I/O Multiplexing

- Consider an example where a TCP client is handling two inputs at the same time: standard input and a TCP socket.
- It is possible for the client to blocked on a call to read (by calling the ***readline*** function), and the server process will timeout.
- The server TCP correctly sends a FIN to the client TCP, but since the client process is blocked reading from standard input, it never sees the end-of-file until it reads from the socket (possibly much later in time).
- A solution to this is to tell the kernel that we want to be notified if one or more I/O conditions are ready (i.e., input is ready to be read, or the descriptor is capable of taking more output).
- This capability is called **I/0 multiplexing** and is provided by the ***select*** and ***poll*** functions.
- I/O multiplexing is typically used in networking applications in the following scenarios:
    - When a client is handling multiple descriptors (normally interactive input and a network socket), 1/0 multiplexing should be used.
    - It is possible, but rare, for a client to handle multiple sockets at the same time. An example of this is a Web client.
    - If a TCP server handles both a listening socket and its connected sockets, I/O multiplexing is normally.
    - If a server handles both TCP and UDP, I/O multiplexing is normally used.
    - If a server handles multiple services and perhaps multiple protocols (e.g., the ***inetd*** daemon), I/O multiplexing is normally used.

Blocking I/0 Model

- By default, all sockets are blocking. Consider the UDP example we looked at in our previous discussions.
- A process calls ***recvfrom*** and the system call does not return until the datagram arrives and is copied into our application buffer, or an error occurs.
- The process is **blocked** the entire time from when it calls ***recvfrom*** until it returns. When recvfrom returns, the application processes the datagram.

Nonblocking I/0 Model

- When we set a socket **nonblocking**, we are telling the kernel that when a requested I/O operation cannot be completed, return an error code instead of blocking.
- Now when we call ***recvfrom***, and there is no data to return, the kernel immediately returns an error of **EWOULDBLOCK** instead of blocking.
- This implies that we have to keep calling ***recvfrom*** until a datagram is received, copied into our application buffer.
- When an application sits in a loop calling ***recvfrom*** on a nonblocking descriptor like this, it is called **polling**.
- The application is continually polling the kernel to see if some operation is ready. This is often a waste of CPU time, but this model is occasionally encountered, normally on systems dedicated to one function.

I/O Multiplexing Model

- With I/O multiplexing, we call ***select*** or ***poll*** and block in one of these two system calls, instead of blocking in the actual I/O system call.
- We block in a call to ***select***, waiting for the datagram socket to be readable. When ***select*** returns that the socket is readable, we then call ***recvfrom*** to copy the datagram into our application buffer.
- On the surface there does not appear to be any advantage in doing this, and in fact there is a slight disadvantage because using ***select*** requires two system calls instead of one.
- As we shall see, the advantage in using ***select*** is that we can wait for more than one descriptor to be ready. In other words we can select from a **set** of socket descriptors.

Synchronous I/0 versus Asynchronous I/O

- Posix.1 defines these two terms as follows:
- A **synchronous I/O operation** causes the requesting process to be blocked until that I/O operation completes.
- An **asynchronous** I/O operation does not cause the requesting process to be blocked.
- Using these definitions the I/O models we have discussed, **blocking**, **nonblocking**, and **I/O multiplexing** all synchronous because the actual I/O operation (***recvfrom***) blocks the process.
- Only the true asynchronous I/O model defined by Posix ("**real-time**" extensions) matches the asynchronous I/O definition.

The *select* Function

- This function allows a process to specify to the kernel to notify the process when any one of multiple events occur or when a specified timeout expires.
- If an application is using several sockets then the ***select()*** function may be used to find out which ones are active, i.e. which ones have outstanding incoming data or which can now accept further data for transmission or which have outstanding exceptional conditions.
- This function would probably only be used with server daemon programs using multiplexing to handle multiple clients.
- For example, say we can call select and tell the kernel to return only when:
    - Any of the descriptors in the set {1, 4, 5} are ready for reading or
    - Any of the descriptors in the set {2, 7} are ready for writing or
    - Any of the descriptors in the set {1, 4} have an exception condition pending
- We tell the kernel what descriptors we are interested in (for reading, writing, or an exception condition) and how long to wait.
- The descriptors in which we are interested in testing can be any type of descriptor. Select is not restricted to socket descriptors.
- The syntax of the call is as follows:

```
#include &ltsys/time.h>
#include &ltsys/types.h>

int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct
                timeval *timeout);
```

- ***select()*** examines the I/O **file descriptor sets** whose addresses are passed in ***readfds***, ***writefds***, and ***exceptfds*** to see if any of their file descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively.

