Know your TCP system call sequences

The sequence of function calls from the kernel to the application level

Bindu Anupama

November 06, 2007

The TCP/IP programming interface provides various system calls to help you effectively use the protocol. The TCP stack code is vast, and a complete call sequence down to the kernel level would help in understanding the TCP stack. In this article, review and study detailed information about the TCP call sequence, including references to FreeBSD and important function calls that occur in the TCP stack after a system call at the user level.

Introduction

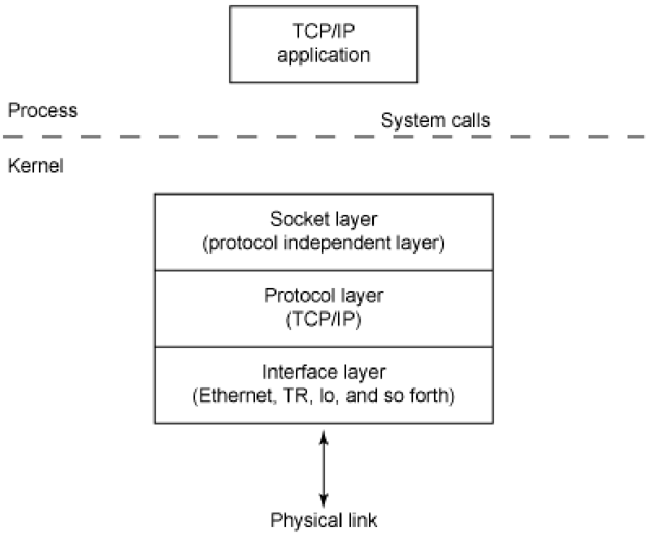
A typical TCP client and server application issues a sequence of TCP system calls to attain certain functions. Some of these system calls include `socket ()`, `bind ()`, `listen ()`, `accept ()`, `send ()`, and `receive()`. This article explains what happens in the lower levels when an application issues the TCP system calls, as shown in Figure 1 below.

Figure 1. Normal sequence of calls made by the TCP application

TCP client application	TCP server application
Socket	Socket
Bind	Bind
-	Listen
Connect	Accept
Send	Receive
Receive	Send

Figure 2 below shows the different layers through which the TCP system call propagates before being sent out on the physical link.

Figure 2. Layers of the TCP system call



Any TCP system call that is made is received by the socket layer. The socket layer validates the correctness of the parameters passed by the TCP application. This is a *protocol-independent* layer because the protocol is not yet hooked onto the call.

Below the socket layer is the protocol layer, which contains the actual implementation of the protocol (in this case, TCP). When the socket layer makes calls into the protocol layer, it ensures that it has exclusive access for the data structures that are shared between the two layers. This is done to avoid any data structure corruption.

The various network device drivers run at the interface layer, which receives and transmits data from and to the physical link.

Each socket has a socket queue, and each interface has an interface queue used for data communication. However, there is only one protocol queue, which is called the IP input queue, for the entire protocol layer. The interface layer inputs data to the protocol layer through this IP input queue. The protocol layer outputs data to the interface using the respective interface queues.

In this article, learn about the following system calls:

- `Socket`
- `Bind`
- `Listen`
- `Accept`
- `Connect`
- `Shutdown`
- `Close`
- `Send`
- `Receive`

Socket

```
socket (struct proc *p, struct socket_args *uap, int retval)
struct sock_args
{
int domain,
int type,
int protocol;
};
```

