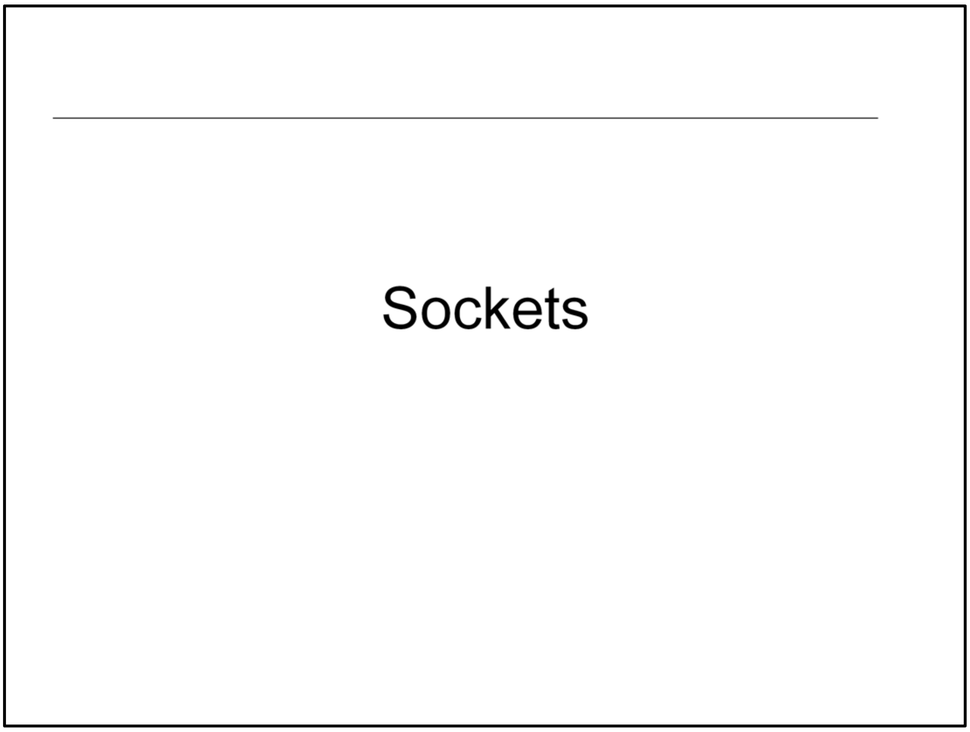
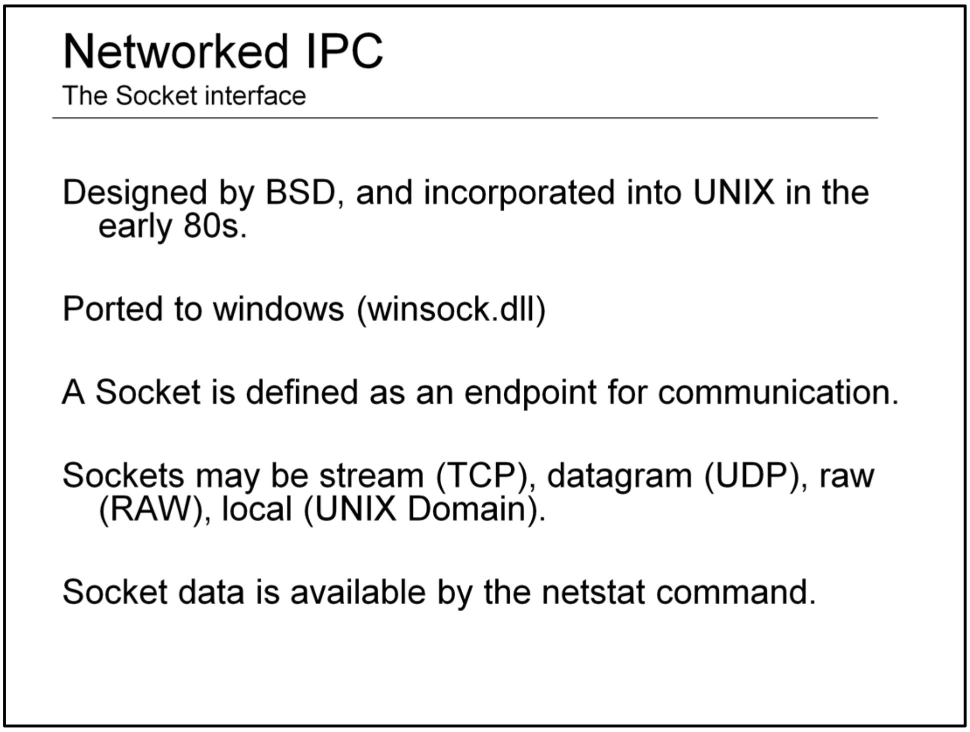


Sockets



Networked IPC

The Socket interface

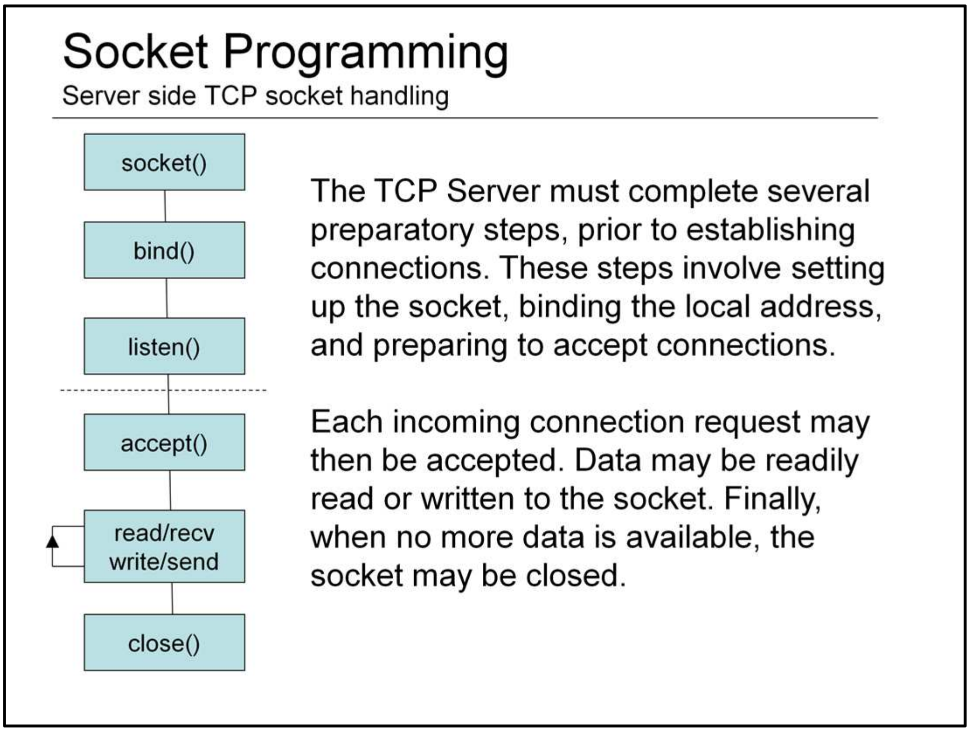
Designed by BSD, and incorporated into UNIX in the early 80s.

Ported to windows (winsock.dll)

A Socket is defined as an endpoint for communication.

Sockets may be stream (TCP), datagram (UDP), raw (RAW), local (UNIX Domain).

Socket data is available by the netstat command.



Socket Programming

Server side TCP socket handling

socket()

bind()

listen()

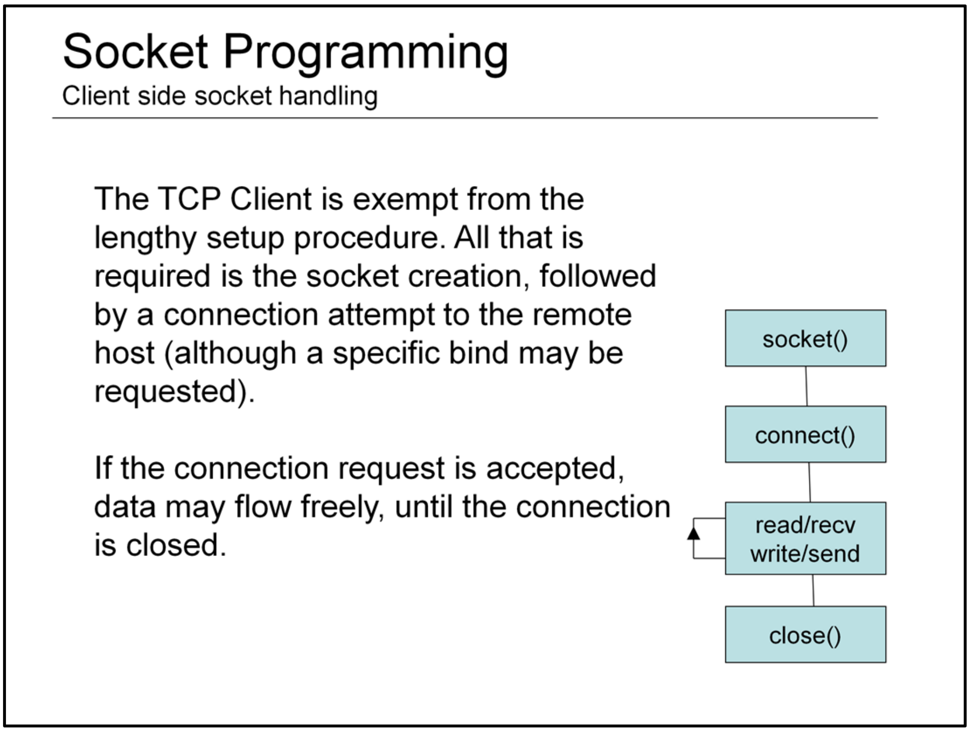
accept()

read/recv
write/send

close()

The TCP Server must complete several preparatory steps, prior to establishing connections. These steps involve setting up the socket, binding the local address, and preparing to accept connections.