- Any of ***readfds***, ***writefds***, and ***exceptfds*** may be given as NULL pointers if no file descriptors are of interest.

- ***nfds*** is the number of bits to be checked in each bit mask that represents a file descriptor; the file descriptors from **0** to ***nfds -1*** in the file descriptor sets are examined.

- On return, ***select()*** replaces the given file descriptor sets with subsets consisting of those file descriptors that are ready for the requested operation.

- The return value from the call to ***select()*** is the number of ready file descriptors.

- If ***timeout*** is not a NULL pointer, it specifies a maximum interval to wait for the selection to complete.

- If timeout is a NULL pointer, the ***select()*** blocks indefinitely. To effect a poll, the ***timeout*** argument should be a non-NULL pointer, pointing to a zero-valued ***timeval*** structure.

- A ***timeval*** structure specifies the number of seconds and microseconds.

```
struct timeval
{
    long tv_sec;    //seconds
    long tv_usec;  //microseconds
}
```

- There are 3 possibilities:

  1. **Wait forever**: return only when one of the specified descriptors is ready for I/O. In this case, the ***timeout*** argument is specified as a null pointer.
  2. **Wait up to a fixed amount of time**: return when one of the specified descriptors is ready for I/O, but do not wait beyond the time specified in the ***timeval*** structure.
  3. Do not wait at all: Return immediately after checking the descriptors. This is called polling. This is done by setting the timer value to 0.

- A design issue to consider is how to specify one or more descriptor values for each of the three descriptor arguments.

- The ***select*** call uses descriptor sets, typically an **array of integers**, with each bit in each integer corresponding to a descriptor.

- For example, using 32-bit integers, the first element of the array corresponds to descriptors 0 through 31, the second element of the array corresponds to descriptors 32 through 63 and so on.

- The implementation details of this variable type are irrelevant as far as the application is concerned and are hidden in the **fd_set** datatype.

- ***select()*** is normally used in conjunction with the macros **FD_SET**, **FD_CLR**, **FD_ISSET** and **FD_ZERO**.

- Before calling ***select***, **FD_SET**, **FD_CLEAR** and **FD_ZERO** should be used to put the relevant descriptors into objects of type **fd_set**.

- For example:

```
void FD_SET(int fd, fd_set &fdset);  // turn on the bit for fd in fdset

void FD_CLR(int fd, fd_set &fdset);  // turn off the bit for fd in fdset

int FD_ISSET(int fd, fd_set &fdset);  // is the bit for fd on in fdset?

void FD_ZERO(fd_set &fdset);          // clear all bits in fdset
```

- For example, to define a variable of type **fd_set** and then turn on the bits for descriptors 1, 4, and 5, we write:

```
fd_set rset;

FD_ZERO(&rset);       // initialize the set: all bits off
FD_SET(1, &rset);     // turn on the bit for fd 1
FD_SET(4, &rset);     // turn on the bit for fd 4
FD_SET(5, &rset);     // turn on the bit for fd 5
```

- ***select*** modifies the descriptor sets pointed to by the **readset**, **writeset**, and **exceptset** pointers.

- When we call the function, we specify the values of the descriptors that we are interested in and upon return the result indicates which descriptors are ready.

- We use **FD_ISSET** to test a specific descriptor in an **fd_set** structure. Any descriptor that is not ready on return will have its corresponding bit cleared in the descriptor set.

- To handle this we turn on all the bits in which we are interested in all the descriptor sets each time we call ***select***.

- The return value from this function indicates the total number of bits that are ready across all descriptor sets.

Descriptor Conditions

- From the above discussion we know that a process will wait for a descriptor to become ready for I/O (reading or writing) or to have an exception condition pending on it (out-of-band data).

- I/O for regular files is relatively straightforward. For sockets however we must be more specific about the conditions that cause select to return "ready" for sockets. These conditions are described below.