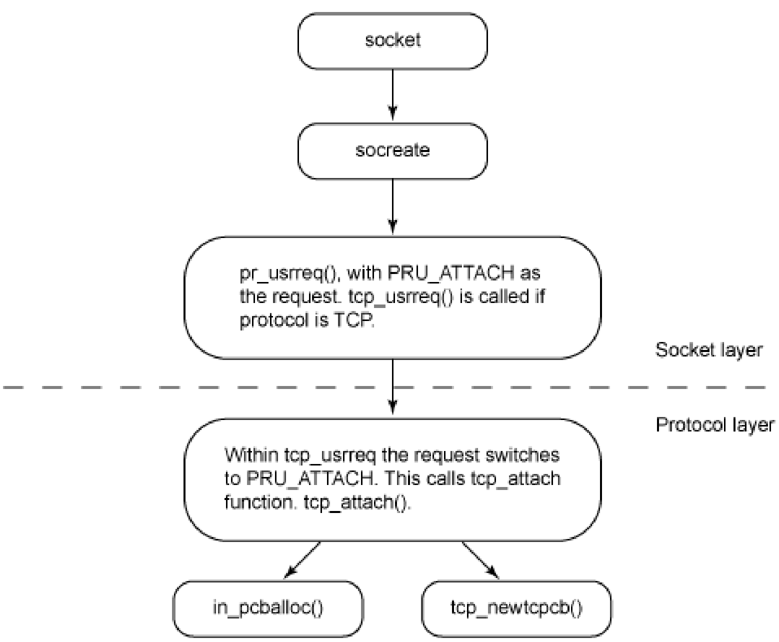
In the `socket` system call:

- `p` is a pointer to the proc structure of the process that makes the `socket` call.
- `uap` is a pointer to the `socket_args` structure that contains the arguments passed to the process in the `socket` system call.
- `retval` is the return value of the system call.

The `socket` system call creates a new socket by assigning a new descriptor. The new descriptor is returned to the calling process. Any subsequent system calls are identified with the created socket. The `socket` system call also assigns the protocol to the created socket descriptor.

The `domain`, `type`, and `protocol` argument values specify the family, type, and protocol to assign to the created socket. Figure 3 shows the call sequence.

Figure 3. Call sequence for socket system call



Once the arguments are retrieved from the process, the `socket` function calls the `socreate` function. The `socreate` function finds the pointer to the protocol switch `protsw` structure, depending on the arguments specified by the process. The `socreate` function then allocates a new socket structure. A protocol-specific call, `pr_usrreq`, is then made, which switches to the corresponding protocol-specific request associated with the socket descriptor. The prototype of the `pr_usrreq` function is:

```
int  pr_usrreq(struct socket *so , int req, struct mbuf  *m0 , *m1 , *m2);
```

In the `pr_usrreq` function:

- `so` is a pointer to the socket structure.
- `req` is what identifies the request. In this case, it's PRU_ATTACH.
- `m0, m1, and m2` are pointers to the `mbuf` structure. The values vary depending on the request.

There are about 16 requests that are serviced by the `pr_usrreq` function.

The `tcp_usrreq()` function calls `tcp_attach( )`, which processes the PRU_ATTACH request. To allocate an Internet protocol control block, `in_pcballoc()` is called. Within `in_pcballoc`, the kernel's memory allocator function is called, which allocates memory to the Internet control block. All the necessary initializing of the Internet control block structure pointer is done and the control returns to `tcp_attach()`.

A new TCP control block is allocated and initialized in `tcp_newtcpcb()`. It also initializes all the TCP timer variables, and control returns to `tcp_attach()`. The socket state is now initialized to CLOSED. On returning to the `tcp_usrreq` function, the socket descriptor is made to point to the socket's TCP control block.

The Internet control block is a doubly linked circular linked list with a pointer pointing to the socket structure, and the `so_pcb` member of the socket structure points to the Internet control block structure. The Internet control block also has a pointer pointing to the TCP control block. For more detailed information on Internet control block and TCP control block structures, see the Related topics section.

Bind

```
bind (struct proc *p, struct bind_args *uap, int *retval)
   struct bind_args
   {   int s;
       caddr_t name;
       int namelen;
   };
```

In the `bind` system call function:

- `s` is the socket descriptor.
- `name` is the pointer to the buffer that contains the network transport address.
- `namelen` is the size of the buffer.

