications are built using UDP: DNS, NFS, and SNMP, for example.

shows the function calls for a typical UDP client/server. The client does not establish a connection with the server. Instead, the client just sends a datagram to the server using the sendto function (described in the next section), which requires the address of the destination (the server) as a parameter. Similarly, the server does not accept a connection from a client. Instead, the server just calls the 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.

Figure . Socket functions for UDP client/server.

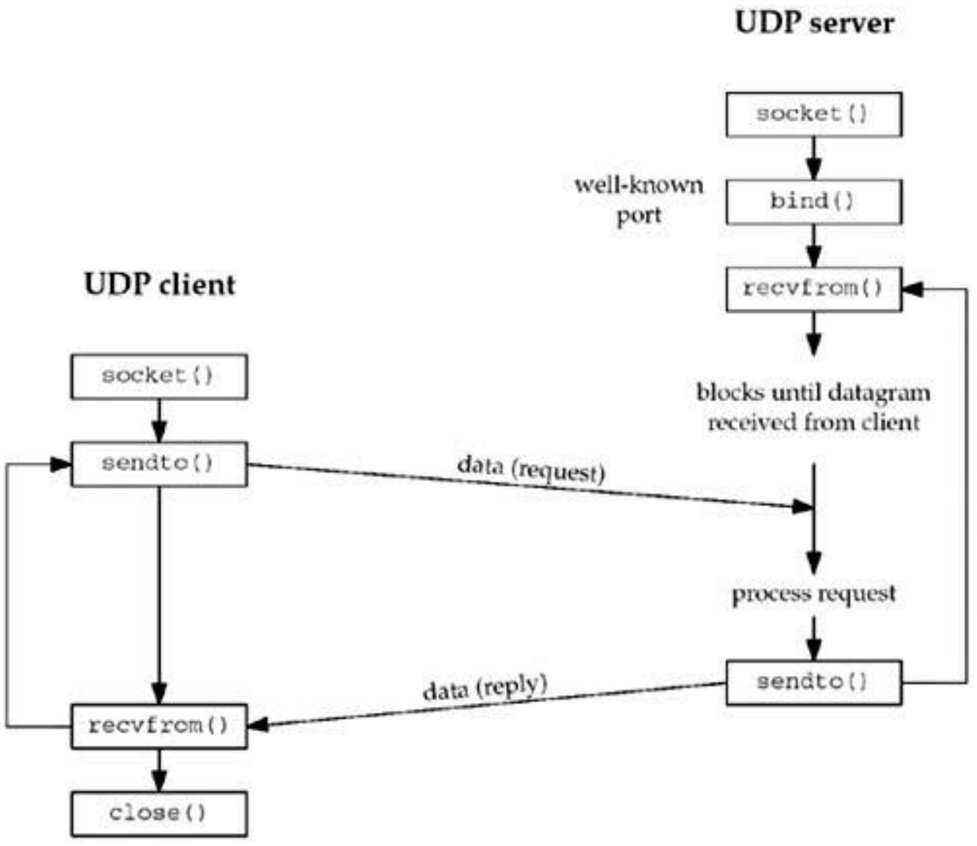


Figure shows a timeline of the typical scenario that takes place for a UDP client/server exchange. We can compare this to the typical TCP exchange that was shown

In this chapter, we will describe the new functions that we use with UDP sockets, recvfrom and sendto, and redo our echo client/server to use UDP. We will also describe the use of the connectfunction with a UDP socket, and the concept of asynchronous errors.

UDP echo server Function

UDP Echo Server: main Function:

We will now redo our simple echo client/server from Chapter 5 using UDP. Our UDP client and server programs follow the function call flow that we diagrammed in Figure 8.1.Figure 8.2 depicts the functions that are used. Figure 8.3 shows the server mainfunction.

Figure Simple echo client/server using UDP.

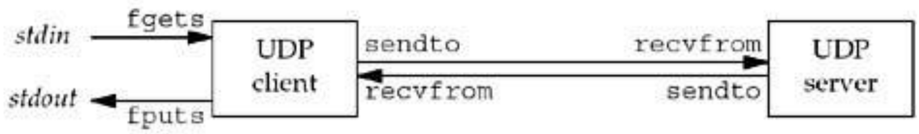


Figure UDP echo server.

udpcliserv/udpserv01.c

```
#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)); }
```

Create UDP socket, bind server's well-known port :

7–12 We create a UDP socket by specifying the second argument to socket as SOCK_DGRAM (a datagram socket in the IPv4 protocol). As with the TCP server example, the IPv4 address for thebind is specified as INADDR_ANY and the server's well-known port is the constant SERV_PORTfrom the unp.hheader.
13 The function dg_echo is called to perform server processing.

UDP Echo Server: dg_echo Function

Figure shows the dg_echo function.

Figure dg_echo function: echo lines on a datagram socket.

lib/dg_echo.c

```
#include        "unp.h"

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, pcliaddr, len);
        }
}
```

UDP Echo Server: dg_echo Function

Figure shows the dg_echo function.

Figure dg_echo function: echo lines on a datagram socket.

lib/dg_echo.c

```
#include        "unp.h"

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

```
 8       for ( ; ; ) {
 12
 13           len = clilen;
14            n = Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

15       Sendto(sockfd, mesg, n, 0, pcliaddr, len);12         }
13 }
```

Read datagram, echo back to sender

8–12 This function is a simple loop that reads the next datagram arriving at the server's port using recvfromand sends it back using sendto.

Despite the simplicity of this function, there are numerous details to consider. First, this function never terminates. Since UDP is a connectionless protocol, there is nothing like an EOF as we have with TCP.

Next, this function provides an *iterative server*, not a concurrent server as we had with TCP. There is no call to fork, so a single server process handles any and all clients. In general, most TCP servers are concurrent and most UDP servers are iterative.

There is implied queuing taking place in the UDP layer for this socket. Indeed, each UDP socket has a receive buffer and each datagram that arrives for this socket is placed in that socket receive buffer. When the process calls recvfrom, the next datagram from the buffer is returned to the process in a first-in, first-out (FIFO) order. This way, if multiple datagrams arrive for the socket before the process can read what's already queued for the socket, the arriving datagrams are just added to the socket receive buffer. But, this buffer has a limited size. We discussed this size and how to increase it with the SO_RCVBUF socket option in Section7.5.

Figure 8.5 summarizes our TCP client/server from Chapter 5 when two clients establish connections with the server.

UDP Echo Client: dg_cli Function:

Figure 8.8 shows the function dg_cli, which performs most of the client processing.

Figure 8.8 dg_cli function: client processing loop.

lib/dg_cli.c

```
 #include           "unp.h"

 1 Void
 2
 3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)4 {
 5       int          n;
 6       char        sendline[MAXLINE], recvline[MAXLINE + 1];
```

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

8                Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

9                n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);

10               recvline[n] = 0;              /* null terminate */
11               Fputs(recvline, stdout);
12          }
13   }
```

7–12 There are four steps in the client processing loop: read a line from 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.

Our client has not asked the kernel to assign an ephemeral port to its socket. (With a TCP client, we said the call to connect is where this takes place.) With a UDP socket, the first time the process calls sendto, if the socket has not yet had a local port bound to it, that is when an ephemeral port is chosen by the kernel for the socket. As with TCP, the client can call bind explicitly, but this is rarely done.

Notice that the call to recvfrom specifies a null pointer as the fifth and sixth arguments. This tells the kernel that we are not interested in knowing who sent the reply. There is a risk that any process, on either the same host or some other host, can send a datagram to the client's IP address and port, and that datagram will be read by the client, who will think it is the server's reply. We will address this in Section 8.8.

As with the server function dg_echo, the client function dg_cli is protocol-independent, but the client main function is protocol-dependent. The main function allocates and initializes a socket address structure of some protocol type and then passes a pointer to this structure, along with its size, to dg_cli.

Lost Datagram :

Our UDP client/server example is not reliable. If a client datagram is lost (say it is discarded by some router between the client and server), the client will block forever in its call to recvfromin the function dg_cli, 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 again block forever in its call to recvfrom. A typical way to prevent this is to place a timeout on the client's call to recvfrom. We will discuss this in Section 14.2.
Just placing a timeout on the recvfrom is not the entire solution. For example, if we do timeout, we cannot tell whether our datagram never made it to the server, or if the server's replynever made it back. If the client's request was something like "transfer a certain amount of money from account A to account B" (instead of our simple echo server), it would make a big difference as to whether the request was lost or the reply was lost. We will talk more about adding reliability to a UDP client/server in Section

In network programming, it is possible for UDP datagrams to be lost during transmission due to a variety of reasons, such as network congestion, hardware failures, or software errors. Since UDP does not provide any reliability or flow control mechanisms, lost datagrams are not automatically retransmitted, and it is up to the application to handle them appropriately.