- A socket is ready for **reading** if any of the following **four conditions** is **true**:

    1. The number of bytes of data in the socket receive buffer is greater than or equal to the current size of the low-water mark for the socket receive buffer.

       We can set this low-water mark using the **SO_RCVLOWAT** socket option. It defaults to I for TCP and UDP sockets.

    2. The read-half of the connection is closed (i.e., a TCP connection that has received a **FIN**). A read operation on the socket will not block and will return 0 (i.e., end-of-file).

    3. The socket is a listening socket and the number of completed connections is nonzero.

    4. A socket error is pending. A read operation on the socket will not block and will return an error (-1) with ***errno*** set to the specific error condition.

       These **pending errors** can also be fetched and cleared by calling ***getsockopt*** specifying the **SO_ERROR** socket option.

- A socket is ready for **writing** if any of the following **three conditions** is **true**:

    5. The number of bytes of available space in the socket send buffer is greater than or equal to the current size of the low-water mark for the socket send buffer **and** either (i) the socket is connected, or (ii) the socket does not require a connection (e.g., UDP).

       This means that if we set the socket **nonblocking**, a write operation will not block and will return a positive value (bytes received).

       We can set this low-water mark using the **SO_SNDLOWAT** socket option. This low-water mark normally defaults to 2048 for TCP and UDP sockets.

    6. The write-half of the connection is closed. A write operation on the socket will generate **SIGPIPE**.

    7. A socket error is pending. A write operation on the socket will not block and will return an error (-1) with ***errno*** set to the specific error condition.

       These pending errors can also be fetched and cleared by calling ***getsockopt*** for the **SO_EPROR** socket option.

- A socket has an **exception condition pending** if there exists **out-of-band data** for the socket or the socket is still at the **out-of-band mark**.

- Note that when an error occurs on a socket it is marked as both **readable** and **writable** by ***select***.

- The purpose of the receive and send low-water marks is to give the application control over how much data must be available for reading or how much space must be available for writing before ***select*** returns readable or writable.

- For example, if we know that our application has nothing productive to do unless at least 64 bytes of data are present, we can set the receive low-water mark to 64 to prevent select from waking us up if less than 64 bytes are ready for reading.

- As long as the send low-water mark for a UDP socket is less than the send buffer size (which should always be the default relationship), the UDP socket is always writable, since a connection is not required.

TCP Example using *select*

- We will use the basic **echo** server as an example. The server will be a single process that uses **select** to handle multiple clients.

- Let us first examine the data structures that are used to keep track of the clients. The first diagram shown shows the state of the server with just one listening socket open.

- The server maintains a read descriptor set as shown in the next diagram. We assume that the server is started in the foreground, so descriptors **fd0**, **fd1**, and **fd2** are set to standard **input**, **output**, and **error**.

- Therefore the first available descriptor for the listening socket is **fd3**.

- Also shown is an array of integers named **client** that contains the connected socket descriptor for each client. All elements in this array are initialized to -1.

- The only nonzero entry in the descriptor set is the entry for the **listening socket** and the first argument to ***select*** will be ***4***.

- When the first client establishes a connection with our server, the listening descriptor becomes readable and our server calls ***accept***.

- The new connected descriptor returned by ***accept*** will be ***fd4***, given the assumptions of this example. The next diagram shows the connection from the client to the server.

- From this point on the server must remember the new connected socket in its ***client*** array, and the connected socket must be **added to the descriptor set**. The updated data structures are shown.

- No assume that some time later a second client establishes a connection and we have the scenario shown. The new connected socket (which we assume is ***fd5***) must be stored, giving the updated data structures shown.

- Next we assume the first client terminates its connection. The client TCP sends a **FIN**, which makes descriptor 4 in the server readable.

- We then close this socket and update our data structures accordingly The value of ***client[0]*** is set to ***-1*** and descriptor ***fd4*** in the descriptor set is set to ***0***.

- This is shown in the diagram. Notice that the value of ***maxfd*** does not change.

- To summarize, as clients arrive we record their connected socket descriptor in the first available entry in the **client** array (i.e., the first entry with a value of -1).

- We must also **add** the connected socket to the **read descriptor set**. The variable maxi is the highest index in the **client** array that is currently in use and the variable ***maxfd*** (plus one) is the current value of the first argument to ***select***.