Each incoming connection request may then be accepted. Data may be readily read or written to the socket. Finally, when no more data is available, the socket may be closed.



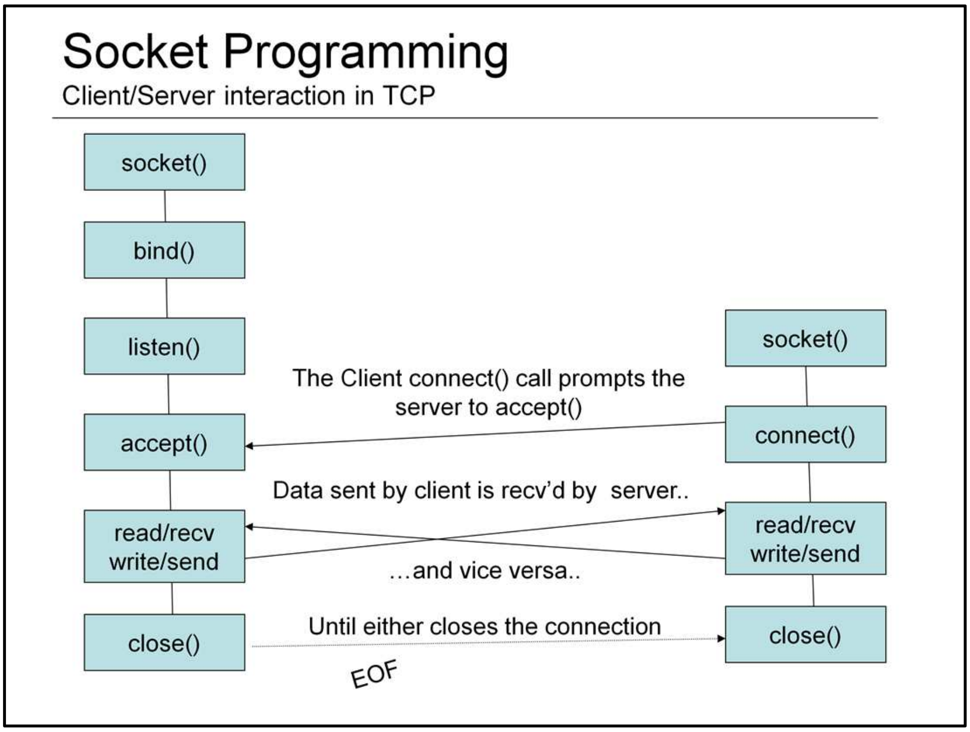
Socket Programming

Client side socket handling

The TCP Client is exempt from the lengthy setup procedure. All that is required is the socket creation, followed by a connection attempt to the remote host (although a specific bind may be requested).

If the connection request is accepted, data may flow freely, until the connection is closed.

socket()
connect()
read/recv
write/send
close()



Socket Programming

Client/Server interaction in TCP

socket()

bind()

listen()

socket()

The Client connect() call prompts the server to accept()

accept()

connect()

Data sent by client is recv'd by server..

read/recv
write/send

read/recv
write/send

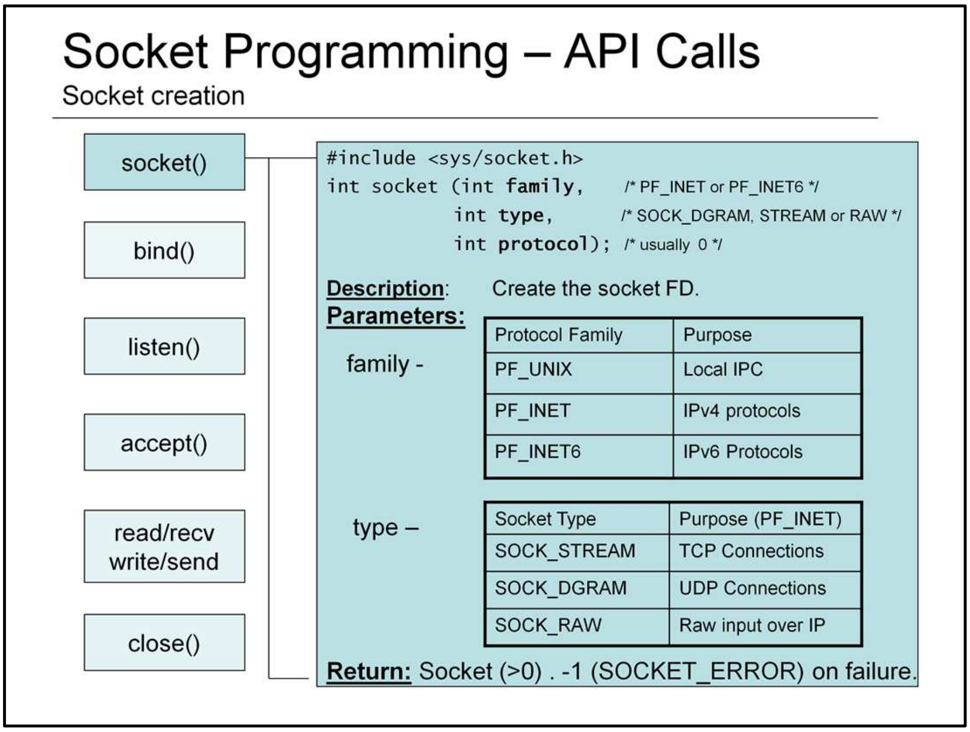
…and vice versa..

Until either closes the connection

close()

close()

EOF



Socket Programming – API Calls

Socket creation

socket()
bind()
listen()
accept()
read/recv
write/send
close()

```
#include <sys/socket.h>
int socket (int family,     /* PF_INET or PF_INET6 */
            int type,       /* SOCK_DGRAM, STREAM or RAW */
            int protocol);  /* usually  0 */
```

Description: Create the socket FD.

Parameters:

family -

Protocol Family	Purpose
PF_UNIX	Local IPC
PF_INET	IPv4 protocols
PF_INET6	IPv6 Protocols

type –

Socket Type	Purpose (PF_INET)
SOCK_STREAM	TCP Connections
SOCK_DGRAM	UDP Connections
SOCK_RAW	Raw input over IP

Return: Socket (>0) . -1 (SOCKET_ERROR) on failure.

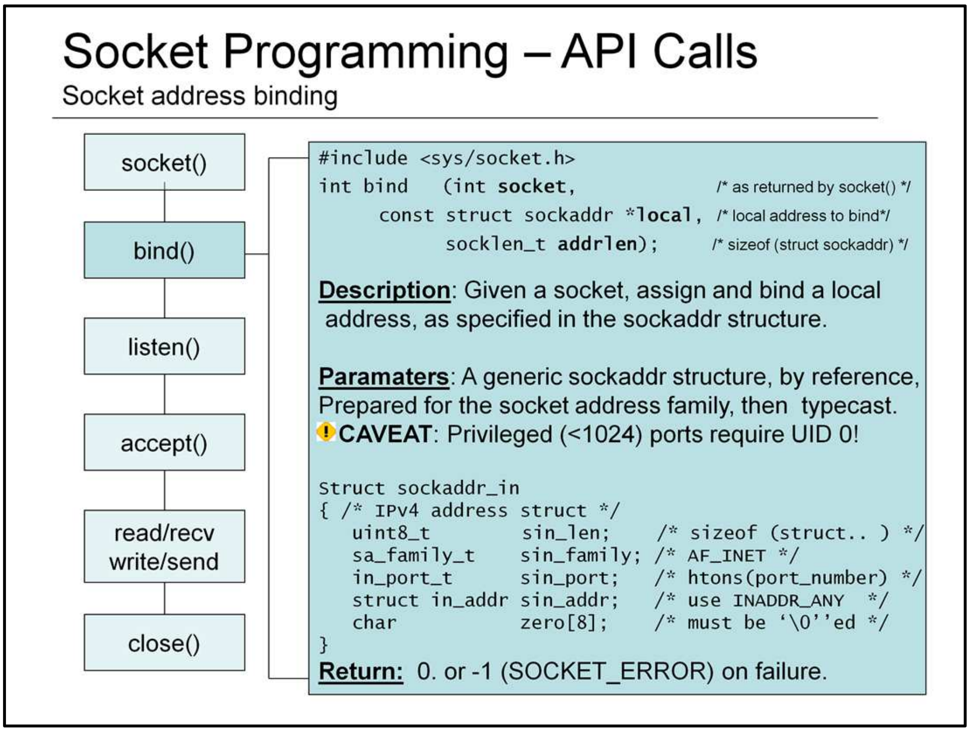
The socket() system call is used to create the local endpoint for communications. The socket may be associated with any one of the myriad address or protocol families. (Some UNIXes go with the AF_xxx constants – others (e.g. Linux) use PF_xxx). Both are defined to be equal.

Once a family is specified, a type must be selected. The types are defined as per the communication semantics required. That is:

SOCK_STREAM: Reliable, two way, connection based byte stream. For IP type sockets, this is usually TCP.
SOCK_DGRAM: Unreliable and connectionless per-packet datagram delivery (For IP: UDP)
SOCK_RAW: Unspecified Layer III and Layer IV protocols : Sender must construct IP and above headers.

SOCK_RAW is usually used in programs that need to construct ICMP packets, and/or in network sniffers.

The protocol field may usually be left at 0, but a specific protocol may be requested using getprotoent().



Socket Programming – API Calls

Socket address binding

```
#include <sys/socket.h>
int bind   (int socket,                    /* as returned by socket() */
      const struct sockaddr *local,  /* local address to bind*/
            socklen_t addrlen);        /* sizeof (struct sockaddr) */
```

Description: Given a socket, assign and bind a local address, as specified in the sockaddr structure.