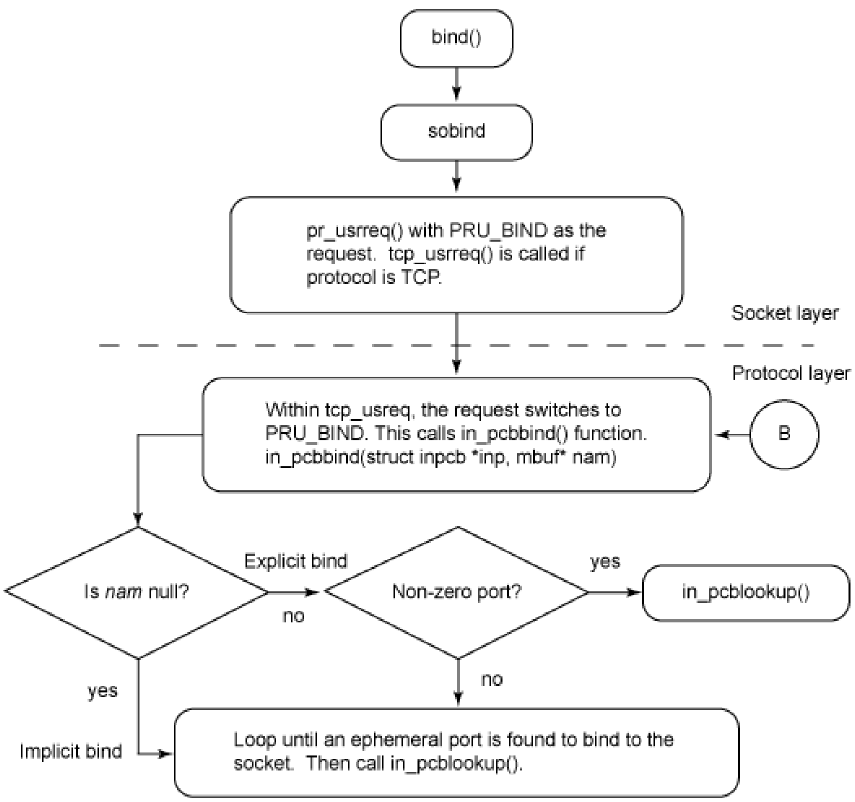
The `bind` system call associates a local network transport address with a socket. For a client process, it is not mandatory to issue a `bind` call. The kernel takes care of doing an implicit binding when the client process issues the connect system call. It is often necessary for a server process to issue an explicit bind request before it can accept connections or start communication with clients.

The `bind` call copies the local address specified by the process into an `mbuf` and calls `sobind`, which in turn calls `tcp_usrreq()` with PRU_BIND as the request. The switch case in `tcp_usrreq()`

calls `in_pcbbind()`, which binds the local address and the port number to the socket. The `in_pcbbind` function first performs some sanity checking to ensure that a socket is not bound twice, and that at least one interface has an IP address assigned. `in_pcbbind` takes care of both implicit and explicit bindings.

If the second argument in the call to `in_pcbbind()` (which is a pointer to the `sockaddr_in` structure) is non-null, then explicit binding happens. Else implicit binding happens. In the case of explicit binding, checks are performed on the IP address being bound, and the socket options are set accordingly.

Figure 4. Call sequence for bind system call



If the local port specified is a non-zero value, a check is made for the super user privilege if the binding is on a reserved port (for example, port number < 1024, per Berkley convention). `in_pcblookup()` is then called to lookup for a control block with the mentioned local IP address and local port number. `in_pcblookup()` verifies if the local address and port pair is not already in use. If the second argument in `in_pcbbind()` is NULL or the local port is zero, then control falls through where it checks to find an ephemeral port (for example, port numbers > 1024 and < 5000, per Berkley convention). `in_pcblookup()` is then called to verify if the found port is unused or not.