- The only limit on the number of clients that this server can handle is the minimum of the two values **FD_SETSIZE** and the maximum number of descriptors allowed for this process by the kernel

The poll Function

- The ***poll*** function originated with SVR3 and was originally limited to streams devices. SVR4 removed this limitation, allowing ***poll*** to work with any descriptor.

- ***poll*** provides functionality that is similar to select, but poll provides additional information when dealing with streams devices.

```
#include <poll.h>

int poll (struct pollfd *fdarray, unsigned long nfds, int timeout)
```

- The function returns one of the following:
  - count of ready descriptors
  - 0 on timeout
  - -1 on error

- The first argument is a pointer to the first element of an array of structures. Each element of the array is a ***pollfd*** structure that specifies the conditions to be tested for a given descriptor ***fd***.

- The structure is defined as follows:

```
struct pollfd
{
    int fd;          // descriptor to check
    short events;    // events of interest on fd
    short revents;   // events that occurred on fd
}
```

- The conditions to be tested are specified by the **events** member, and the function returns the status for that descriptor in the corresponding **revents** member.

- Each of these two members is composed of one or more bits that specify a certain condition. The table below shows the constants used to specify the **events** flag and to test the **revents** flag against.

Constant	Input to *events*?	Result from *revents*?	Description
POLLIN	♦	♦	normal or priority band data can be read
POLLRDNORM	♦	♦	normal data can be read
POLLRDBAND	♦	♦	priority band data can be read
POLLPRI	♦	♦	high-priority data can be read
POLLOUT	♦	♦	normal data can be written
POLLWRNORM	♦	♦	normal data can be written
POLLWRBAND	♦	♦	priority band data can be written
POLLERR		♦	an error has occurred
POLLHUP		♦	hangup has occurred
POLLNVAL		♦	descriptor is not an open file

- The first four constants deal with input, the next three deal with output, and the final three deal with errors.
- Note that the final three cannot be set in events but are always returned in ***revents*** when the corresponding condition exists.
- There are three classes of data identified by ***poll***: **normal, priority band,** and **high** priority.
- With regard to TCP and UDP sockets, the following conditions cause ***poll*** to return the specified ***revent.***
- Not however that Posix.1g leaves many optional ways to return the same condition in its definition of ***poll***:
    - All regular TCP data and all UDP data is considered normal.
    - TCP's out-of-band data is considered priority band.
    - When the read-half of a TCP connection is closed (e.g., a FIN is received), this is also considered normal data and a subsequent read operation will return 0.
    - The presence of an error for a TCP connection can be considered either normal data or an error (**POLLERR**).
    - In either case a subsequent read will return -1 with ***errno*** set to the appropriate value. This handles conditions such as the receipt of an **RST** or a **timeout**.
    - The availability of a new connection on a listening socket can be considered either normal data or priority data. Most implementations consider this normal data.
- The number of elements in the array of structures is specified by the ***nfds*** argument.
- The ***timeout*** argument specifies how long the function is to wait before returning.

- A positive value specifies the number of milliseconds to wait. The possible values for the *timeout* argument are as follows:

  - **INFTIM** - wait forever
  - **0** - return immediately, do not block
  - **> 0** wait specified number of milliseconds

- The constant **INFTIM** is defined to be a negative value. If the system does not provide a timer with millisecond accuracy, the value is rounded up to the nearest supported value.

- The return value from ***poll*** is -1 if an error occurred, 0 if no descriptors are ready before the timer expires, otherwise it is the number of descriptors that have a nonzero ***revents*** member.

- If we are no longer interested in a particular descriptor, we just set the ***fd*** member of the ***pollfd*** structure to a negative value. Then the ***events*** member is ignored and the ***revents*** member is set to 0 on return.

TCP Example using *poll*

- We will again use the basic **echo** server as an example. We now redo our TCP echo server using poll instead of select.

- In the previous version using select we had to allocate a client array along with a descriptor set named ***rset***.

- With poll we must allocate an array of ***pollfd*** structures and use it to maintain the client information, instead of allocating another array.

- We handle the ***fd*** member of this array the same way we handled the client array in the previous example.

- A value of -1 means the entry is not in use; otherwise it is the descriptor value. Recall from the previous section that any entry in the array of ***pollfd*** structures passed to poll with a negative value for the ***fd*** member is just ignored.