Paramaters: A generic sockaddr structure, by reference, Prepared for the socket address family, then typecast.
❗**CAVEAT**: Privileged (<1024) ports require UID 0!

```
Struct sockaddr_in
{ /* IPv4 address struct */
   uint8_t         sin_len;    /* sizeof (struct.. ) */
   sa_family_t     sin_family; /* AF_INET */
   in_port_t       sin_port;   /* htons(port_number) */
   struct in_addr sin_addr;   /* use INADDR_ANY  */
   char            zero[8];    /* must be '\0''ed */
}
```

Return: 0. or -1 (SOCKET_ERROR) on failure.

Once the socket has been successfully created, the next step is to bind it to some local address. This readies the local port, and associates it with the applications. This step is NOT required for a client, but is mandatory for a server.

Initializing the sockaddr struct:

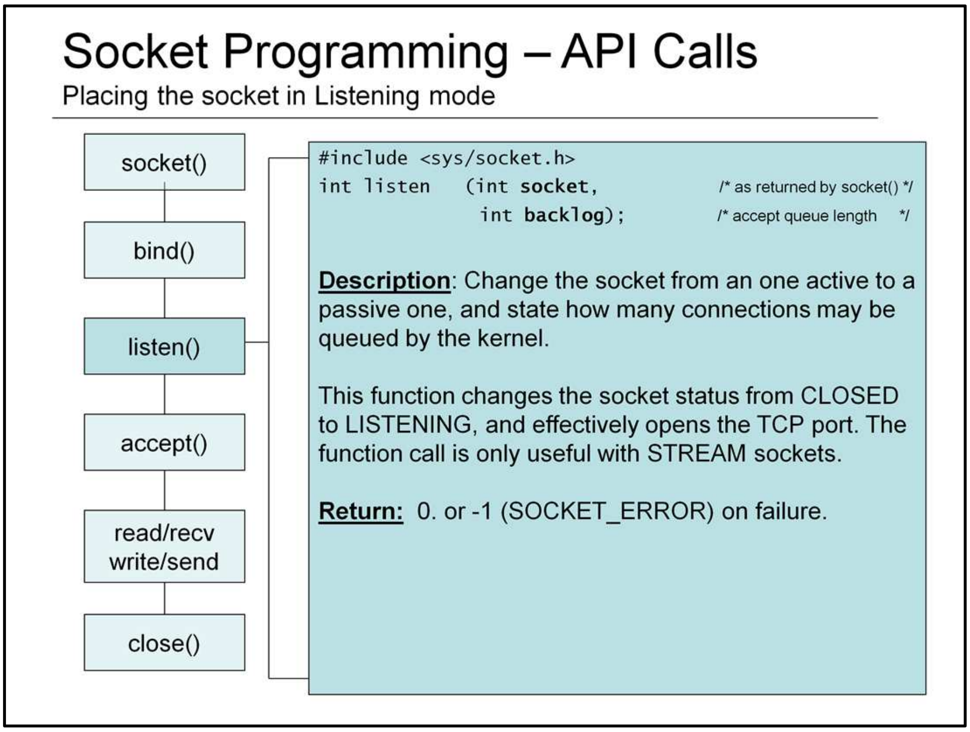
```
int     server_port =  2410 ; /* or any port you wish… */
int     socket_descriptor = socket(AF_INET, SOCK_STREAM,0)
struct sockaddr_in local;

memset(&local, '\0', sizeof(local));
local.sin_family = AF_INET;
local.sin_addr.s_addr = htonl (INADDR_ANY) /* Use any interface */
local.sin_port = htonl(server_port);

bind (socket_descriptor, (struct sockaddr *) &local, sizeof(local));
```

Unless explicitly requested otherwise, only one application may bind to a specific port at any given time. It is thus always important to check bind's return value. If it is -1, check errno.

EADDRINUSE: "Address already in use" – someone else got to bind the socket first.
EBADF/ENOTSOCKET: "Bad file descriptor"/"not a socket" – The 1st parameter was not created with a call to socket()
EINVAL: Socket is already bound.



Socket Programming – API Calls

Placing the socket in Listening mode

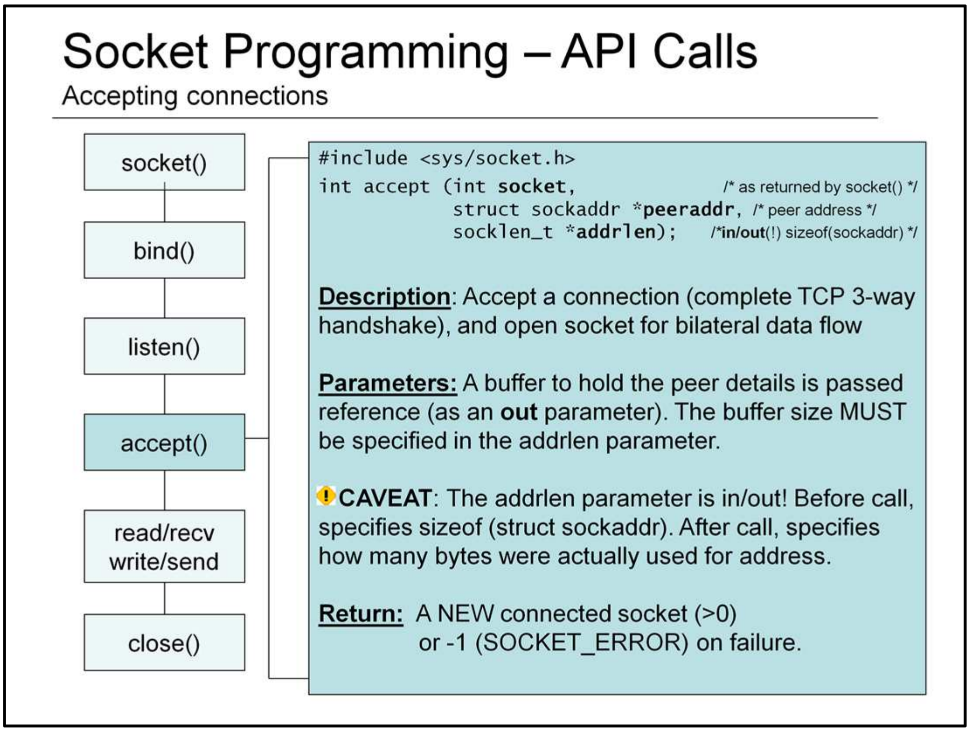
```
#include <sys/socket.h>
int listen   (int socket,          /* as returned by socket() */
               int backlog);       /* accept queue length    */
```

Description: Change the socket from an one active to a passive one, and state how many connections may be queued by the kernel.

This function changes the socket status from CLOSED to LISTENING, and effectively opens the TCP port. The function call is only useful with STREAM sockets.

Return: 0. or -1 (SOCKET_ERROR) on failure.

It is important to set the backlog parameter to a correct value. If the backlog limit of connection requests is met, and none is processed (by accept()), any further connection request will be refused (TCP RST) by the server.



Socket Programming – API Calls

Accepting connections

socket()
bind()
listen()
accept()
read/recv
write/send
close()

```
#include <sys/socket.h>
int accept (int socket,                  /* as returned by socket() */
            struct sockaddr *peeraddr,   /* peer address */
            socklen_t *addrlen);         /*in/out(!) sizeof(sockaddr) */
```

Description: Accept a connection (complete TCP 3-way handshake), and open socket for bilateral data flow

Parameters: A buffer to hold the peer details is passed reference (as an **out** parameter). The buffer size MUST be specified in the addrlen parameter.

! **CAVEAT**: The addrlen parameter is in/out! Before call, specifies sizeof (struct sockaddr). After call, specifies how many bytes were actually used for address.

Return: A NEW connected socket (>0)
or -1 (SOCKET_ERROR) on failure.

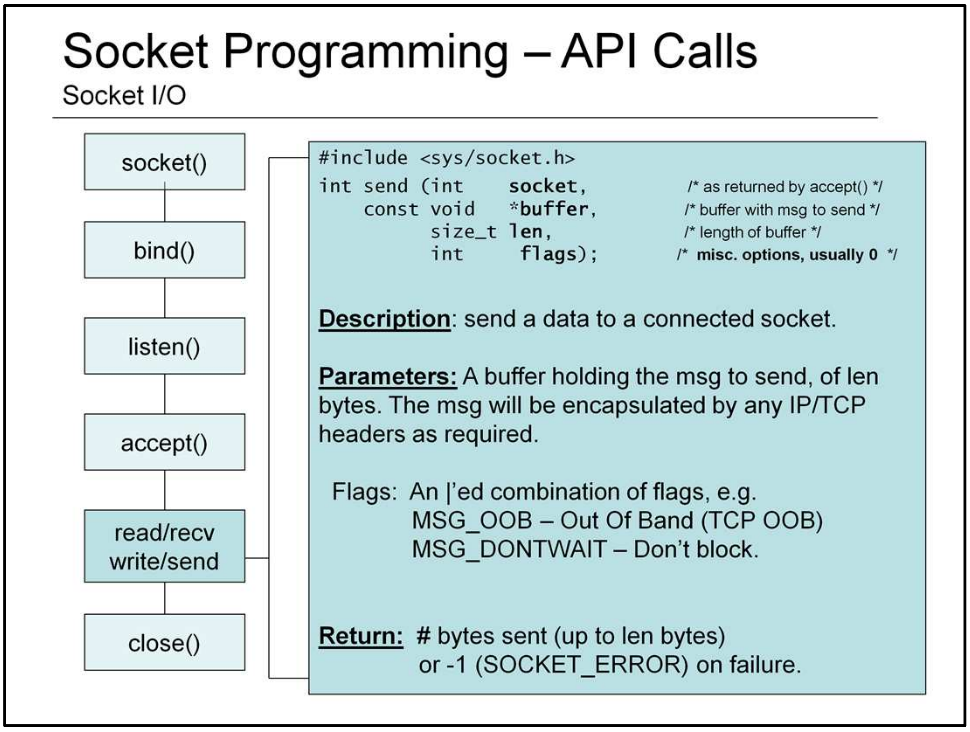
The accept() call is responsible for dequeuing a connection from the listen backlog. It usually called when the
socket signals I/O pending (see select), but may be called any time, putting the process to sleep until a request is received.

