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.