- We use the first entry in the client array for the listening socket, and set the descriptor for the remaining entries to -1.

- We also set the **POLLRDNORM** event for this descriptor, to be notified by ***poll*** when a new connection is ready to be accepted.

- We call poll to wait for either a new connection or data on an existing connection. When a new connection is accepted, we find the first available entry in the client array by looking for the first one with a negative descriptor.

- Note that we start the search with the index of 1, since ***client[0]*** is used for the listening socket. When an available entry is found, we save the descriptor and set the **POLLRDNORM** event.

- The two return events that we check for are **POLLRDNORM** and **POLLERR**. The second of these we did not set in the event member, because it is always returned when the condition is true.

- The reason we check for **POLLERR** is because some implementations return this event when an RST is received for a connection, while others just return **POLLRDNORM**.

- When an existing connection is terminated by the client, we just set the ***fd*** member to -1.

Socket Options

- There are various ways to get and set the options that affect a socket:
  - The ***getsockopt*** and ***setsockopt*** functions
  - The ***fcntl*** function
  - The ***ioctl*** function.

- We will focus on the ***setsockopt*** and ***getsockopt*** functions.
- The **getsockopt and setsockopt functions** have the following prototypes:

  #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, and -1 on error.
- ***sockfd*** must refer 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 code (e.g., lPv4, lPv6, or TCP).
- ***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 ***setsockopt*** and as a value-result for ***getsockopt***.
- It is very good idea to determine whether or not your system supports all the various options before designing an application using this call.
- Your text provides an sample program to check whether most of the options defined for these calls supported, and if so, print their default value.

Socket States

- For some socket options there are timing considerations about when to set or fetch the option versus the state of the socket.

- The following socket options are **inherited** by a **connected** TCP socket from the **listening** socket:

    - **SO_DEBUG**
    - **SO_DONTROUTE**
    - **SO_KEEPALIVE**
    - **SO_LINGER**
    - **SO_OOBINLINE**
    - **SO_RCVBUF**
    - **SO_SNDBUF**

- This is important with TCP because the connected socket is not returned to a server by accept until the **three-way handshake** is completed by the TCP layer.

- If we want to ensure that one of these socket options is set for the connected socket when the three-way handshake completes, we must set that option for the listening socket.

Generic Socket Options

- These options are protocol dependent (that is, they are handled by the protocol-independent code within the kernel).

- 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 **UDP** 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 the concept of a broadcast message (e.g., Ethernet, token ring, etc.).

    - 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.

- **SO_DEBUG** Socket Option

    - This option is supported only by TCP.

    - When enabled for a TCP socket, the kernel keeps track of detailed information about all the packets sent or received by TCP for the socket.

- **SO_DONTROUTE** Socket Option
  - This option specifies that outgoing packets are to bypass the normal routing mechanisms of the underlying protocol.
  - For example, with lPv4, 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 not on a shared network), **ENETUNREACH** is returned.
- **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 Exxx 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, for either readability or writability, select returns with either or both conditions set.

    2 . If the process is using signal-driven I/), the SIGIO signal is generated for either the process or the process group.

- **SO_KEEPALIVE** Socket Option
  - When the keepalive option is set for a TCP socket and no data has been exchanged across the socket in either direction for 2 hours, TCP automatically sends a keepalive probe to the peer.
  - This probe is a TCP packet 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 OK). TCP will send another probe following another 2 hours of inactivity

    2. The peer responds with an 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.
    3. There is no response from the peer to the keepalive probe.

       Berkeley-derived TCPs send eight additional probes, 75 seconds apart, trying to elicit a response.

       TCP will give up if there is no response within 11 minutes and 15 seconds after sending the first probe.

If there is no response at all to TCP's keepalive probes, the socket's pending error is set to ETIMEDOUT and the socket is closed.

If the socket receives an ICMP error in response to one of the keepalive probes, the corresponding error is returned instead (and the socket is still closed).

- **SO_LINGER** Socket Option
  - This option specifies how the ***close*** function operates for a connection-oriented protocol (e.g., for TCP but not for UDP).
  - 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 to the peer.
  - This socket option lets us change this default. This option requires the following structure to be passed between the user process and the kernel. It is defined by including <sys/socket. h>.

```
struct linger
{
   int l_onoff;       // 0 = off, nonzero = on
   int l_linger;    // linger time, Posix.1g specifies units as seconds
}
```

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

1. If ***l_onoff*** is 0, the option is turned off. The value of ***l_linger*** is ignored and the previously discussed TCP default 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 sends an RST to the peer, not the normal four packet connection termination sequence.