When accept returns, it returns a NEW socket, bound and connected to the remote address. The original socket is left unchanged, in its listening state.

To obtain the remote address, the following code may be used:

```
char *remote_IP_Address = (char *) malloc (BUFSIZE);
inet_ntop(AF_INET, &peeraddr.sin_addr, remote_IP_Address, BUFSIZE);
int  remote_Port = ntohs(peeraddr.sin_port);

printf (“Incoming: %s:%d requested connection”, remote_IP_Address, remote_Port);
```



Socket Programming – API Calls

Socket I/O

socket()

bind()

listen()

accept()

read/recv
write/send

close()

```
#include <sys/socket.h>
int send (int    socket,          /* as returned by accept() */
    const void   *buffer,         /* buffer with msg to send */
          size_t len,             /* length of buffer */
          int     flags);        /*  misc. options, usually 0  */
```

Description: send a data to a connected socket.

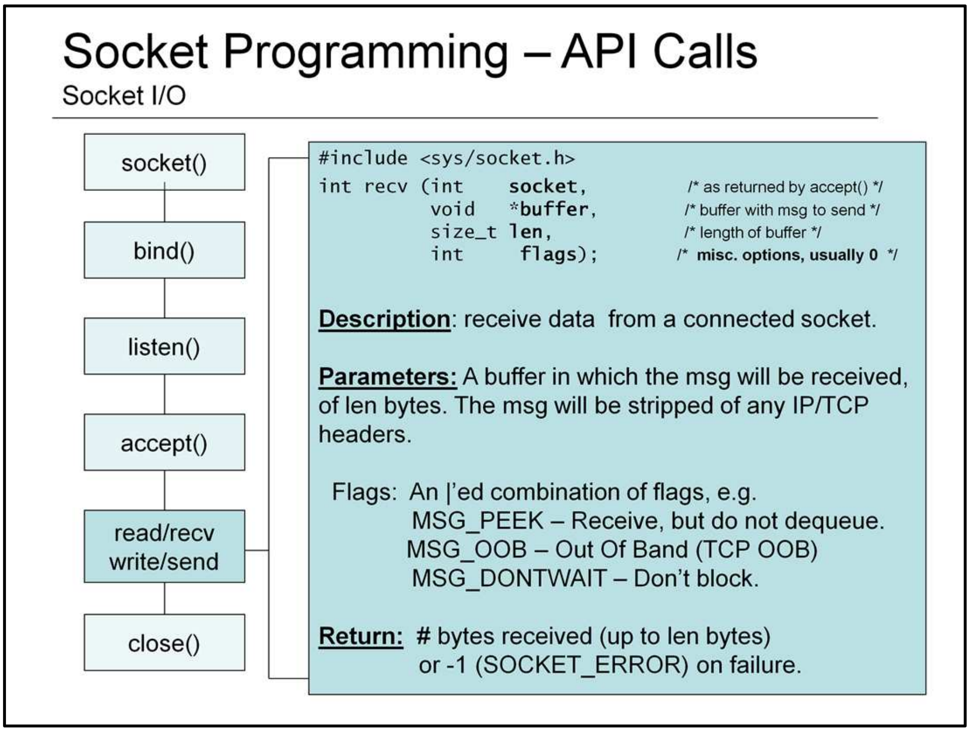
Parameters: A buffer holding the msg to send, of len bytes. The msg will be encapsulated by any IP/TCP headers as required.

Flags: An |'ed combination of flags, e.g.
MSG_OOB – Out Of Band (TCP OOB)
MSG_DONTWAIT – Don't block.

Return: # bytes sent (up to len bytes)
or -1 (SOCKET_ERROR) on failure.

The send function is used to send bytes. The function is responsible for encapsulating the data sent in any IP and TCP headers, as well as fragmenting the data, if necessary.

The send operation is a **blocking call** – that is, the function will not return until the bytes are sent. Should a
non-blocking mode of operation be required, use MSG_DONTWAIT as a flag. If the call would block, the function will return SOCKET_ERROR, and errno will be EAGAIN/EWOULDBLOCK.



Socket Programming – API Calls

Socket I/O

socket()

bind()

listen()

accept()

read/recv
write/send

close()

```
#include <sys/socket.h>
int recv (int     socket,          /* as returned by accept() */
          void    *buffer,         /* buffer with msg to send */
          size_t  len,             /* length of buffer */
          int      flags);        /*  misc. options, usually 0  */
```

Description: receive data from a connected socket.

Parameters: A buffer in which the msg will be received, of len bytes. The msg will be stripped of any IP/TCP headers.

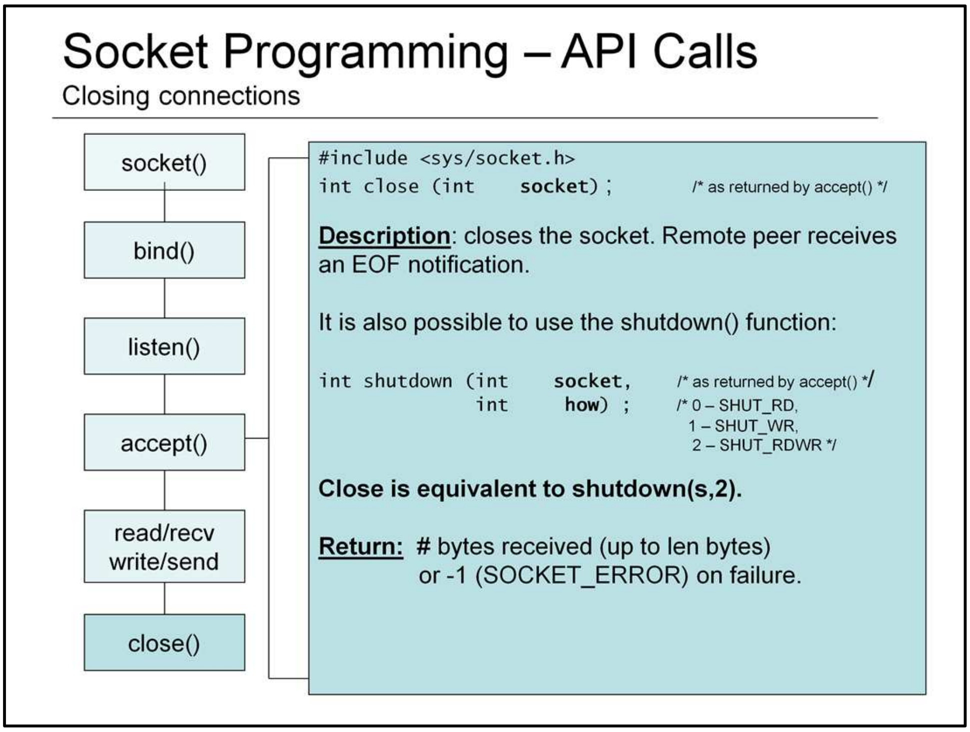
Flags: An |'ed combination of flags, e.g.
MSG_PEEK – Receive, but do not dequeue.
MSG_OOB – Out Of Band (TCP OOB)
MSG_DONTWAIT – Don't block.

Return: # bytes received (up to len bytes)
or -1 (SOCKET_ERROR) on failure.

The receive function is used to receive bytes. The function returns the data, sans any IP and TCP headers from the data obtained, as well returning the data reassembled.

The receive operation is a **blocking call** – that is, the function will not return until there are bytes to receive. Should a non-blocking mode of operation be required, use MSG_DONTWAIT as a flag. If the call would block, the function will return SOCKET_ERROR, and errno will be EAGAIN/EWOULDBLOCK.

To determine if bytes are available, use the poll() or select() system calls.



Socket Programming – API Calls

Closing connections

socket()

bind()

listen()

accept()

read/recv
write/send

close()

```
#include <sys/socket.h>
int close (int    socket);          /* as returned by accept() */
```

Description: closes the socket. Remote peer receives an EOF notification.

It is also possible to use the shutdown() function:

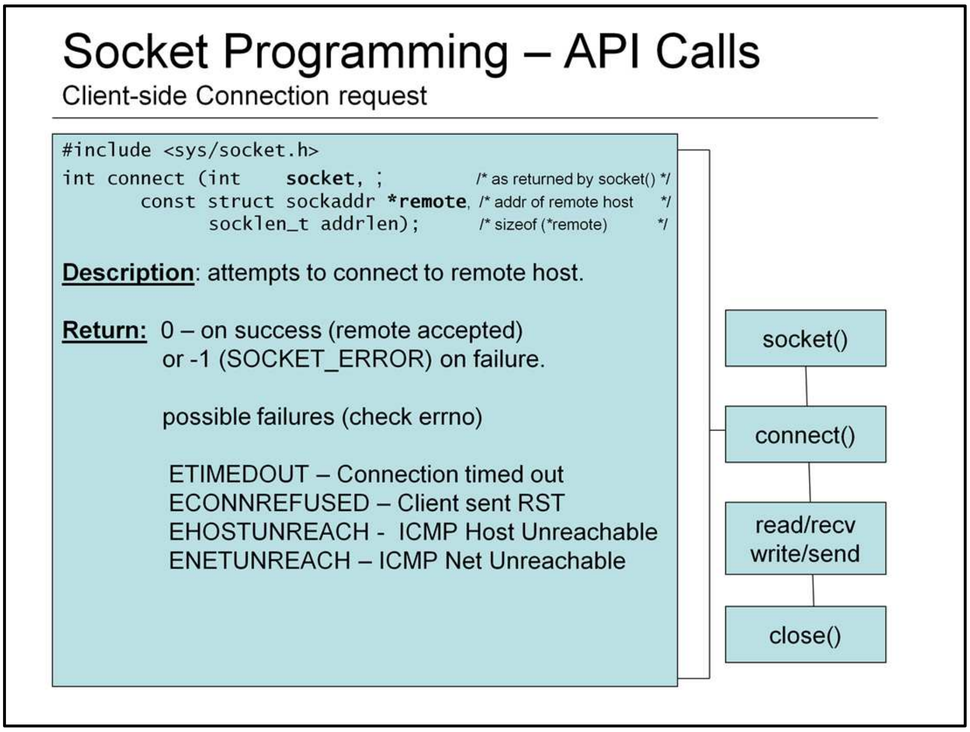
```
int shutdown (int    socket,    /* as returned by accept() */
              int     how) ;    /* 0 – SHUT_RD,
                                   1 – SHUT_WR,
                                   2 – SHUT_RDWR */
```

Close is equivalent to shutdown(s,2).