One common technique for dealing with lost datagrams in UDP is to use a timeout mechanism. The application can set a timer when it sends a datagram, and if it does not receive a response from the receiver within a certain amount of time, it assumes that the datagram was lost and retransmits it. This technique can be effective for small networks with low packet loss rates, but it can be inefficient in larger networks with higher packet loss rates, as it can lead to excessive retransmissions and increased network congestion.

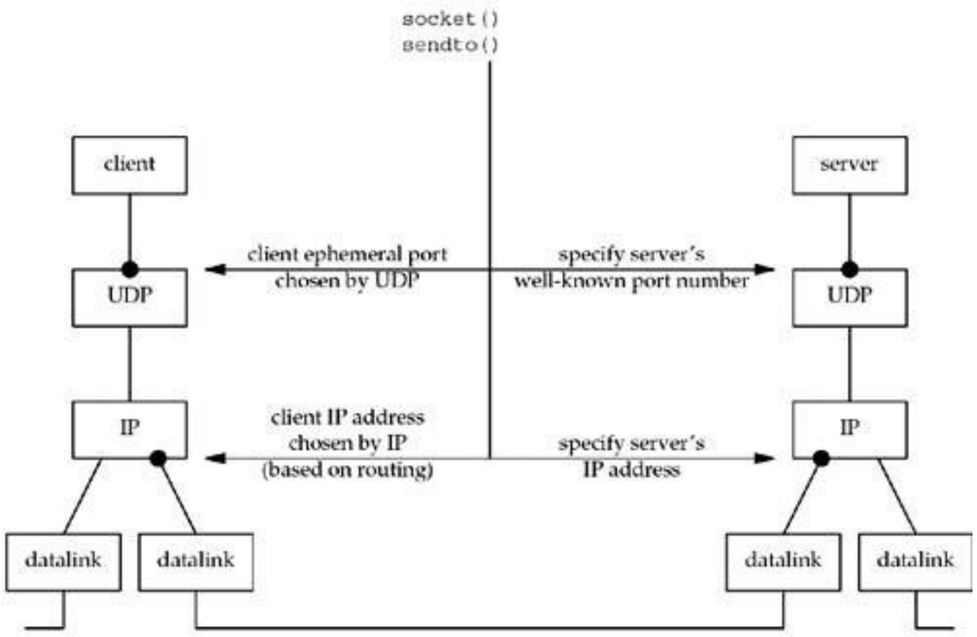
Another technique for dealing with lost datagrams in UDP is to use sequence numbers. The application can assign a unique sequence number to each datagram it sends, and the receiver can use these sequence numbers to detect lost datagrams and request retransmissions. This technique is commonly used in reliable UDP protocols, such as the Real-Time Transport Protocol (RTP) used for streaming media.

In addition to these techniques, there are several other approaches that can be used to handle lost datagrams in UDP, such as forward error correction (FEC), automatic repeat request (ARQ) schemes, and selective retransmission. The choice of technique depends on the specific requirements of the application and the characteristics of the network.

Summary of UDP example :

Figure 8.11 shows as bullets the four values that must be specified or chosen when the client sends a UDP datagram.

Figure. Summary of UDP client/server from client's perspective.



socket()
sendto()
client
server
UDP
UDP
client ephemeral port chosen by UDP
specify server's well-known port number
IP
IP
client IP address chosen by IP (based on routing)
specify server's IP address
datalink
datalink
datalink
datalink

The client must specify the server's IP address and port number for the call to sendto. Normally, the client's IP address and port are chosen automatically by the kernel, although we mentioned that the client can call bind if it so chooses. If these two values for the client are chosen by the kernel, we also mentioned that the client's ephemeral port is chosen once, on the first sendto, and then it never changes. The client's IP address, however, can change for every UDP datagram that the client sends, assuming the client does not bind a specific IP address to the socket. The reason is shown in Figure 8.11: If the client host is multihomed, the client could alternate between two destinations, one going out the datalink on the left, and the other going out the datalink on the right. In this worst-case scenario, the client's IP address, as chosen by the kernel based on the outgoing datalink, would change for every datagram.

What happens if the client binds an IP address to its socket, but the kernel decides that an outgoing datagram must be sent out some other datalink? In this case the IP datagram will contain a source IP address that is different from the IP address of the outgoing datalink (see Exercise 8.6).

Figure 8.12 shows the same four values, but from the server's perspective.

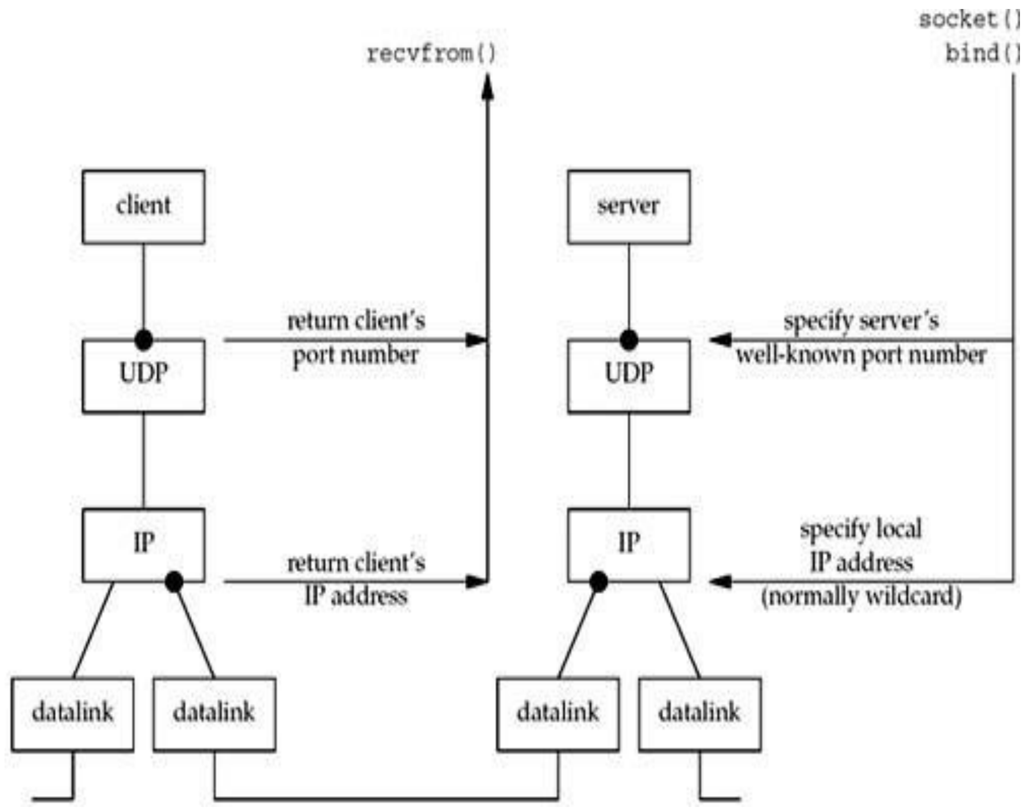

recvfrom()
socket()
bind()
client
server
return client's port number
specify server's well-known port number
UDP
UDP
IP
IP
return client's IP address
specify local IP address (normally wildcard)
datalink
datalink
datalink
datalink

Figure 8.12. Summary of UDP client/server from server's

Lack of flow control with UDP :

One of the key characteristics of User Datagram Protocol (UDP) in network programming is the lack of flow control mechanisms. Unlike Transmission Control Protocol (TCP), which provides flow control to ensure reliable and ordered delivery of data, UDP simply sends packets to the destination without checking whether the destination is ready to receive them or not.

This lack of flow control can lead to several issues:

1. Packet loss: Since there is no flow control in UDP, packets may be lost during transmission due to network congestion or other factors. If a packet is lost, there is no mechanism for retransmission, and the receiving application will never receive the lost data.
2. Out-of-order delivery: Another issue that can arise due to the lack of flow control in UDP is out-of-order delivery of packets. Packets may arrive at the destination out of order, which can cause problems for applications that rely on the packets arriving in a specific order.
3. Overloading the network: Since UDP does not have flow control mechanisms, it is possible for an application to flood the network with packets, leading to congestion and potentially causing performance issues for other applications running on the same network.

Despite these potential issues, UDP is still widely used in applications where speed and efficiency are more important than reliability, such as online gaming, video conferencing, and real-time streaming. In such cases, the lack of flow control may be acceptable as long as the data is delivered quickly and the occasional loss or out-of-order delivery of packets does not significantly affect the user experience

We now examine the effect of UDP not having any flow control. First, we modify our dg_cli func fixed number of datagrams. It no longer reads from standard input. Figure 8.19 shows the new vfunction writes 2,000 1,400-byte UDP datagrams to the server.

We next modify the server to receive datagrams and count the number received. This server no ldatagrams back to the client. Figure 8.20 shows the new dg_echo function. When we terminate t our terminal interrupt key (SIGINT), it prints the number of received datagrams and terminates.