3. If ***l_onoff*** is nonzero and ***l_linger*** is nonzero, then the kernel will **linger** when the socket is closed.

   If there is any data still remaining in the socket send buffer, the process is put to sleep until either (a) all the data is sent and acknowledged by the peer TCP, or (b) the linger time expires.

   If the socket has been set nonblocking , it will not wait for the close to complete, even if the linger time is nonzero.

- **SO_OOBINLINE** Socket Option
  - When this option is set, out-of-band data will be placed in the normal input queue (i.e., inline).
  - When this occurs, the **MSG_OOB** flag to the receive functions cannot be used to read the out-of-band data.

- **SO_RCVBUF** and **SO_SNDBUF** Socket Options
  - Using these two socket options we can change the default sizes of the TCP and UDP receive buffers. In effect we can change the default packet sizes.
  - The default values differ widely between implementations. Older Berkeley-derived implementations would default the TCP send and receive buffers to 4096 bytes, but newer systems use larger values, anywhere from 8192 to 61440 bytes.
  - The UDP send buffer size often defaults to a value around 9000 bytes if the host supports NFS, and the UDP receive buffer size often defaults to a value around 40000 bytes.

- **SO_RCVLOWAT** and **SO_SNDLOWAT** Socket Options
  - Every socket also has a receive low-water mark and a send low-water mark. These are used by the ***select*** function, as described earlier.
  - These two socket options let us change these two low-water marks.
  - The receive low-water mark is the amount of data that must be in the socket receive buffer for select to return "readable." It defaults to I 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 select to return "writable." This low-water mark normally defaults to 2048 for TCP sockets.

- **SO_RCVTIMEO** and **SO_SNDTIMEO** Socket Options
  - These two socket options allow us to place a timeout on socket **receives** and **sends**.
  - Recall one of the arguments to the two **sockopt** functions is a pointer to a ***timeval*** structure, the same one used with select.
  - This lets us specify the timeouts in seconds and microseconds. We disable a timeout by setting its value to 0 seconds and 0 microseconds. Both timeouts are disabled by default.
  - The receive timeout affects the five input functions: ***read***, ***readv***, ***recv***, *recvfrom*, and ***recvmsg***.

- The send timeout affects the five output functions: ***write***, ***writev***, ***send***, ***sendto***, and ***sendmsg***.

- **SO_REUSEADDR** and **SO_REUSEPORT** Socket Options

  - The **SO_REUSEADDR** socket option serves four different purposes.

    1. **SO-REUSEADDR** allows a listening server to start and bind its well-known port even if previously established connections exist that use this port as their local port.

    2. **SO_REUSEADDR** allows multiple instances of the same server to be started on the same port, as long as each instance binds a different local IP address.

       This is common for a site hosting multiple HTTP servers using the IP alias technique.

       Assume the local host's primary IP address is 198.69.10.2 but it has two aliases of 198.69.10.128 and 198.69.10.129.

       Three HTTP servers are started. The first HTTP server would call bind with a local IP address of 198.69.10.128 and a local port of 80.

       The second server would bind 198.69.10.129 and port 80. But this second call to bind fails unless SO_REUSEADDR is set before the call.

       The third server would be to call bind with the wildcard as the local IP address and a local port of 80. Again, SO_REUSEADDR is required for this final call to succeed.

       Assuming SO_REUSEADDR is set and the three servers are started, incoming TCP connection requests with a destination IP address of 198.69.10.128 and a destination port of 80 are delivered to the first server.

       Incoming requests with a destination IP address of 198.69.10.129 and a destination port of 80 are delivered to the second server.

       All other TCP connection requests with a destination port of 80 are delivered to the third server. This final server handles requests destined for 198.69.10.2 in addition to any other IP aliases that the host may have configured.

    3. SO_REUSEADDR allows a single process to bind the same port to multiple sockets, as long as each bind specifies a different local IP address.

       This is common for UDP servers that need to know the destination IP address of client requests on systems that do not provide the

IP_RECVDSTADDR socket option.