Return: # bytes received (up to len bytes)
or -1 (SOCKET_ERROR) on failure.

The close() function terminates the connection by closing the socket. The remote peer will receive an EOF notification upon further reads/writes to the socket.

Should a half-duplex close be required, use shutdown(), instead.



Socket Programming – API Calls

Client-side Connection request

```
#include <sys/socket.h>
int connect (int    socket, ;               /* as returned by socket() */
       const struct sockaddr *remote,  /* addr of remote host     */
             socklen_t addrlen);        /* sizeof (*remote)        */
```

Description: attempts to connect to remote host.

Return: 0 – on success (remote accepted)
or -1 (SOCKET_ERROR) on failure.

possible failures (check errno)

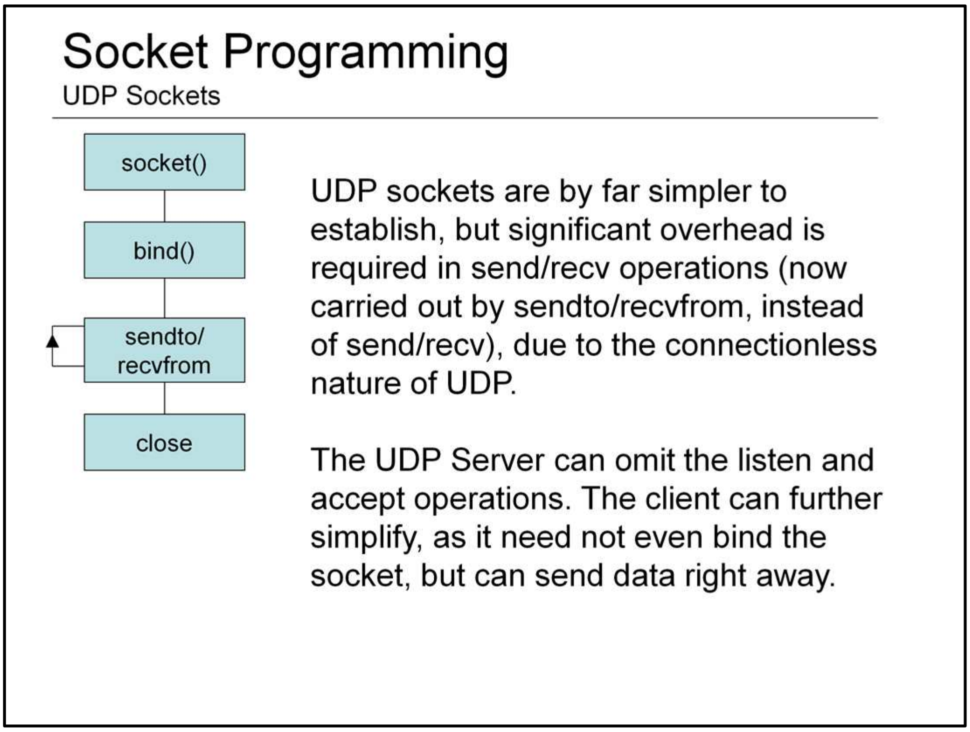
ETIMEDOUT – Connection timed out
ECONNREFUSED – Client sent RST
EHOSTUNREACH - ICMP Host Unreachable
ENETUNREACH – ICMP Net Unreachable

socket()
connect()
read/recv write/send
close()

The connect() function attempts to establish a connection with the remote host. For the AF_INET family, this means sending a TCP SYN to the remote host, and waiting for the SYN/ACK to return.

If the SYN/ACK does not return within a specified timeout (usually 75 seconds), the function fails, and errno is set to ETIMEDOUT.

If any ICMP messages are returned by the network as a result of the connection attempt, errno is set to EHOSTUNREACH, or ENETREACH, for ICMP Host or Net Unreachable, respectively.



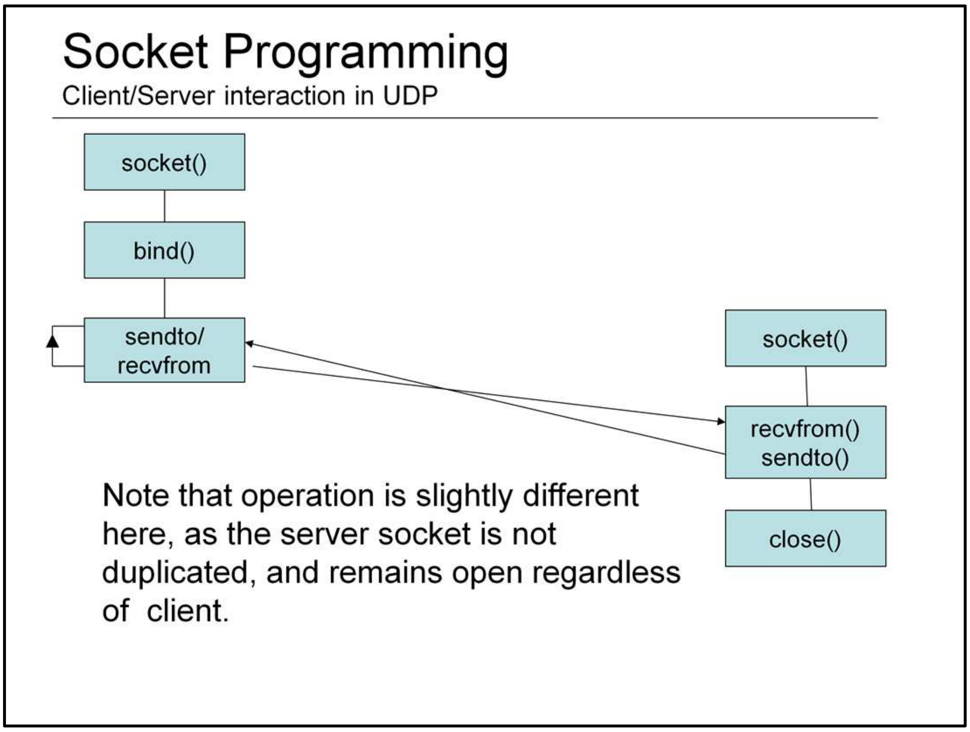
Socket Programming

UDP Sockets

socket()
bind()
sendto/
recvfrom
close

UDP sockets are by far simpler to establish, but significant overhead is required in send/recv operations (now carried out by sendto/recvfrom, instead of send/recv), due to the connectionless nature of UDP.

The UDP Server can omit the listen and accept operations. The client can further simplify, as it need not even bind the socket, but can send data right away.



Socket Programming

Client/Server interaction in UDP

socket()

bind()

sendto/
recvfrom

socket()

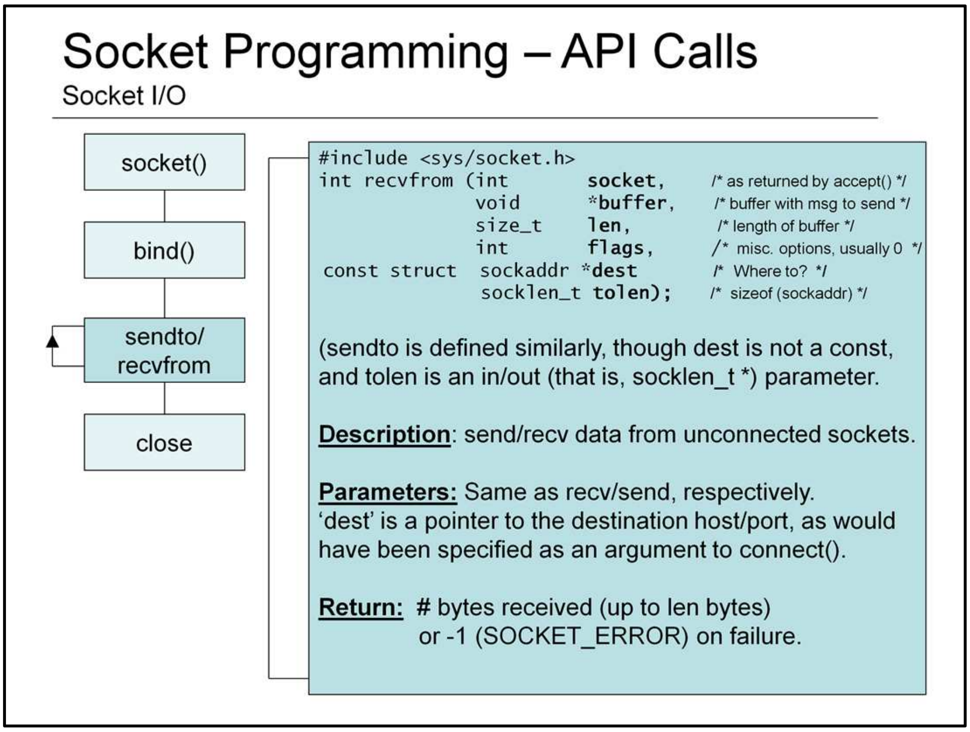
recvfrom()
sendto()

close()

Note that operation is slightly different here, as the server socket is not duplicated, and remains open regardless of client.

Since the server did not listen and accept, the socket was NOT duplicated. Therefore one socket serves all clients concurrently. It remains open even after client disconnection.

The server has to manage the client list itself, with each datagram containing the address of the peer, obtained from the sockaddr *) struct.