Socket options :

Getsockopt() and setsockopt():

getsockopt() and setsockopt() are two functions in network programming that allow applications to access and modify socket options for a given socket.

1. getsockopt():

getsockopt() is a function used to retrieve socket options. This function takes four arguments:

- Socket: A socket file descriptor returned by the socket() system call.
- Level: The protocol level at which the option resides, such as SOL_SOCKET, IPPROTO_TCP, or IPPROTO_IP.
- Option name: The name of the option to be retrieved.
- Option value: A pointer to a buffer where the value of the option will be stored.

When called, getsockopt() retrieves the current value of the specified socket option and stores it in the buffer pointed to by the option value argument.

2. setsockopt():

3. setsockopt() is a function used to set socket options. This function takes five arguments:

- Socket: A socket file descriptor returned by the socket() system call.
- Level: The protocol level at which the option resides, such as SOL_SOCKET, IPPROTO_TCP, or IPPROTO_IP.
- Option name: The name of the option to be set.
- Option value: A pointer to a buffer containing the new value of the option.
- Option length: The size of the buffer pointed to by the option value argument.

When called, setsockopt() sets the specified socket option to the new value specified in the option value argument.

Some examples of socket options that can be set or retrieved using getsockopt() and setsockopt() include the following:

- SO_REUSEADDR: Allows a socket to be bound to an address that is already in use.
- SO_KEEPALIVE: Enables the sending of periodic keep-alive messages on a connection to detect if the other end of the connection has gone down.
- TCP_NODELAY: Disables Nagle's algorithm, which delays the sending of small packets in order to improve network efficiency.

sockfd must refer to an open socket descriptor. *level* specifies the code in the system that interprets the option: the general socket code or some protocol-specific code (e.g., IPv4, IPv6, TCP, or SCTP).

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 getsockopt. The size of this variable is specified by the final argument, as a value for setsockoptand as a value-result forgetsockopt.

Figures 7.1 and 7.2 summarize the options that can be queried by getsockopt or set by setsockopt. The "Datatype" column shows the datatype of what the *optval* pointer must point to for each option. We use the notation of two braces to indicate a structure, as in linger{}to mean a struct linger.

```
#include <sys/socket.h>
int  getsockopt(intsockfd,intlevel,intoptname,void  *optval,socklen_t *optlen);
int  setsockopt(intsockfd,intlevel,intoptname,const  void  *optvalsocklen_t
optlen);
```

Socket State :

Regardless of the socket type (TCP or UDP), operation mode (blocking, non-blocking, or asynchronous) or application type (single- or multi-threaded), changes in socket states are what propel a network application. Therefore, for a network application to be efficient, it needs to detect and handle changes in socket states as efficient as possible.

The state of a socket determines which network operations will succeed, which operations will block, and which operations with will fail (the socket state even determines the error code). Sockets have a finite number of states, and the WinSock API clearly defines the conditions that trigger a transition from one state to another. Note that different types of sockets (stream vs. datagram have different states and different transitions.

In network programming, a socket is a combination of an IP address and a port number that provides a communication endpoint for two processes to exchange data. A socket can be in one of several states, which reflect its current state of operation. The exact states and terminology may vary depending on the specific implementation and protocol, but here are some common states that a socket can be in:

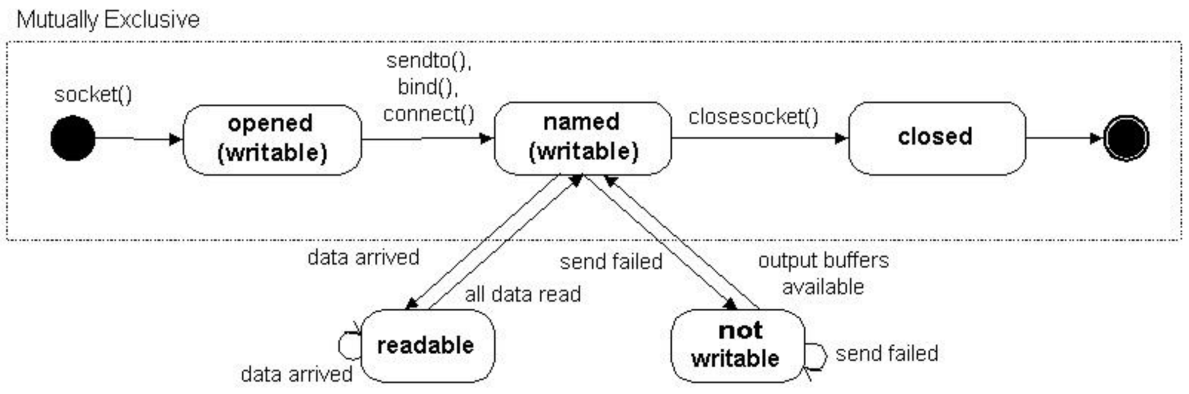


Fig : Socket state diagram

1.Opened:

socket() returned an unnamed socket (an unnamed socket is one that is not bound to a local address and port). The socket can be named explicitly with bind or implicitly with **sendto()** or **connect()**.

2.Named :

A named socket is one that is bound to a local address and a port. The socket can now send and/or receive.

3.Readable:

The network system received data and is ready to be read by the application not using **recv()** or **recvfrom()**.

4.Writable :

The network system does not have enough buffers to accommodate outgoing data.

5.Closed :

The socket handle is invalid.

1.

2. LISTEN:

   The socket is waiting for incoming connection requests from clients. This state is used by servers that listen for incoming connections on a specific port.

3. SYN-SENT:

   The socket has sent a SYN packet to initiate a connection request to a remote host.

4. SYN-RECEIVED:

   The socket has received a SYN packet from a remote host and has sent a SYN-ACK packet in response.

5. ESTABLISHED:

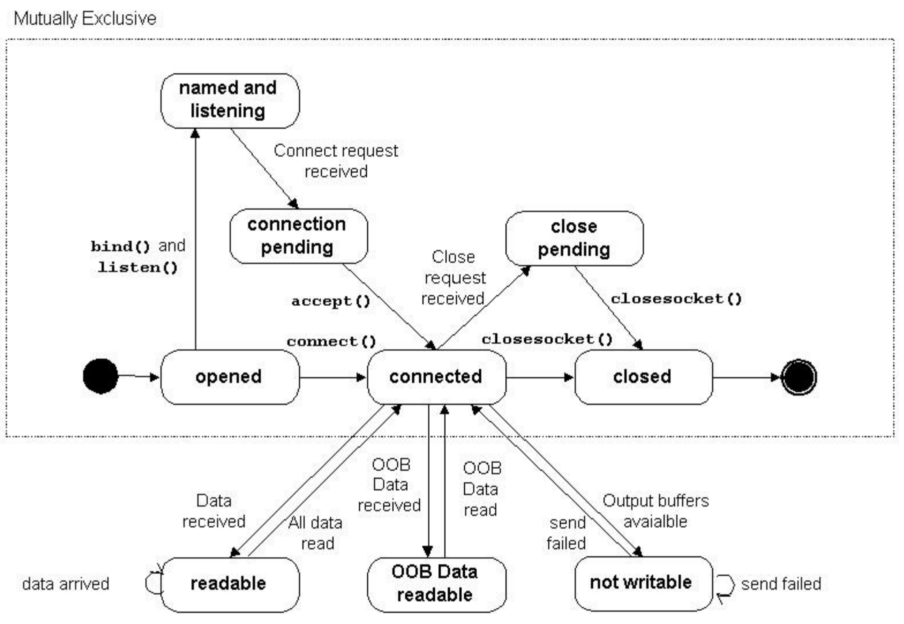
The socket has successfully established a connection with a remote host and is ready to exchange data.

FIN-WAIT-1:

The socket has sent a FIN packet to initiate the termination of the connection.

6. FIN-WAIT-2: The socket is waiting for a FIN packet from the remote host to acknowledge the termination request.
7. TIME-WAIT: The socket is waiting for any remaining packets to be delivered before fully closing the connection. This is to ensure that all data has been transmitted and received.
8. CLOSING: The socket is waiting for the remote host to acknowledge the termination request.
9. CLOSE-WAIT: The socket has received a FIN packet from the remote host and is waiting for the application to close the connection.
10. LAST-ACK: The socket has received a FIN packet from the remote host and has sent an ACK packet in response.
11. UNKNOWN: The socket is in an unknown state, which may indicate an error or unexpected condition.

These socket states are typically used in the context of TCP connections. Other protocols, such as UDP, may use a different set of states or may not use states at all due to their connectionless nature.



Generic Socket options:

a discussion of the generic socket options. These options are protocol- independent (that is, they are handled by the protocol-independent code within the kernel, notby one particular protocol module such as IPv4), but some of the options apply to only certain types of sockets. For example, even though the SO_BROADCAST socket option is called "generic," it applies only to datagram sockets.