Listen

```
listen (struct proc *p, struct listen_args *uap, int *retval)
struct listen_args
{ int s;
   int backlog;
};
```

In the `listen` system call:

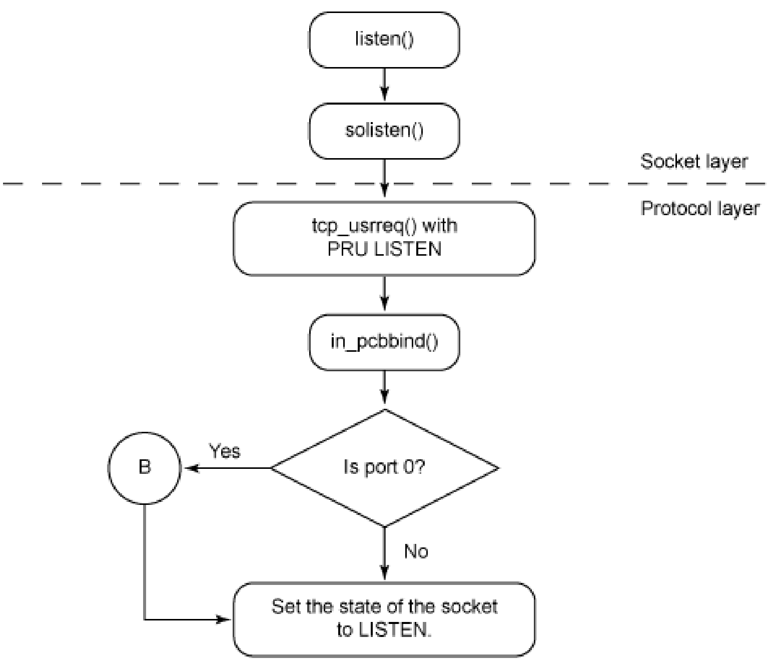
- `s` is the socket descriptor.
- `backlog` is the queue limit for the number of connections on a socket.

The `listen` call indicates to the protocol that the server process is ready to accept any new incoming connections on the socket. There is a limit on the number of connections that can be queued up, after which any further connection requests are ignored.

The `listen` system call calls `solisten` with the socket descriptor and backlog values specified in the `listen` call. `solisten` simply calls the `tcp_usrreq` function with PRU_LISTEN as the request. Within the switch statement of the `tcp_usrreq()` function, the case for PRU_LISTEN checks if the socket is bound to the port. If the port is zero, then `in_pcbbind()` is called to bind the socket to a port (as described in the Bind section).

If there's already a listening socket on a port, then the state of the socket is changed to LISTEN. Normally, all the server processes listen on a well-known port number. It's very rare that `in_pcbbind` gets called to perform an implicit bind for a server process. Figure 5 shows the call sequence for listen.

Figure 5. Call sequence for listen system call



Accept

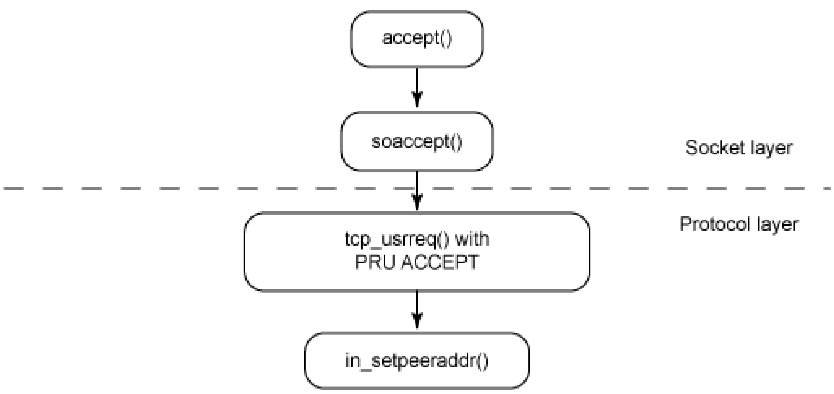
```
accept(struct proc *p, struct accept_args *uap, int *retval);
 struct  accept_args
{
 int s;
 caddr_t name;
 int *anamelen;
};
```

In the `accept` system call:

- `s` is the socket descriptor.
- `name` is a buffer (an OUT parameter), which contains the network transport address of the foreign host.
- `anamelen` is the size of the `name` buffer.

The `accept` system call is a blocking call that waits for incoming connections. Once a connection request is processed, a new socket descriptor is returned by `accept`. This new socket is connected to the client and the other socket `s` remains in LISTEN state to accept further connections.