Socket Programming – API Calls

Socket I/O

socket()

bind()

sendto/ recvfrom

close

```
#include <sys/socket.h>
int recvfrom (int        socket,     /* as returned by accept() */
              void       *buffer,    /* buffer with msg to send */
              size_t     len,        /* length of buffer */
              int        flags,      /* misc. options, usually 0 */
 const struct  sockaddr *dest        /* Where to? */
               socklen_t tolen);     /* sizeof (sockaddr) */
```

(sendto is defined similarly, though dest is not a const, and tolen is an in/out (that is, socklen_t *) parameter.

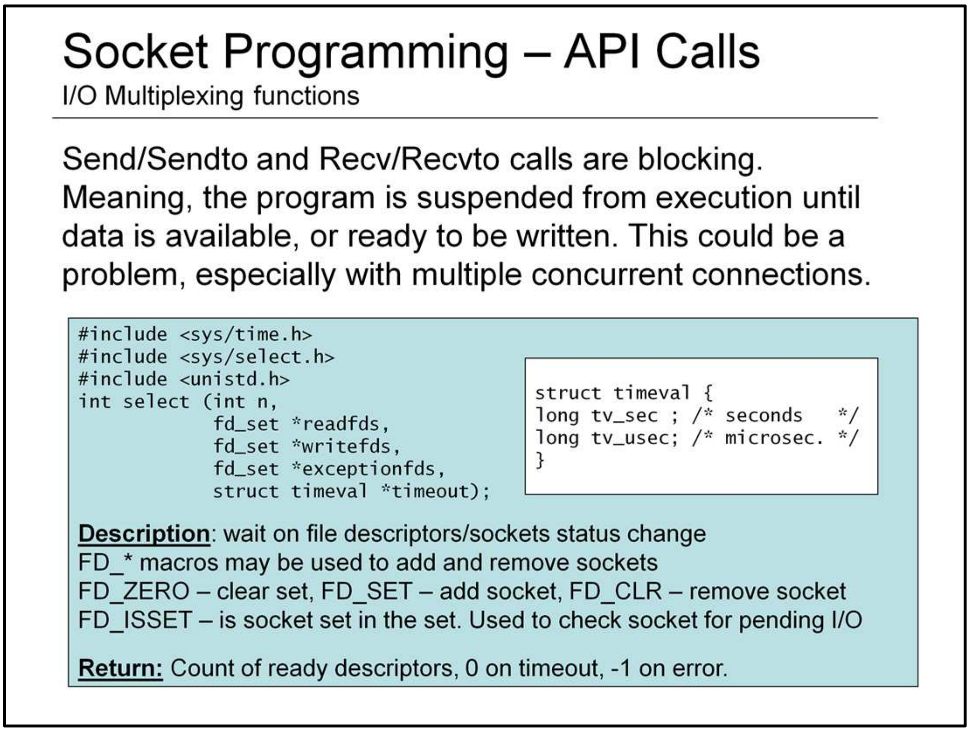
Description: send/recv data from unconnected sockets.

Parameters: Same as recv/send, respectively.
'dest' is a pointer to the destination host/port, as would have been specified as an argument to connect().

Return: # bytes received (up to len bytes)
or -1 (SOCKET_ERROR) on failure.

Note that some flags (e.g. MSG_OOB) are not applicable for unconnected sockets.

Otherwise, the behavior of sendto() is nearly identical to send, and recv – to recvfrom().



Socket Programming – API Calls

I/O Multiplexing functions

Send/Sendto and Recv/Recvto calls are blocking. Meaning, the program is suspended from execution until data is available, or ready to be written. This could be a problem, especially with multiple concurrent connections.

```
#include <sys/time.h>
#include <sys/select.h>
#include <unistd.h>
int select (int n,
            fd_set *readfds,
            fd_set *writefds,
            fd_set *exceptionfds,
            struct timeval *timeout);
```

```
struct timeval {
long tv_sec ; /* seconds    */
long tv_usec; /* microsec. */
}
```

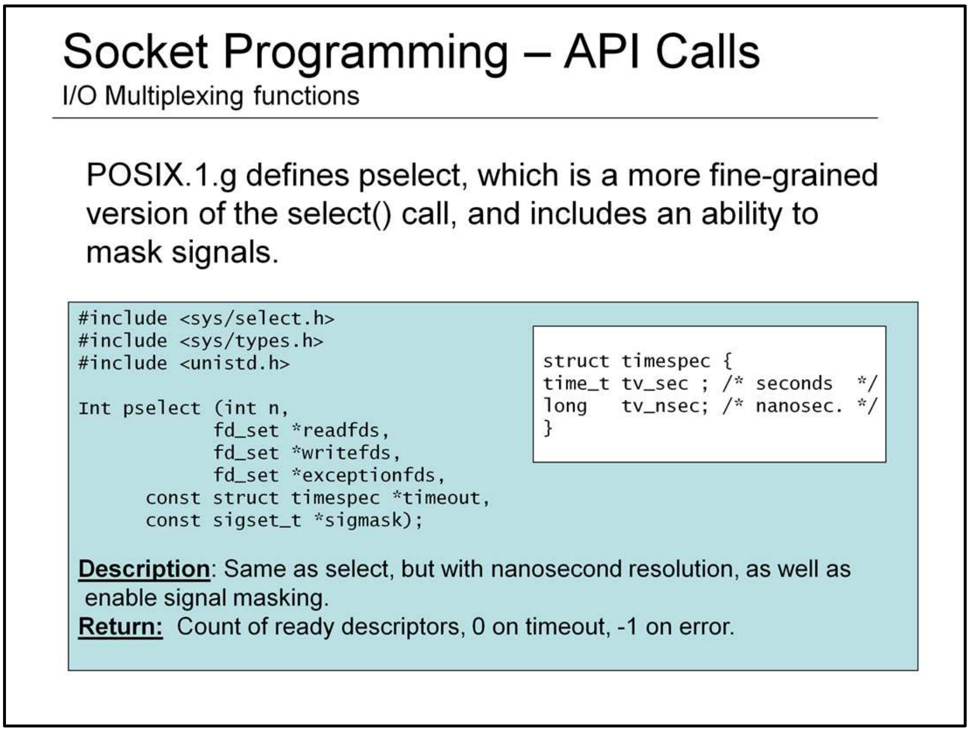
Description: wait on file descriptors/sockets status change
FD_* macros may be used to add and remove sockets
FD_ZERO – clear set, FD_SET – add socket, FD_CLR – remove socket
FD_ISSET – is socket set in the set. Used to check socket for pending I/O

Return: Count of ready descriptors, 0 on timeout, -1 on error.

Most applications perform I/O Multiplexing. That is, reading and writing to multiple descriptors, as well as doing other things, in between read/write operations.
Select/Pselect and Poll are used to check socket readiness for read/write operations.
On connected sockets, these calls detect available data, or TCP buffer space availability. On listening sockets, these calls detect incoming connections (and thus, whether accept() may be called).

On unconnected (UDP) sockets, these calls are particularly important, as data may come at arbitrary intervals (or not come at all for a while..).

```
void FD_ZERO (fd_set *fdset);         /* clear all bits in fdset */
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 */
void FD_ISSET(int fd, fd_set *fdset); /* is the bit for fd on in the fdset? */
```



Socket Programming – API Calls

I/O Multiplexing functions

POSIX.1.g defines pselect, which is a more fine-grained version of the select() call, and includes an ability to mask signals.

```
#include <sys/select.h>
#include <sys/types.h>
#include <unistd.h>

Int pselect (int n,
            fd_set *readfds,
            fd_set *writefds,
            fd_set *exceptionfds,
      const struct timespec *timeout,
      const sigset_t *sigmask);
```

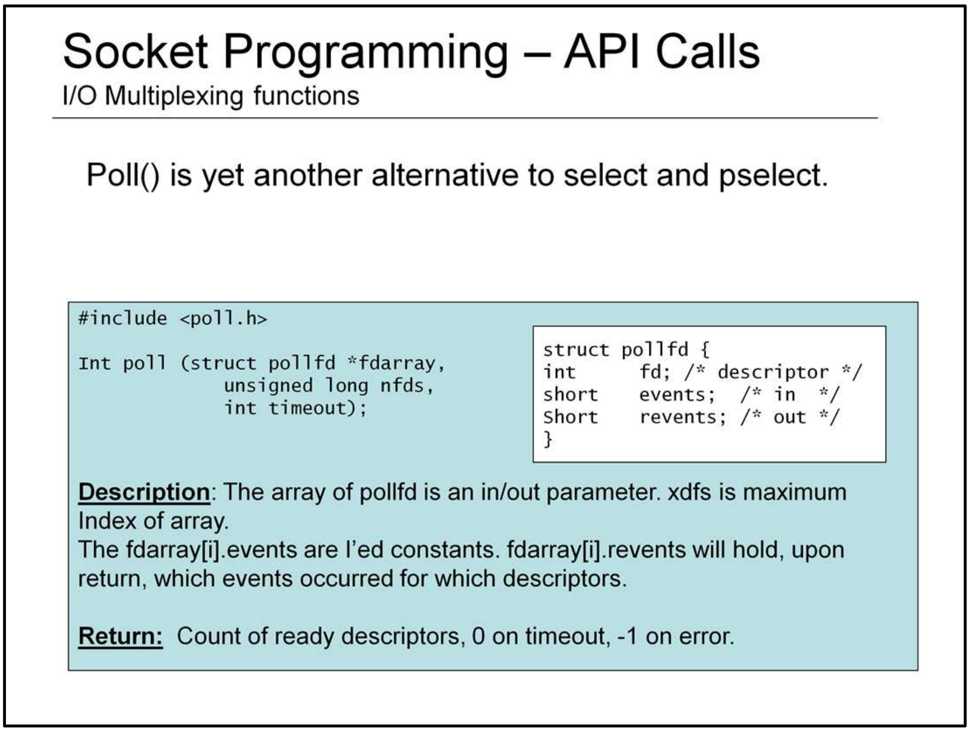
```
struct timespec {
time_t tv_sec ; /* seconds  */
long   tv_nsec; /* nanosec. */
}
```

Description: Same as select, but with nanosecond resolution, as well as enable signal masking.
Return: Count of ready descriptors, 0 on timeout, -1 on error.