SO_BROADCAST Socket Option:

This option enables or disables the ability of the process to send broadcast messages. Broadcasting is supported for only datagram sockets and only on networks that support theconcept of a broadcast message (e.g., Ethernet, token ring, etc.). You cannot broadcast on apoint-to-point link or any connection-based transport protocol such as SCTP or TCP. We will talk more about broadcasting in Chapter 20.

Since an application must set this socket option before sending a broadcast datagram, it prevents a process from sending a broadcast when the application was never designed to broadcast. For example, a UDP application might take the destination IP address as a command-line argument, but the application never intended for a user to type in a broadcast address. Rather than forcing the application to try to determine if a given address is a broadcast address or not, the test is in the kernel: If the destination address is a broadcast address and this socket option is not set, EACCESis returned (p. 233 of TCPv2).

SO_DEBUG Socket Option:

This option is supported only by TCP. When enabled for a TCP socket, the kernel keeps track ofdetailed information about all the packets sent or received by TCP for the socket. These are kept in a circular buffer within the kernel that can be examined with the trpt program. Pages 916–920 of TCPv2 provide additional details and an example that uses this option.

SO_DONTROUTE Socket Option:

This option specifies that outgoing packets are to bypass the normal routing mechanisms of theunderlying protocol. For example, with IPv4, the packet is directed to the appropriate local interface, as specified by the network and subnet portions of the destination address. If the local interface cannot be determined from the destination address (e.g., the destination is not on the other end of a point-to-point link, or is not on a shared network), ENETUNREACH is returned.

The equivalent of this option can also be applied to individual datagrams using the MSG_DONTROUTE flag with the send, sendto, or sendmsg functions.

This option is often used by routing daemons (e.g., routed and gated) to bypass the routing table and force a packet to be sent out a particular interface.

SO_ERROR Socket Option:

When an error occurs on a socket, the protocol module in a Berkeley-derived kernel sets a variable named so_error for that socket to one of the standard Unix E*xxx* values. This is called the *pending error* for the socket. The process can be immediately notified of the error in one of two ways:

1. If the process is blocked in a call to select on the socket (Section 6.3), for either readability or writability, select returns with either or both conditions set.
2. If the process is using signal-driven I/O (Chapter 25), the SIGIO signal is generated for either the process or the process group.

The process can then obtain the value of so_error by fetching the SO_ERROR socket option. The integer value returned by getsockopt is the pending error for the socket. The value of so_error is then reset to 0 by the kernel (p. 547 of TCPv2).

Ifso_error is nonzero when the process calls read and there is no data to return, read returns–1 with errno set to the value of so_error (p. 516 of TCPv2). The value of so_error isthen reset to 0. If there is data queued for the socket, that data is returned by read instead of the error condition. If so_error is nonzero when the process calls write, –1 is returned with errno set to the value of so_error(p. 495 of TCPv2) and so_error is reset to 0.

There is a bug in the code shown on p. 495 of TCPv2 in that so_error is not reset to 0. This has been fixed in most modern releases. Anytime the pending error for a socket is returned, it must be reset to 0.

This is the first socket option that we have encountered that can be fetched but cannot be set.

SO_KEEPALIVE Socket Option:

When the keep-alive option is set for a TCP socket and no data has been exchanged across thesocket in either direction for two hours, TCP automatically sends a *keep-alive probe* to the peer. This probe is a TCP segment to which the peer must respond. One of three scenarios results:

1. The peer responds with the expected ACK. The application is not notified (since everything is okay). TCP will send another probe following another two hours of inactivity.
2. The peer responds with an RST, which tells the local TCP that the peer host has crashedand rebooted. The socket's pending error is set to ECONNRESET and the socket is closed.
3. There is no response from the peer to the keep-alive probe. Berkeley-derived TCPs send 8 additional probes, 75 seconds apart, trying to elicit a response. TCP will give up if thereis no response within 11 minutes and 15 seconds after sending the first probe.

HP-UX 11 treats the keep-alive probes in the same way as it would treat data, sending the second probe after a retransmission timeout and doubling the timeout for each packet until the configured maximum interval, with a default of 10 minutes.

SO_REUSEADDR:

Allows multiple sockets to bind to the same port on the same IP address. This is useful for servers that need to listen on the same port for incoming connections from different clients.

SO_SNDBUF:
Sets or retrieves the size of the socket send buffer. This buffer is used to temporarily store data that is being sent over the socket.

SO_RCVBUF:
Sets or retrieves the size of the socket receive buffer. This buffer is used to temporarily store data that is being received over the socket.

SO_LINGER:
Determines what happens when a socket is closed. If SO_LINGER is set, the close() function will block until all pending data has been sent or a timeout occurs.

TCP_NODELAY:
Disables Nagle's algorithm, which delays the sending of small packets in order to improve network efficiency. This can be useful for applications that need to send small amounts of data quickly.

TCP_KEEPIDLE:
Sets the amount of time that a connection can remain idle before TCP starts sending keep-alive messages.

TCP_KEEPINTVL:
Sets the amount of time between keep-alive messages.

TCP_KEEPCNT:
Sets the number of keep-alive messages that can be sent before the connection is considered dead.

These are just a few examples of the many socket options available in network programming. The exact set of options may vary depending on the operating system and protocol being used.

IPV4 Socket Options :

These socket options are processed by IPv4 and have a *level* of IPPROTO_IP. We defer discussion of the multicasting socket options until Section 21.6.

IP_HDRINCLSocket Option:

If this option is set for a raw IP socket (Chapter 28), we must build our own IP header for all the datagrams we send on the raw socket. Normally, the kernel builds the IP header for datagrams sent on a raw socket, but there are some applications (notably traceroute) that build their own IP header to override values that IP would place into certain header fields.

When this option is set, we build a complete IP header, with the following exceptions:

- IP always calculates and stores the IP header checksum.
- If we set the IP identification field to 0, the kernel will set the field.
- If the source IP address is INADDR_ANY, IP sets it to the primary IP address of the outgoing interface.
- Setting IP options is implementation-dependent. Some implementations take any IP options that were set using the IP_OPTIONS socket option and append these to the header that we build, while others require our header to also contain any desired IP options.
- Some fields must be in host byte order, and some in network byte order. This is implementation-dependent, which makes writing raw packets with IP_HDRINCL not as portable as we'd like.

We show an example of this option in Section 29.7. Pages 1056–1057 of TCPv2 provide additional details on this socket option.

IP_OPTIONSSocket Option:

Setting this option allows us to set IP options in the IPv4 header. This requires intimate knowledge of the format of the IP options in the IP header. We will discuss this option withregard to IPv4 source routes in Section 27.3.

IP_RECVDSTADDR Socket Option:

This socket option causes the destination IP address of a received UDP datagram to be returned as ancillary data by recvmsg. We will show an example of this option in Section 22.2.

IP_RECVIF Socket Option:

This socket option causes the index of the interface on which a UDP datagram is received to be returned as ancillary data by recvmsg. We will show an example of this option in Section 22.2.

IP_TOSSocket Option :

This option lets us set the type-of-service (TOS) field (which contains the DSCP and ECN fields, Figure A.1) in the IP header for a TCP, UDP, or SCTP socket. If we call getsockoptfor this option, the current value that would be placed into the DSCP and ECN fields in the IP header (which defaults to 0) is returned. There is no way to fetch the value from a received IP datagram.

An application can set the DSCP to a value negotiated with the network service provider to receive prearranged services, e.g., low delay for IP telephony or higher throughput for bulk data transfer. The diffserv architecture, defined in RFC 2474 [Nichols et al. 1998], provides foronly limited backward compatibility with the historical TOS field definition (from RFC 1349 [Almquist 1992]). Application that set IP_TOS to one of the contents from <netinet/ip.h>, for instance,IPTOS_LOWDELAY or IPTOS_THROUGHPUT, should instead use a user-specified DSCP value. The only TOS values that diffserv retains are precedence levels 6 ("internetwork control") and 7 ("network control"); this means that applications that set IP_TOS to IPTOS_PREC_NETCONTROL or IPTOS_PREC_INTERNETCONTROL*will* work in a diffserv network.

RFC 3168 [Ramakrishnan, Floyd, and Black 2001] contains the definition of the ECN field. Applications should generally leave the setting of the ECN field to the kernel, and should specifyzero values in the low two bits of the value set with IP_TOS.

IP_TTL Socket Option:

With this option, we can set and fetch the default TTL (Figure A.1) that the system will use forunicast packets sent on a given socket. (The multicast TTL is set using the IP_MULTICAST_TTLsocket option, described in Section 21.6.) 4.4BSD, for example, uses the default of 64 for both TCP and UDP sockets (specified in the IANA's "IP Option Numbers" registry [IANA]) and 255 forraw sockets. As with the TOS field, calling getsockopt returns the default value of the field that the system will use in outgoing datagrams—there is no way to obtain the value from a received datagram. We will set this socket option with our traceroute program in Figure 28.19.