Figure 6. Call sequence for the accept system call



The `accept` call first validates the arguments and waits for a connection request to arrive. Until then, the function blocks in a while loop. Once a new connection arrives, the protocol layer wakes up the server process. `Accept` then checks for any socket errors that might have occurred when it was blocking. If there were any socket errors, the function returns, and it proceeds further by picking up the new connection from the queue and calls `soaccept`. The `tcp_usrreq ()` function is called in `soaccept()`, with the request as PRU_ACCEPT. The switch in the `tcp_usrreq` function calls `in_setpeeraddr()`, which copies the foreign IP address and foreign port number from the protocol control block and returns these to the server process.

Connect

```
connect (struct proc *p, struct connect_args *uap, int *retval);
struct connect_args
{
    int s;
    caddr_t name;
    int namelen;
};
```

In the `connect` system call:

- `s` is the socket descriptor.
- `name` is the pointer to the buffer that has the foreign IP/port address pair.
- `namelen` is the length of the buffer.

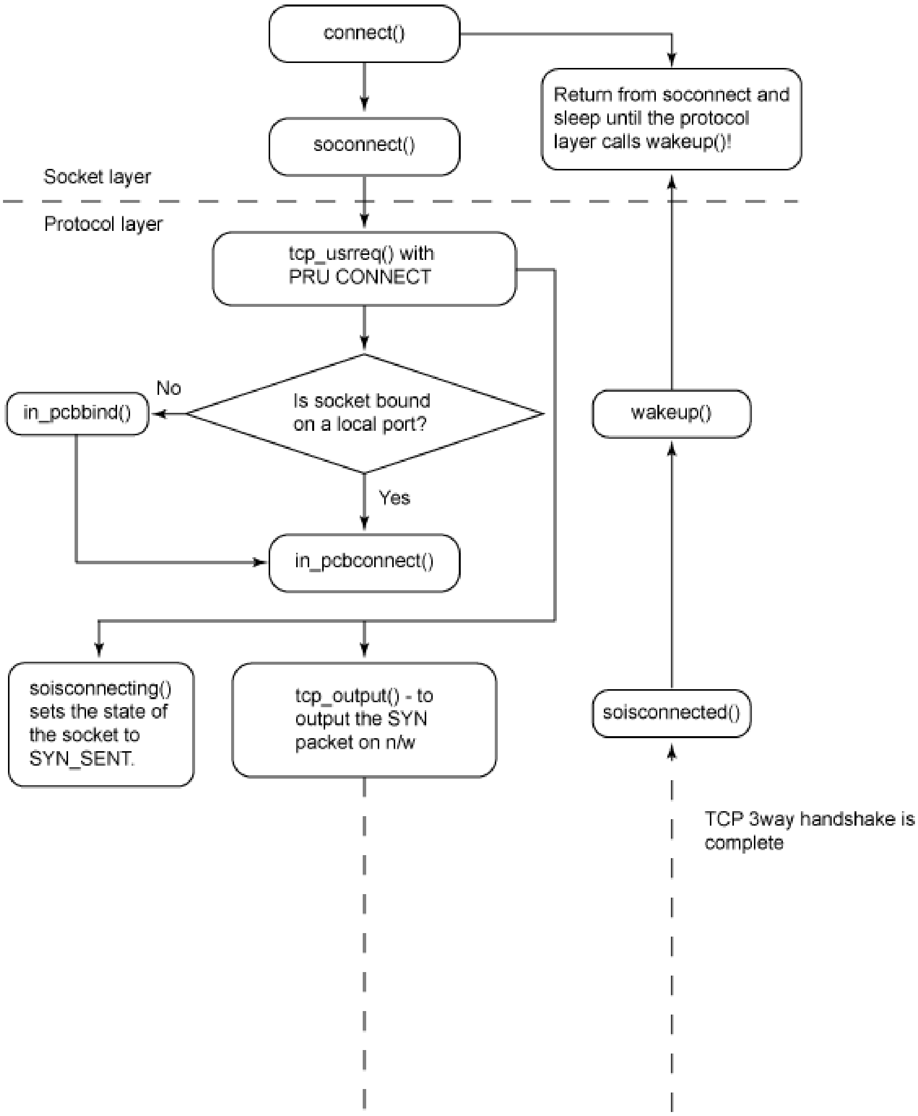
The `connect` system call is normally called by the client process to connect to the server process. If the client process has not explicitly issued a `bind` system call before initiating the connection, implicit binding on the local socket is taken care of by the stack.