Example of using pselect's signal handling:

```
sigset_t   newmask, oldmask, zeromask;

sigemptyset(&zeromask) ; /* Set all signals to default */
sigemptyset(&newmask);
sigaddset(&newmask, SIGINT); /* tell pselect to return on SIGINT */
sigprocmask(SIG_BLOCK, &newmask, &oldmask); /* block SIGINT */
ready_descriptors = pselect (……., &zeromask)) < 0) ;
If (ready_descriptors < 0 )
{
  if (errno == EINTR)
   {
    /*We were interrupted */
  }
}
```



Socket Programming – API Calls

I/O Multiplexing functions

Poll() is yet another alternative to select and pselect.

```
#include <poll.h>

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

```
struct pollfd {
int     fd; /* descriptor */
short   events;  /* in  */
Short   revents; /* out */
}
```

Description: The array of pollfd is an in/out parameter. xdfs is maximum Index of array.
The fdarray[i].events are I'ed constants. fdarray[i].revents will hold, upon return, which events occurred for which descriptors.

Return: Count of ready descriptors, 0 on timeout, -1 on error.

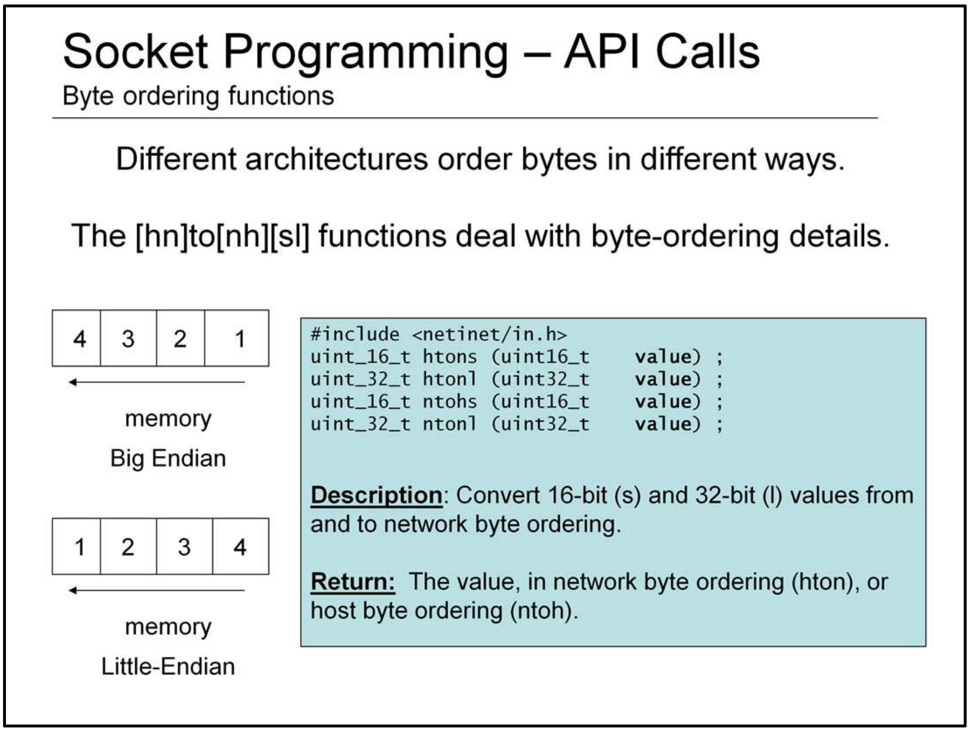
Events:

POLLIN – Incoming Data (Normal or OOB)
POLLRDNORM – Incoming Normal Data
POLLRDBAND – Incoming OOB data
POLLPRI – High priority incoming data

POLLOUT | POLLWRNORM – Outgoing Normal Data may be written
POLLWRBAND – Priority data may be written

Also, for revents:

POLLERR – An error has occurred
POLLHUP – Hangup occurred
POLLNVAL – Descriptor is not an open file/socket



Socket Programming – API Calls

Byte ordering functions

Different architectures order bytes in different ways.

The [hn]to[nh][sl] functions deal with byte-ordering details.

```
#include <netinet/in.h>
uint_16_t htons (uint16_t    value) ;
uint_32_t htonl (uint32_t    value) ;
uint_16_t ntohs (uint16_t    value) ;
uint_32_t ntonl (uint32_t    value) ;
```

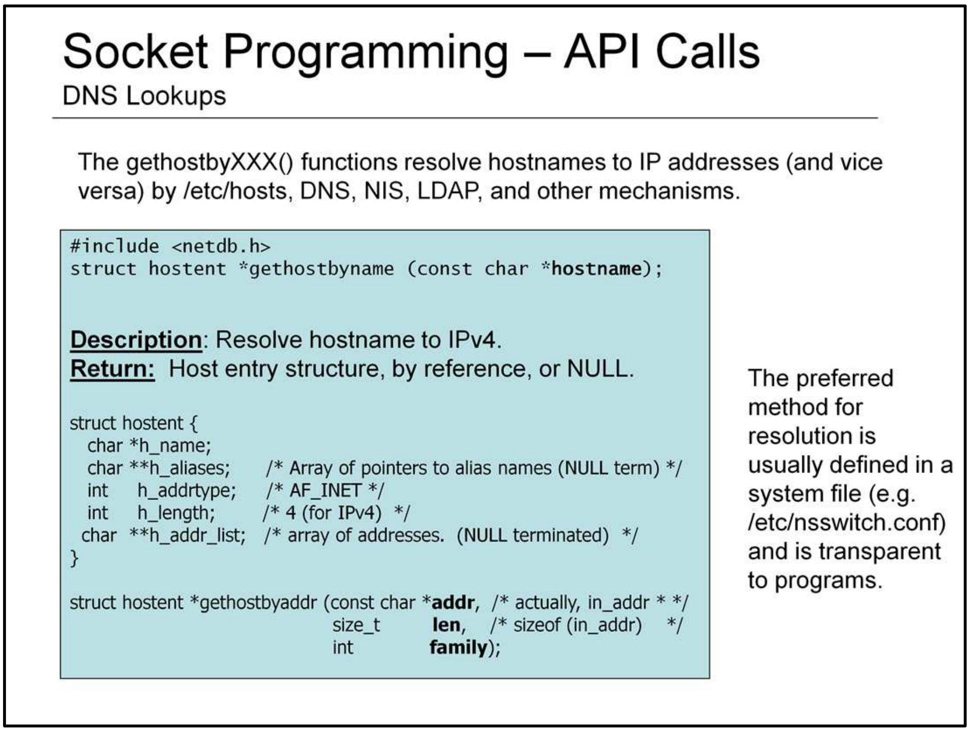
Description: Convert 16-bit (s) and 32-bit (l) values from and to network byte ordering.

Return: The value, in network byte ordering (hton), or host byte ordering (ntoh).

When sending data across a network, you may not always encounter similar architectures with your peers. (e.g. Intel x86 machines vs. Sparc servers).

In order for programs to be fully source code portable, it is recommended to ALWAYS use the XtoYZ (X,Y – h,n, Z – s,l) functions. In cases where the host and network byte ordering functions are identical, the functions are defined as null macros.

It is especially important to use these functions when constructing sockaddr_in structures (htons() for sin_port, and htonl() for sin_addr).



Socket Programming – API Calls

DNS Lookups

The gethostbyXXX() functions resolve hostnames to IP addresses (and vice versa) by /etc/hosts, DNS, NIS, LDAP, and other mechanisms.

```
#include <netdb.h>
struct hostent *gethostbyname (const char *hostname);
```

Description: Resolve hostname to IPv4.
Return: Host entry structure, by reference, or NULL.

```
struct hostent {
   char *h_name;
   char **h_aliases;      /* Array of pointers to alias names (NULL term) */
   int     h_addrtype;     /* AF_INET */
   int     h_length;       /* 4 (for IPv4)  */
  char  **h_addr_list;  /* array of addresses.  (NULL terminated)  */
}

struct hostent *gethostbyaddr (const char *addr,  /* actually, in_addr * */
                               size_t        len,    /* sizeof (in_addr)    */
                               int           family);
```

The preferred method for resolution is usually defined in a system file (e.g. /etc/nsswitch.conf) and is transparent to programs.

h_addr is #defined as h_addr_list[0];

The nsswitch.conf file is a table, specifying the order of resolvers. It is used for other namespaces, as can be seen below:

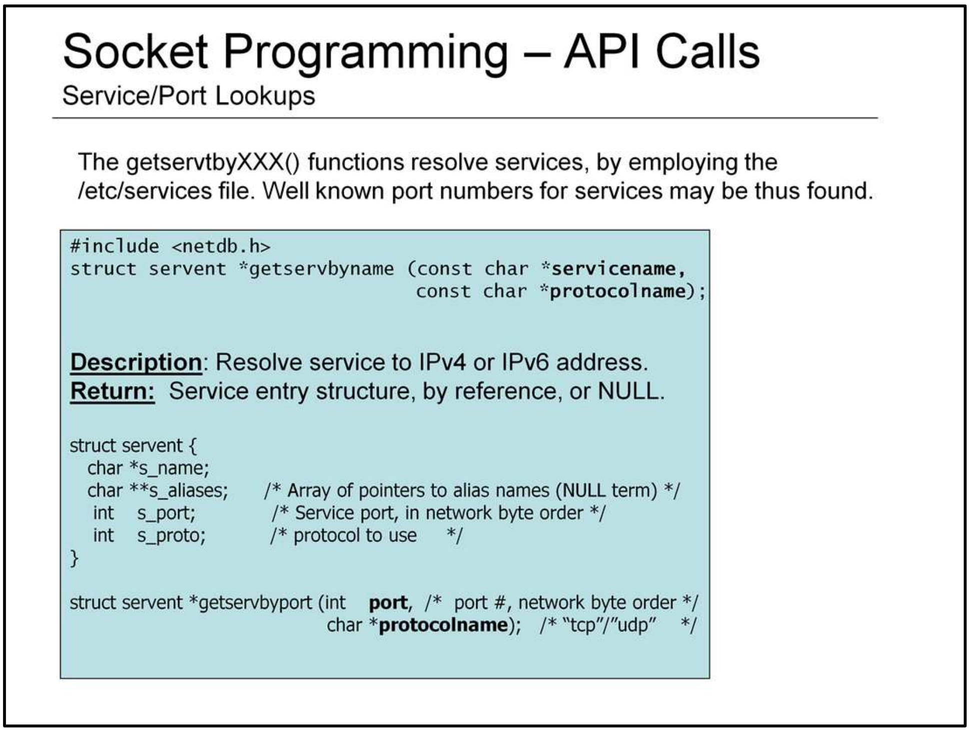
```
# Sample /etc/nsswitch.conf file:
# namespaces: passwd, shadow, group, hosts, services, networks, rpc…
# resolvers: nisplus, nis, dns, files, db (local database), hesiod (rare)..