IPV6 Socket options :

These socket options are processed by IPv6 and have a *level* of IPPROTO_IPV6. We defer discussion of the multicasting socket options until Section 21.6. We note that many of these options make use of *ancillary data* with the recvmsg function, and we will describe this in Section 14.6. All the IPv6 socket options are defined in RFC 3493 [Gilligan et al. 2003] and RFC3542 [Stevens et al. 2003].

IPV6_CHECKSUM Socket Option:

This socket option specifies the byte offset into the user data where the checksum field is located. If this value is non-negative, the kernel will: (i) compute and store a checksum for alloutgoing packets, and (ii) verify the received checksum on input, discarding packets with an invalid checksum. This option affects all IPv6 raw sockets, except ICMPv6 raw sockets. (The kernel always calculates and stores the checksum for ICMPv6 raw sockets.) If a value of -1 is specified (the default), the kernel will not calculate and store the checksum for outgoing packets on this raw socket and will not verify the checksum for received packets.

All protocols that use IPv6 should have a checksum in their own protocol header. These checksums include a pseudoheader (RFC 2460 [Deering and Hinden 1998]) that includes the source IPv6 address as part of the checksum (which differs from all the other protocols that are normally implemented using a raw socket with IPv4). Rather than forcing the application using the raw socket to perform source address selection, the kernel will do this and then calculate and store the checksum incorporating the standard IPv6 pseudo header.

IPV6_DONTFRAG Socket Option:

Setting this option disables the automatic insertion of a fragment header for UDP and raw sockets. When this option is set, output packets larger than the MTU of the outgoing interface will be dropped. No error needs to be returned from the system call that sends the packet, since the packet might exceed the path MTU en-route. Instead, the application should enable theIPV6_RECVPATHMTU option (Section 22.9) to learn about path MTU changes.

IPV6_NEXTHOP Socket Option:

This option specifies the next-hop address for a datagram as a socket address structure, and isa privileged operation. We will say more about this feature in Section 22.8.

IPV6_PATHMTU Socket Option:

This option cannot be set, only retrieved. When this option is retrieved, the current MTU asdetermined by path-MTU discovery is returned (see Section 22.9).

IPV6_RECVDSTOPTS Socket Option:

Setting this option specifies that any received IPv6 destination options are to be returned as ancillary data by recvmsg. This option defaults to OFF. We will describe the functions that are used tobuildandprocesstheseoptions

IPV6_RECVHOPLIMIT Socket Option:

Setting this option specifies that the received hop limit field is to be returned as ancillary data by recvmsg. This option defaults to OFF. We will describe this option

There is no way with IPv4 to obtain the received TTL field.

IPV6_RECVHOPOPTS Socket Option:

Setting this option specifies that any received IPv6 hop-by-hop options are to be returned as ancillary data by recvmsg. This option defaults

IPV6_RECVPATHMTU Socket Option:

Setting this option specifies that the path MTU of a path is to be returned as ancillary data byrecvmsg (without any accompanying data) when it changes. We will describe this option in

IPV6_RECVPKTINFO Socket Option:

Setting this option specifies that the following two pieces of information about a received IPv6 datagram are to be returned as ancillary data by recvmsg: the destination IPv6 address andthe arriving interface index. We will describe this option in

IPV6_RECVRTHDR Socket Option:

Setting this option specifies that a received IPv6 routing header is to be returned as ancillary data by recvmsg. This option defaults to OFF. We will describe the functions that are used to build and process an IPv6 routing header

IPV6_RECVTCLASS Socket Option:

Setting this option specifies that the received traffic class (containing the DSCP and ECN fields) is to be returned as ancillary data by recvmsg. This option defaults to OFF. We will describe this option

IPV6_UNICAST_HOPS Socket Option:

This IPv6 option is similar to the IPv4 IP_TTL socket option. Setting the socket option specifiesthe default hop limit for outgoing datagrams sent on the socket, while fetching the socket option returns the value for the hop limit that the kernel will use for the socket. The actual hoplimit field from a received IPv6 datagram is obtained by using the IPV6_RECVHOPLIMIT socket option. We will set this socket option with our traceroute program in Figure 28.19.

IPV6_USE_MIN_MTU Socket Option:

Setting this option to 1 specifies that path MTU discovery is not to be performed and that packets are sent using the minimum IPv6 MTU to avoid fragmentation. Setting it to 0 causespath MTU discovery to occur for all destinations. Setting it to–1 specifies that path MTU discovery is performed for unicast destinations but the minimum MTU is used when sending tomulticast destinations. This option defaults to –1. We will describe this option in Section 22.9.

IPV6_V6ONLY Socket Option:

Setting this option on an AF_INET6 socket restricts it to IPv6 communication only. This optiondefaults to OFF, although some systems have an option to turn it ON by default. We will describe IPv4 and IPv6 communication using AF_INET6 sockets in Sections 12.2 and 12.3.

IPV6_XXX Socket Options:

Most of the IPv6 options for header modification assume a UDP socket with 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 getsockopt and setsockopt instead. The socket option is the same as the type of the ancillary data, and the buffer contains the same information as would be present in the ancillary data.

ICMPV6 Socket Options and TCP socket options

ICMPv6 Socket Option :

ICMPv6 (Internet Control Message Protocol version 6) is a protocol that is used by network devices to send error messages and operational information about the state of the network. In network programming, ICMPv6 socket options can be used to set or retrieve parameters related to ICMPv6 messages. Here are some of the ICMPv6 socket options that are commonly used:

This socket option is processed by ICMPv6 and has a *level* of IPPROTO_ICMPV6.

ICMP6_FILTER Socket Option :

This option lets us fetch and set an icmp6_filter structure that specifies which of the 256 possible ICMPv6 message types will be passed to the process on a raw socket.

1. IPV6_CHECKSUM:
   Enables or disables the calculation of the ICMPv6 checksum. If the checksum is disabled, the network stack will not verify the integrity of the ICMPv6 message.
2. IPV6_RECVERR:
   Enables the reception of ICMPv6 error messages. When this option is enabled, the network stack will generate an error message if an ICMPv6 message is received that indicates an error condition.
3. IPV6_PKTINFO:
   Enables the reception of additional information about the incoming packet. This information includes the source and destination addresses of the packet, as well as the incoming interface index.
4. IPV6_NEXTHOP:
   Specifies the next-hop address for the outgoing packet. This option is used in routing applications to specify the address of the next router that the packet should be sent to.
5. IPV6_RECVHOPLIMIT:
   Enables the reception of the hop limit field from incoming ICMPv6 packets.
6. IPV6_RECVTCLASS:
   Enables the reception of the traffic class field from incoming ICMPv6 packets.
7. IPV6_UNICAST_HOPS:
   Sets the maximum number of hops that a unicast packet can travel before it is discarded.

8. IPV6_MULTICAST_HOPS:

   Sets the maximum number of hops that a multicast packet can travel before it is discarded.

9. IPV6_MULTICAST_IF:

   Specifies the interface that multicast packets should be sent on.

10. IPV6_MULTICAST_LOOP:
Enables or disables loopback of multicast packets. If loopback is enabled, multicast packets will be received by the sending host as well as other hosts on the same multicast group.

programming. The exact set of options may vary depending on the operating system and protocol being used.

PV6 Socket Options :

These socket options are processed by IPv6 and have a *level* of IPPROTO_IPV6. We defer discussion of the multicasting socket options until Section 21.6. We note that many of these options make use of *ancillary data* with the recvmsg function, and we will describe this in Section 14.6. All the IPv6 socket options are defined in RFC 3493 [Gilligan et al. 2003] and RFC3542 [Stevens et al. 2003].

IPV6_CHECKSUM Socket Option :

This socket option specifies the byte offset into the user data where the checksum field is located. If this value is non-negative, the kernel will: (i) compute and store a checksum for all outgoing packets, and (ii) verify the received checksum on input, discarding packets with an invalid checksum. This option affects all IPv6 raw sockets, except ICMPv6 raw sockets. (The kernel always calculates and stores the checksum for ICMPv6 raw sockets.) If a value of -1 is specified (the default), the kernel will not calculate and store the checksum for outgoing packets on this raw socket and will not verify the checksum for received packets.

All protocols that use IPv6 should have a checksum in their own protocol header. These checksums include a pseudoheader (RFC 2460 [Deering and Hinden 1998]) that includes the source IPv6 address as part of the checksum (which differs from all the other protocols that are normally implemented using a raw socket with IPv4). Rather than forcing the application using the raw socket to perform source address selection, the kernel will do this and then calculate and store the checksum incorporating the standard IPv6 pseudoheader.