4. SO_REUSEADDR allows completely duplicate bindings: a bind of an IP address and port, when that same IP address and port are already bound to another socket.

   Normally this feature is supported only on systems that support multicasting, when that system does not support the SO_REUSEPORT socket option and only for UDP sockets.

- 4.4BSD introduced the **SO_REUSEPORT** socket option when support for multicasting was added.
- Instead of overloading SO_REUSEADDR with the desired multicast semantics that allow completely duplicate bindings, this new socket option was introduced with the following semantics:

  1. This option allows completely duplicate bindings but only if each socket that wants to bind the same IP address and port specify this socket option.
  2. SO_REUSEADDR is considered equivalent to SO_REUSEPORT if the IP address being bound is a multicast address.

- The problem with this socket option is that not all systems support it.

- **SO_TYPE** Socket Option
  - This option returns the socket type. The integer value returned is a value such as SOCK_STREAM or SOCK_DGRAM.
  - This option is typically used by a process that inherits a socket when it is started.

- **SO_USELOOPBACK** Socket Option
  - This option applies only to sockets in the routing domain (**AF_ROUTE**).
  - This option defaults on for these sockets.
  - When this option is enabled, the socket receives a copy of everything sent.

Ipv4 Socket Options

- These socket options are processed by lPv4 and have a ***level*** of **IPPROTO_IP.**

- **IP_HDRINCL**
  - If this option is set for a raw IP socket, we must build our own IP header for all the datagrams that 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 (***traceroute*** and ***ping***) that build their own IP header to override values that IP would place into certain header fields.

- **IP_OPTIONS** Socket Option
  - Setting this option allows us to set IP options in the lPv4 header.

- **IP_RECVDSTADDR**
  - This socket option causes the destination IP address of a received UDP datagram to be returned as ancillary data (essential data) by ***recvmsg***.

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

- **IP_TOS**
  - This option lets us set the type-of-service field in the IP header for a TCP or UDP socket.
  - If we call ***getsockopt*** for this option, the current value that would be placed into the TOS field in the IP header (which defaults to 0) is returned.
  - We can set the TOS to one of the constants shown below, which are defined by including <***netinet/ip. h***>:

Constant	Description
IPTOS_LOWDELAY	minimize delay
IPTOS_THROUGHPUT	maximize throughput
IPTOS_RELIABILITY	maximize reliability
IPTOS_LOWCOST	minimize cost

- **IP_TTL**
    - With this option we can set and fetch the default TTL (time-to-live field) that the system will use for a given socket.
    - 4.4BSD, for example, uses the default of 64 for both TCP and UDP sockets and 255 for raw sockets.

TCP Socket Options

- There are five socket options for TCP, but three are new with Posix.1g and not widely supported. The ***level*** is specified as **IPPROTO_TCP**.

- **TCP_KEEPALIVE** Socket Option
    - This option is new with Posix.1g. It specifies the idle time in seconds for the connection before TCP starts sending keepalive probes.
    - The default value must be at least 7200 seconds. This option is effective only when the **SO_KEEPALIVE** socket option is enabled.

- **TCP_MAXRT** Socket Option
    - This option is new with Posix.1g. It specifies the amount of time in seconds before a connection is broken once TCP starts retransmitting data.
    - A value of 0 means to use the system default, and a value of -1 means to retransmit forever. If a positive value is specified, it may be rounded up to the implementation's next retransmission time.

- **TCP_MAXSEG** Socket Option
    - This socket option allows us to fetch or set the **maximum segment size** (**MSS**) for a TCP connection.
    - The value returned is the maximum amount of data that our TCP will send to the other end; often it is the MSS announced by the other end with its SYN, unless our TCP chooses to use a smaller value than the peer's announced MSS.

- **TCP_NODELAY Socket** Option
    - If set, this option disables TCP's **Nagle algorithm** (See pp. 582 in Garcia & Widjaja). By default this algorithm is enabled.

- **TCP_STDURG** Socket Option
    - This option is new with Posix.1g and it affects the interpretation of TCP's urgent pointer.
    - By default the urgent pointer points to the data byte following the byte sent with the MSG_OOB flag.
    - This is how almost all implementations interoperate today.
    - But if this socket option is set nonzero, the urgent pointer will point to the data byte sent with the MSG_OOB flag.