hosts: dns files nisplus
passwd: nisplus [NOTFOUND=return] files
```

While most systems use /etc/nsswitch.conf, AIX uses /etc/netsvc.conf. Digital UNIX uses /etc/svc.conf.

DNS lookup is performed against the DNS specified in /etc/resolv.conf.

NOTE THIS FUNCTION IS NOT IPv6-Compatible!



Socket Programming – API Calls

Service/Port Lookups

The getservtbyXXX() functions resolve services, by employing the /etc/services file. Well known port numbers for services may be thus found.

```
#include <netdb.h>
struct servent *getservbyname (const char *servicename,
                               const char *protocolname);
```

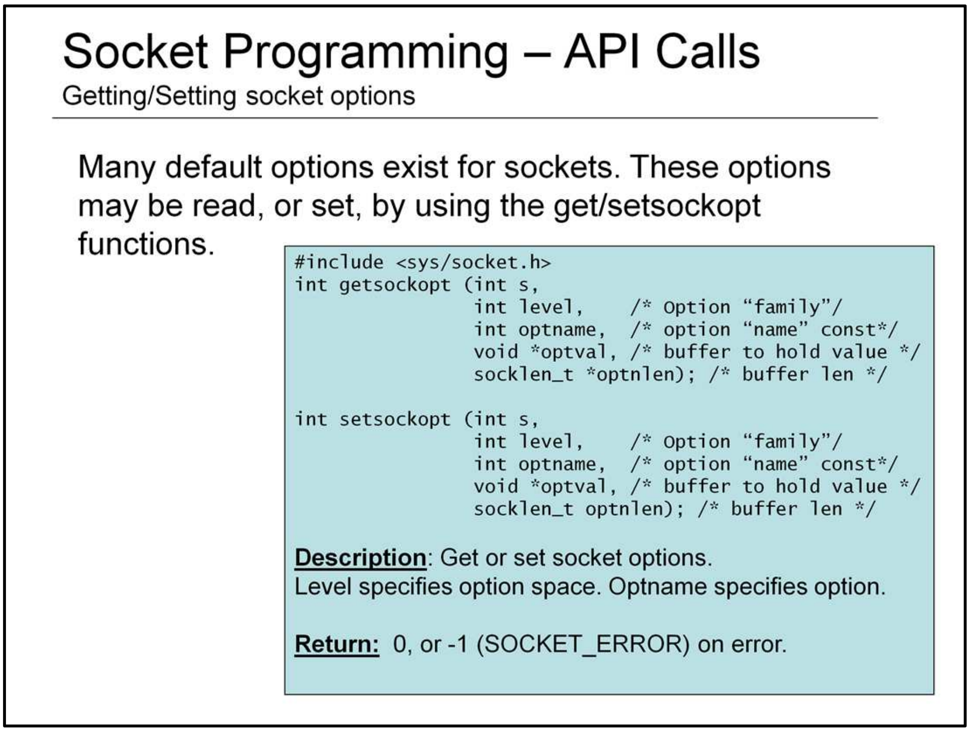
Description: Resolve service to IPv4 or IPv6 address.
Return: Service entry structure, by reference, or NULL.

```
struct servent {
  char *s_name;
  char **s_aliases;     /* Array of pointers to alias names (NULL term) */
   int    s_port;         /* Service port, in network byte order */
   int    s_proto;        /* protocol to use     */
}

struct servent *getservbyport (int    port,  /*  port #, network byte order */
                               char *protocolname);   /* "tcp"/"udp"    */
```

Example Usage:
getservbyname (“domain”, “udp”);
getservbyport (htons(53), “udp”);

The following, however, will fail:
getservbyname (“smtp”, “udp”);



Socket Programming – API Calls

Getting/Setting socket options

Many default options exist for sockets. These options may be read, or set, by using the get/setsockopt functions.

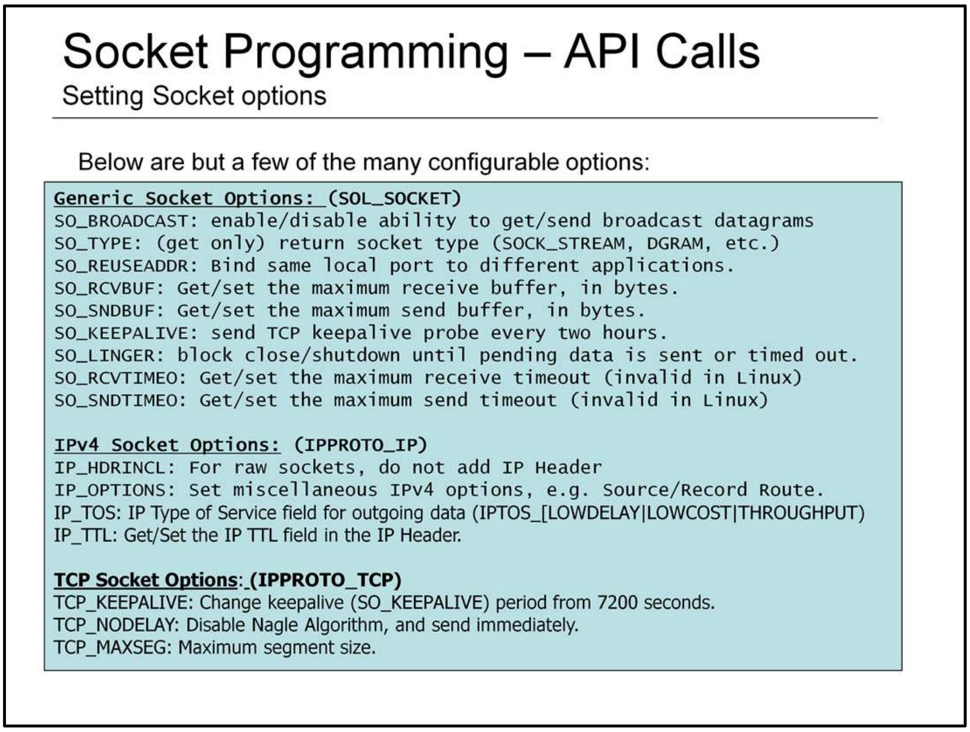
```
#include <sys/socket.h>
int getsockopt (int s,
                int level,    /* Option “family”/
                int optname,  /* option “name” const*/
                void *optval, /* buffer to hold value */
                socklen_t *optnlen); /* buffer len */

int setsockopt (int s,
                int level,    /* Option “family”/
                int optname,  /* option “name” const*/
                void *optval, /* buffer to hold value */
                socklen_t optnlen); /* buffer len */
```

Description: Get or set socket options.
Level specifies option space. Optname specifies option.

Return: 0, or -1 (SOCKET_ERROR) on error.

Level may be: SOL_SOCKET, SOL_RAW, IPPROTO_IP, IPPROTO_TCP, IPPROTO_IPV6, etc. Option names may be found by looking through the man pages. The next page shows the commonly used ones.



Socket Programming – API Calls

Setting Socket options

Below are but a few of the many configurable options:

Generic Socket Options: (SOL_SOCKET)
SO_BROADCAST: enable/disable ability to get/send broadcast datagrams
SO_TYPE: (get only) return socket type (SOCK_STREAM, DGRAM, etc.)
SO_REUSEADDR: Bind same local port to different applications.
SO_RCVBUF: Get/set the maximum receive buffer, in bytes.
SO_SNDBUF: Get/set the maximum send buffer, in bytes.
SO_KEEPALIVE: send TCP keepalive probe every two hours.
SO_LINGER: block close/shutdown until pending data is sent or timed out.
SO_RCVTIMEO: Get/set the maximum receive timeout (invalid in Linux)
SO_SNDTIMEO: Get/set the maximum send timeout (invalid in Linux)

IPv4 Socket Options: (IPPROTO_IP)
IP_HDRINCL: For raw sockets, do not add IP Header
IP_OPTIONS: Set miscellaneous IPv4 options, e.g. Source/Record Route.
IP_TOS: IP Type of Service field for outgoing data (IPTOS_[LOWDELAY|LOWCOST|THROUGHPUT)
IP_TTL: Get/Set the IP TTL field in the IP Header.

TCP Socket Options: (IPPROTO_TCP)
TCP_KEEPALIVE: Change keepalive (SO_KEEPALIVE) period from 7200 seconds.
TCP_NODELAY: Disable Nagle Algorithm, and send immediately.
TCP_MAXSEG: Maximum segment size.

SO_REUSEADDR: Rebinding a socket even if a connection is already established on it, rebinding the same port on different interfaces, or even allowing duplicate bindings on UDP & Multicast sockets.

Linux adds quite a few IP and generic socket options, which are specified in ip(7) and socket(7), respectively.

Linux also adds a useful ICMP_FILTER option (level: SOL_RAW), to filter ICMP packets from other raw data.