IPV6_DONTFRAG Socket Option:

Setting this option disables the automatic insertion of a fragment header for UDP and raw sockets. When this option is set, output packets larger than the MTU of the outgoing interfacewill be dropped. No error needs to be returned from the system call that sends the packet, since the packet might exceed the path MTU en-route. Instead, the application should enable theIPV6_RECVPATHMTUoption (Section 22.9) to learn about path MTU changes.

IPV6_NEXTHOP Socket Option:

This option specifies the next-hop address for a datagram as a socket address structure, and isa privileged operation. We will say more about this feature

IPV6_PATHMTU Socket Option:

This option cannot be set, only retrieved. When this option is retrieved, the current MTU asdetermined by path-MTU discovery is returned (see Section 22.9).

IPV6_RECVDSTOPTS Socket Option:

Setting this option specifies that any received IPv6 destination options are to be returned as ancillary data by recvmsg. This option defaults to OFF. We will describe the functions that are used to build and process these options in

IPV6_RECVHOPLIMIT Socket Option:

Setting this option specifies that the received hop limit field is to be returned as ancillary data by recvmsg. This option defaults to OFF. We will describe this option

There is no way with IPv4 to obtain the received TTL field.

IPV6_RECVHOPOPTS Socket Option:

Setting this option specifies that any received IPv6 hop-by-hop options are to be returned as ancillary data by recvmsg. This option defaults to OFF. We will describe the functions that are used to build and process these options

IPV6_RECVPATHMTU Socket Option:

Setting this option specifies that the path MTU of a path is to be returned as ancillary data by recvmsg (without any accompanying data) when it changes. We will describe this option

IPV6_RECVPKTINFO Socket Option:

Setting this option specifies that the following two pieces of information about a received IPv6 datagram are to be returned as ancillary data by recvmsg: the destination IPv6 address and the arriving interface index. We will describe this option

IPV6_RECVRTHDR Socket Option:

Setting this option specifies that a received IPv6 routing header is to be returned as ancillary data by recvmsg. This option defaults to OFF. We will describe the functions that are used to build and process an IPv6 routing header

IPV6_RECVTCLASS Socket Option

Setting this option specifies that the received traffic class (containing the DSCP and ECN fields) is to be returned as ancillary data by recvmsg. This option defaults to OFF. We will describe this option

IPV6_UNICAST_HOPS Socket Option:

This IPv6 option is similar to the IPv4 IP_TTL socket option. Setting the socket option specifiesthe default hop limit for outgoing datagrams sent on the socket, while fetching the socket option returns the value for the hop limit that the kernel will use for the socket. The actual hop limit field from a received IPv6 datagram is obtained by using the IPV6_RECVHOPLIMIT socket option. We will set this socket option with our tracerouteprogram in

IPV6_USE_MIN_MTU Socket Option:

Setting this option to 1 specifies that path MTU discovery is not to be performed and that packets are sent using the minimum IPv6 MTU to avoid fragmentation. Setting it to 0 causespath MTU discovery to occur for all destinations. Setting it to–1 specifies that path MTU discovery is performed for unicast destinations but the minimum MTU is used when sending to multicast destinations. This option defaults to –1. We will describe this option

IPV6_V6ONLY Socket Option:

Setting this option on an AF_INET6 socket restricts it to IPv6 communication only. This option defaults to OFF, although some systems have an option to turn it ON by default. We will describe IPv4 and IPv6 communication using AF_INET6 sockets in Sections 12.2 and 12.3.

IPV6_XXX Socket Options:

Most of the IPv6 options for header modification assume a UDP socket with 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 getsockopt and setsockopt instead. The socket option is the same as the type of the ancillary data, and the buffer contains the same information as would be present in the ancillary data. We will describe this in Section 27.7.

SCTP Socket options :

The relatively large number of socket options for SCTP (17 at present writing) reflects the finer grain of control SCTP provides to the application developer. We specify the *level* as IPPROTO_SCTP.

Several options used to get information about SCTP require that data be passed into the kernel(e.g., association ID and/or peer address). While some implementations of getsockopt support passing data both into and out of the kernel, not all do. The SCTP API defines a sctp_opt_info function (Section 9.11) that hides this difference. On systems on which getsockopt does support this, it is simply a wrapper around getsockopt. Otherwise, it performs the required action, perhaps using a custom ioctl or a new system call. We recommend always using sctp_opt_info when retrieving these opti†ns for maximum portability. These options are marked with a dagger ( ) in Figure 7.2 and include SCTP_ASSOCINFO,SCTP_GET_PEER_ADDR_INFO,SCTP_PEER_ADDR_PARAMS,SCTP_PRIMARY_ADDR, SCTP_RTOINFO, and SCTP_STATUS.

SCTP_ADAPTION_LAYER Socket Option :

During association initialization, either endpoint may specify an adaption layer indication. Thisindication is a 32-bit unsigned integer that can be used by the two applications to coordinate any local application adaption layer. This option allows the caller to fetch or set the adaptionlayer indication that this endpoint will provide to peers.

When fetching this value, the caller will only retrieve the value the local socket will provide to all future peers. To retrieve the peer's adaption layer indication, an application must subscribeto adaption layer events.

SCTP_ASSOCINFO Socket Option:

The SCTP_ASSOCINFO socket option can be used for three purposes: (i) to retrieve information about an existing association, (ii) to change the parameters of an existing association, and/or
(iii) to set defaults for future associations. When retrieving information about an existing association, the sctp_opt_info function should be used instead of getsockopt. This option takes as input the sctp_assocparamsstructure.

```
struct      sctp_assocparams      {
   sctp_assoc_t     sasoc_assoc_id;
   u_int16_t sasoc_asocmaxrxt;
   u_int16_t sasoc_number_peer_destinations; u_int32_t
   sasoc_peer_rwnd;
   u_int32_t      sasoc_local_rwnd;
   u_int32_t sasoc_cookie_life;
};
```

These fields have the following meaning:

- sasoc_assoc_id holds the identification for the association of interest. If this value is setto 0 when calling the setsockopt function, then sasoc_asocmaxrxt and sasoc_cookie_life represent values that are to be set as defaults on the socket. Calling getsockopt will return association-specific information if the association ID is supplied; otherwise, if this field is 0, the default endpoint settings will be returned.

- sasoc_asocmaxrxt holds the maximum number of retransmissions an association will make without acknowledgment before giving up, reporting the peer unusable and closing the association.
- sasoc_number_peer_destinations holds the number of peer destination addresses. It cannot be set, only retrieved.
- sasoc_peer_rwnd holds the peer's current calculated receive window. This value represents the total number of data bytes that can yet be sent. This field is dynamic; as the local endpoint sends data, this value decreases. As the remote application reads datathat has been received, this value increases. This value cannot be changed by this socketoption call.
- sasoc_local_rwnd represents the local receive window the SCTP stack is currently reporting to the peer. This value is dynamic as well and is influenced by the SO_SNDBUF socket option. This value cannot be changed by this socket option call.
- sasoc_cookie_life represents the number of milliseconds for which a cookie, given to a remote peer, is valid. Each state cookie sent to a peer has a lifetime associated with it to prevent replay attacks. The default value of 60,000 milliseconds can be changed by setting this option with a sasoc_assoc_idvalue of 0.

We will provide advice on tuning the value of sasoc_asocmaxrxt for performance in The sasoc_cookie_life can be reduced for greater protection against cookie replay attacks but less robustness to network delay during association initiation. The other values are useful for debugging.

SCTP_AUTOCLOSE Socket Option:

This option allows us to fetch or set the auto close time for an SCTP endpoint. The auto close time is the number of seconds an SCTP association will remain open when idle. Idle is definedby the SCTP stack as neither endpoint sending or receiving user data. The default is for the auto close function to be disabled.

The auto close option is intended to be used in the one-to-many-style SCTP interface (Chapter9). When this option is set, the integer passed to the option is the number of seconds before an idle connection should be closed; a value of 0 disables autoclose. Only future associations created by this endpoint will be affected by this option; existing associations retain their current setting.

Auto close can be used by a server to force the closing of idle associations without the server needing to maintain additional state. A server using this feature needs to carefully assess the longest idle time expected on all its associations. Setting the autoclose value smaller than needed results in the premature closing of associations.

SCTP_DEFAULT_SEND_PARAM Socket Option:

SCTP has many optional send parameters that are often passed as ancillary data or used with the sctp_sendmsg function call (which is often implemented as a library call that passes ancillary data for the user). An application that wishes to send a large number of messages, all with the same parameters, can use this option to set up the default parameters and thus avoid using ancillary data or the sctp_sendmsg call. This option takes as input the sctp_sndrcv info structure.

```
struct sctp_sndrcvinfo
{       u_int16_t       sinfo_stream;
   u_int16_t  sinfo_ssn;  u_int16_t
   sinfo_flags;               u_int32_t
   sinfo_ppid;                u_int32_t
   sinfo_context;             u_int32_t
   sinfo_timetolive;          u_int32_t
   sinfo_tsn;                 u_int32_t
   sinfo_cumtsn;          sctp_assoc_t
   sinfo_assoc_id;
};
```

These fields are defined as follows:

- sinfo_stream specifies the new default stream to which all messages will be sent.
- sinfo_ssn is ignored when setting the default options. When receiving a message with therecvmsg function or sctp_recvmsg function, this field will hold the value the peer placed in the stream sequence number (SSN) field in the SCTP DATA chunk.

sinfo_flags dictates the default flags to apply to all future message sends. Allowable

-

SCTP_DISABLE_FRAGMENTS Socket Option:

SCTP normally fragments any user message that does not fit in a single SCTP packet into multiple DATA chunks. Setting this option disables this behavior on the sender. When disabled by this option, SCTP will return the error EMSGSIZE and not send the message. The default behavior is for this option to be disabled; SCTP will normally fragment user messages.

This option may be used by applications that wish to control message sizes, ensuring that every user application message will fit in a single IP packet. An application that enables this option must be prepared to handle the error case (i.e., its message was too big) by either providing application-layer fragmentation of the message or a smaller message.

SCTP_EVENTS Socket Option:

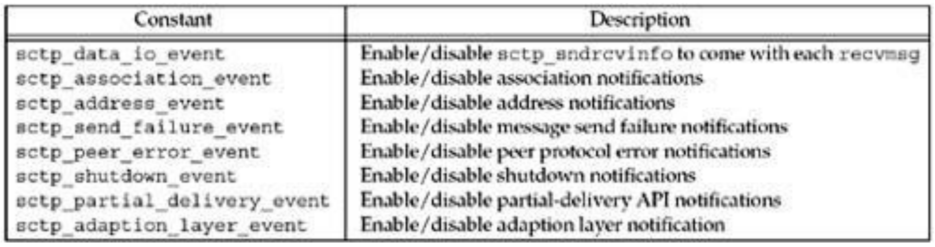
This socket option allows a caller to fetch, enable, or disable various SCTP notifications. An SCTP notification is a message that the SCTP stack will send to the application. The message is read as normal data, with the msg_flags field of the recvmsg function being set to MSG_NOTIFICATION. An application that is not prepared to use either recvmsg or sctp_recvmsg should not enable events. Eight different types of events can be subscribed to by using this option and passing an sctp_event_subscribe structure. Any value of 0 represents a non- subscription and a value of 1 represents a subscription.

Thesctp_event_subscribe structure takes the following form:

```
struct  sctp_event_subscribe  {  u_int8_t
   sctp_data_io_event;             u_int8_t
   sctp_association_event;         u_int8_t
   sctp_address_event;             u_int8_t
   sctp_send_failure_event;        u_int8_t
   sctp_peer_error_event;          u_int8_t
   sctp_shutdown_event;
   u_int8_t        sctp_partial_delivery_event;
   u_int8_t sctp_adaption_layer_event;
```

Figure 7.17 summarizes the various events. Further details on notifications can be found in Section 9.14.

Figure 7.17. SCTP event subscriptions.



Constant	Description
sctp_data_io_event	Enable/disable sctp_sndrcvinfo to come with each recvmsg
sctp_association_event	Enable/disable association notifications
sctp_address_event	Enable/disable address notifications
sctp_send_failure_event	Enable/disable message send failure notifications
sctp_peer_error_event	Enable/disable peer protocol error notifications
sctp_shutdown_event	Enable/disable shutdown notifications
sctp_partial_delivery_event	Enable/disable partial-delivery API notifications
sctp_adaption_layer_event	Enable/disable adaption layer notification

SCTP_GET_PEER_ADDR_INFO Socket Option:

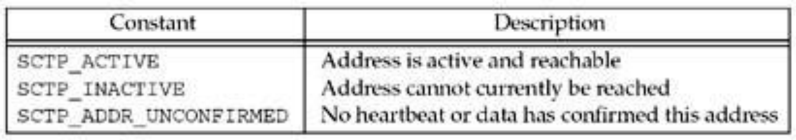
This option retrieves information about a peer address, including the congestion window, smoothed RTT and MTU. This option may only be used to retrieve information about a specific peer address. The caller provides a sctp_paddrinfo structure with the spinfo_address field filled in with the peer address of interest, and should use sctp_opt_info instead of getsockopt for maximum portability. The sctp_paddrinfo structure has the following format:

```
struct        sctp_paddrinfo        {
   sctp_assoc_t spinfo_assoc_id;
   struct  sockaddr_storage  spinfo_address;  int32_t
   spinfo_state;
   u_int32_t    spinfo_cwnd;
   u_int32_t      spinfo_srtt;
   u_int32_t       spinfo_rto;
   u_int32_t spinfo_mtu;
};
```

The data returned to the caller provides the following:

- spinfo_assoc_id contains association identification information, also provided in the "communication up" notification (SCTP_COMM_UP). This unique value can be used as a shorthand method to represent the association for almost all SCTP operations.
- spinfo_address is set by the caller to inform the SCTP socket on which address to return information. On return, its value should be unchanged.
- spinfo_state holds one or more of the values seen

Figure 7.18. SCTP peer address states.



Constant	Description
SCTP_ACTIVE	Address is active and reachable
SCTP_INACTIVE	Address cannot currently be reached
SCTP_ADDR_UNCONFIRMED	No heartbeat or data has confirmed this address

An*unconfirmed address* is one that the peer had listed as a valid address, but the local SCTP endpoint has not been able to confirm that the peer holds that address. An SCTP endpoint confirms an address when a heartbeat or user data, sent to that address, is acknowledged. Note that an unconfirmed address will also not have a valid retransmission timeout (RTO) value. Active addresses represent addresses that are considered available for use.

- spinfo_cwnd represents the current congestion window recorded for the peer address. A description of how the the cwnd value is managed can be found on page 177 of [Stewart and Xie 2001].
- spinfo_srtt represents the current estimate of the smoothed RTT for this address.
- spinfo_rto represents the current retransmission timeout in use for this address.
- spinfo_mturepresents the current path MTU as discovered by path MTU discovery.

One interesting use for this option is to translate an IP address structure into an association identification that can be used in other calls. We will illustrate the use of this socket option in Another possibility is for the application to track performance to each address of a multihomed peer and update the primary address of the association to the peer's best address.These values are also useful for logging and debugging.

SCTP_I_WANT_MAPPED_V4_ADDR Socket Option:

This flag can be used to enable or disable IPv4-mapped addresses on an AF_INET6-type socket. Note that when enabled (which is the default behavior), all IPv4 addresses will be mapped to a IPv6 address before sending to the application. If this option is disabled, the SCTP socket will *not* map IPv4 addresses and will instead pass them as a sockaddr_in structure.

acknowledged. Note that an unconfirmed address will also not have a valid retransmission timeout (RTO) value. Active addresses represent addresses that are considered available for use.

- spinfo_cwnd represents the current congestion window recorded for the peer address. A description of how the the cwnd value is managed can be found on page 177 of [Stewart and Xie 2001].
- spinfo_srtt represents the current estimate of the smoothed RTT for this address.
- spinfo_rto represents the current retransmission timeout in use for this address.
- spinfo_mturepresents the current path MTU as discovered by path MTU discovery.

One interesting use for this option is to translate an IP address structure into an association identification that can be used in other calls. We will illustrate the use of this socket option in Another possibility is for the application to track performance to each address of a multihomed peer and update the primary address of the association to the peer's best address.These values are also useful for logging and debugging.

SCTP_I_WANT_MAPPED_V4_ADDR Socket Option:

This flag can be used to enable or disable IPv4-mapped addresses on an AF_INET6-type socket. Note that when enabled (which is the default behavior), all IPv4 addresses will be mapped to a IPv6 address before sending to the application. If this option is disabled, the SCTPsocket will *not* map IPv4 addresses and will instead pass them as a sockaddr_in structure.

SCTP_INITMSG Socket Option:

This option can be used to get or set the default initial parameters used on an SCTP socket when sending out the INIT message. The option uses the sctp_initmsg structure, which is defined as:

```
struct   sctp_initmsg    {   uint16_t
   sinit_num_ostreams;
   uint16_t       sinit_max_instreams;
   uint16_t        sinit_max_attempts;
   uint16_t sinit_max_init_timeo;
};
```

SCTP_MAXBURST Socket Option:

This socket option allows the application to fetch or set the *maximum burst size* used when sending packets. When an SCTP implementation sends data to a peer, no more than SCTP_MAXBURST packets are sent at once to avoid flooding the network with packets. An implementation may apply this limit by either: (i) reducing its congestion window to the current flight size plus the maximum burst size times the path MTU, or (ii) using this value as a separate micro-control, sending at most maximum burst packets at any single send opportunity.

SCTP_MAXSEG Socket Option:

This socket option allows the application to fetch or set the *maximum fragment size* used during SCTP fragmentation. This option is similar to the TCP option TCP_MAXSEG described in

When an SCTP sender receives a message from an application that is larger than this value, the SCTP sender will break the message into multiple pieces for transport to the peer endpoint. The size that the SCTP sender normally uses is the smallest MTU of all addresses associated with the peer. This option overrides this value downward to the value specified. Note that the SCTP stack may fragment a message at a smaller boundary than requested by this option. Thissmaller fragmentation will occur when one of the paths to the peer endpoint has a smaller MTU than the value requested in the SCTP_MAXSEGoption.

This value is an endpoint-wide setting and may affect more than one association in the one-to-many interface style.

SCTP_NODELAY Socket Option:

If set, this option disables SCTP's *Nagle algorithm* . This option is OFF by default (i.e., the Nagle algorithm is ON by default). SCTP's Nagle algorithm works identically to TCP's except that it is trying to coalesce multiple DATA chunks as opposed to simply coalescing bytes on a stream.
For a further discussion of the Nagle algorithm, see TCP_MAXSEG.

SCTP_PEER_ADDR_PARAMS Socket Option:

This socket option allows an application to fetch or set various parameters on an association. The caller provides the sctp_paddrparams structure, filling in the association identification. The sctp_paddrparam structure has the following format:

```
struct    sctp_paddrparams    {
  sctp_assoc_t spp_assoc_id;
  struct    sockaddr_storage    spp_address;
  u_int32_t spp_hbinterval;
  u_int16_t spp_pathmaxrxt;
};
```

These fields are defined as follows:

- spp_assoc_id holds the association identification for the information being requested or set. If this value is set to 0, the endpoint default values are set or retrieved instead of the association-specific values.

- spp_address specifies the IP address for which these parameters are being requested or set. If the spp_assoc_idfield is set to 0, then this field is ignored.

- spp_hbinterval is the interval between heartbeats. A value of SCTP_NO_HB disables heartbeats. A value of SCTP_ISSUE_HB requests an on-demand heartbeat. Any other value changes the heartbeat interval to this value in milliseconds. When setting the default parameters, the value of SCTP_ISSUE_HB is not allowed.
- spp_hbpathmaxrxt holds the number of retransmissions that will be attempted on this destination before it is declared INACTIVE. When an address is declared INACTIVE, if it is the primary address, an alternate address will be chosen as the primary.

SCTP_PRIMARY_ADDR Socket Option :

This socket option fetches or sets the address that the local endpoint is using as primary. The primary address is used, by default, as the destination address for all messages sent to a peer.To set this value, the caller fills in the association identification and the peer's address that should be used as the primary address. The caller passes this information in a sctp_setprim structure, which is defined as:

```
struct sctp_setprim {
   sctp_assoc_t                    ssp_assoc_id;
   struct sockaddr_storage ssp_addr;
};
```

These fields are defined as follows:

- spp_assoc_id specifies the association identification on which the requester wishes to set or retrieve the current primary address. For the one-to-one style, this field is ignored.
- sspp_addr specifies the primary address, which must be an address belonging to the peer. If the operation is a setsockopt function call, then the value in this field represents the new peer address the requester would like to be made into the primary destination address. Note that retrieving the value of this option on a one-to-one socket that has only one local address associated with it is the same as calling getsock name.

SCTP_RTOINFO Socket Option:

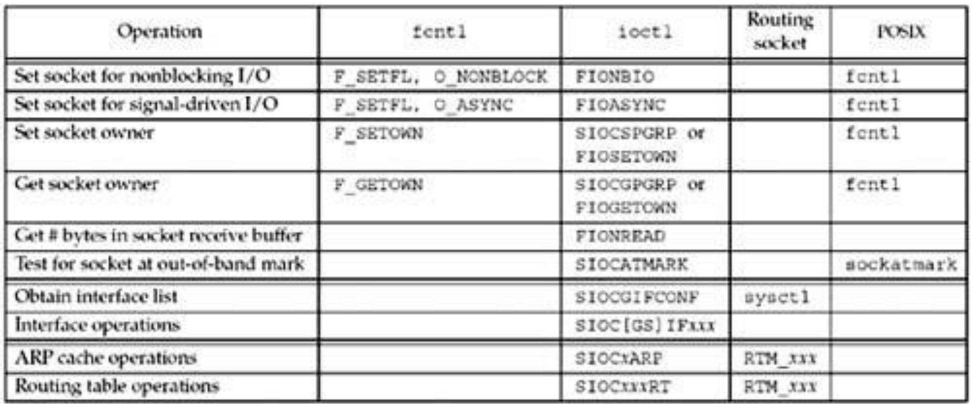
This socket option can be used to fetch or set various RTO information on a specific association or the default values used by this endpoint. When fetching, the caller should use sctp_opt_info instead of getsockopt for maximum portability. The caller provides a sctp_rtoinfo structure of the following form:

```
struct sctp_rtoinfo
{
    sctp_assoc          srto_assoc_id;
    uint32_t            srto_initial;
    uint32_t            srto_max;
    uint32_t            srto_min;
};
```

Fnctl Functions :

fcntl stands for "file control" and this function performs various descriptor control operations. Before describing the function and how it affects a socket, we need to look at the bigger picture. summarizes the different operations performed by fcntl,ioctl, and routing sockets.

Figure 7.20. Summary of fcntl,ioctl, and routing socket operations.



Operation	fcntl	ioctl	Routing socket	POSIX
Set socket for nonblocking I/O	F_SETFL, O_NONBLOCK	FIONBIO		fcntl
Set socket for signal-driven I/O	F_SETFL, O_ASYNC	FIOASYNC		fcntl
Set socket owner	F_SETOWN	SIOCSPGRP or FIOSETOWN		fcntl
Get socket owner	F_GETOWN	SIOCGPGRP or FIOGETOWN		fcntl
Get # bytes in socket receive buffer		FIONREAD		
Test for socket at out-of-band mark		SIOCATMARK		sockatmark
Obtain interface list		SIOCGIFCONF	sysctl	
Interface operations		SIOC[GS]IFxxx		
ARP cache operations		SIOCxARP	RTM_xxx	
Routing table operations		SIOCxxxRT	RTM_xxx	

The first six operations can be applied to sockets by any process; the second two (interface operations) are less common, but are still general-purpose; and the last two (ARP and routing table) are issued by administrative programs such as ifconfig and route. We will talk more about the various ioctl operations in Chapter 17 and routing sockets in Chapter 18.

There are multiple ways to perform the first four operations, but we note in the final column that POSIX specifies that fcntl is the preferred way. We also note that POSIX provides the sockatmark function (Section 24.3) as the preferred way to test for the out-of-band mark. The remaining operations, with a blank final column, have not been standardized by POSIX.

> We also note that the first two operations, setting a socket for nonblocking I/O and for signal-driven I/O, have been set historically using the FNDELAY and FASYNC commands withfcntl. POSIX defines the O_*XXX* constants.

Thefcntl function provides the following features related to network programming:

- Nonblocking I/O— We can set the O_NONBLOCKfile status flag using the F_SETFL command to set a socket as nonblocking. We will describe nonblocking I/O in Chapter 16. variable that stores this ID is the so_pgid member of the socket structure (p. 438 ofTCPv2).

Each descriptor (including a socket) has a set of file flags that is fetched with the F_GETFL command and set with the F_SETFLcommand. The two flags that affect a socket are

- O_NONBLOCK—nonblocking I/O
- O_ASYNC—signal-driven I/O

We will describe both of these features in more detail later. For now, we note that typical codeto enable nonblocking I/O, using fcntl, would be:

```
int          flags;

        /* Set a socket as nonblocking */
if  ( (flags = fcntl (fd, F_GETFL, 0)) < 0)err_sys("F_GETFL error");
flags |= O_NONBLOCK;
if    (fcntl(fd,    F_SETFL,    flags)    <    0)
      err_sys("F_SETFL error");
```

Beware of code that you may encounter that simply sets the desired flag.

```
      /* Wrong way to set a socket as nonblocking */ if
(fcntl(fd, F_SETFL, O_NONBLOCK) < 0)
      err_sys("F_SETFL error");
```

While this sets the nonblocking flag, it also clears all the other file status flags. The only correct way to set one of the file status flags is to fetch the current flags, logically OR in the new flag, and then set the flags.

The following code turns off the nonblocking flag, assuming flagswas set by the call to fcntl shown above:

```
flags &= ~O_NONBLOCK;
```

- Signal-driven I/O— We can set the O_ASYNC file status flag using the F_SETFL command, which causes the SIGIO signal to be generated when the status of a socket changes. We will discuss this in Chapter 25.

- TheF_SETOWN command lets us set the socket owner (the process ID or process group ID) to receive the SIGIO and SIGURG signals. The former signal is generated when signal-driven I/O is enabled for a socket (Chapter 25) and the latter signal is generated when new out-of-band data arrives for a socket