The `connect` system call copies the foreign address (the address to which the connection request needs to be sent) from the process to the kernel and calls `soconnect()`. Upon returning from `soconnect()`, the `connect()` function issues a sleep until the protocol layer wakes it up, indicating that the connection is ESTABLISHED or there has been some error on the socket. The `soconnect()` function checks for the valid state of the socket and calls `pr_usrreq()` with PRU_CONNECT as the request.

The switch case in the `tcp_usrreq()` function checks for the binding of a local port with the socket. If the socket is not already bound, `in_pcbbind()` is called, which performs an implicit binding. `in_pcbconnect()` is then called, which gets the route to the destination, finds the interface on which the packet has to be output, and verifies that the foreign socket pair (IP address and port number) specified by the `connect()` is unique or not. Then, it updates its Internet control block with the foreign IP address and port number and returns to the PRU_CONNECT case statement.

`tcp_usrreq ()` now calls `soisconnecting ()`, which sets the state of the socket on the client host as SYN_SENT. The function `tcp_output` is called, which outputs the SYN packet onto the network. The control now returns to the `connect()` function, which sleeps until the protocol layer wakes up—indicating that the connection is now ESTABLISHED or that there has been an error on the socket.

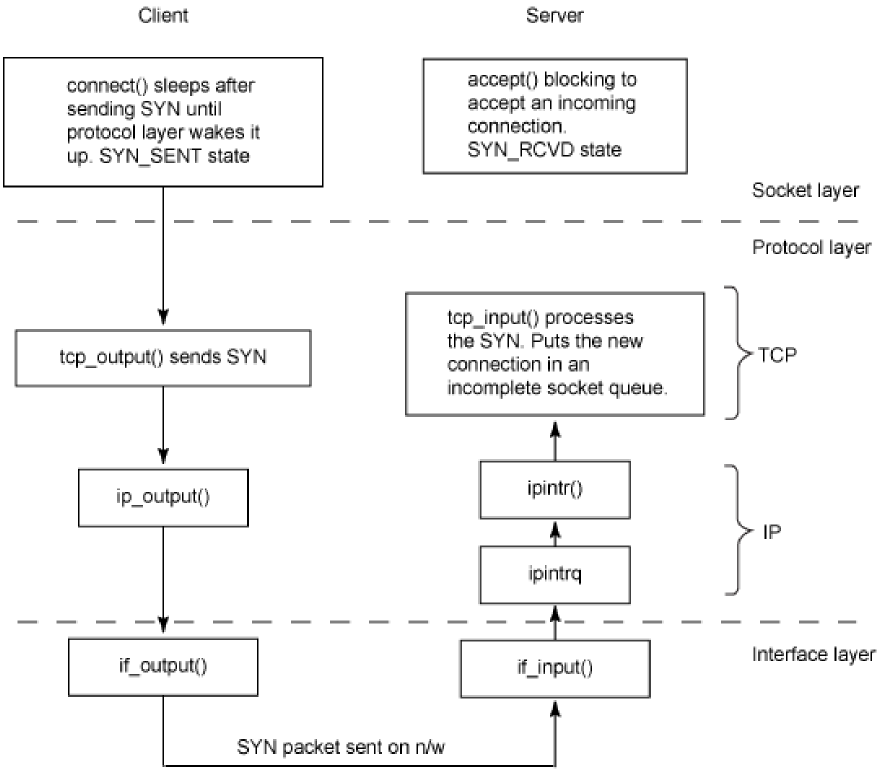
Figure 7. Call sequence for connect system call



The 3-way TCP handshake

Figure 8, Figure 9, and Figure 10 show the call sequence when the client issues `connect`, and the server issues `accept` to initiate and establish a TCP connection.

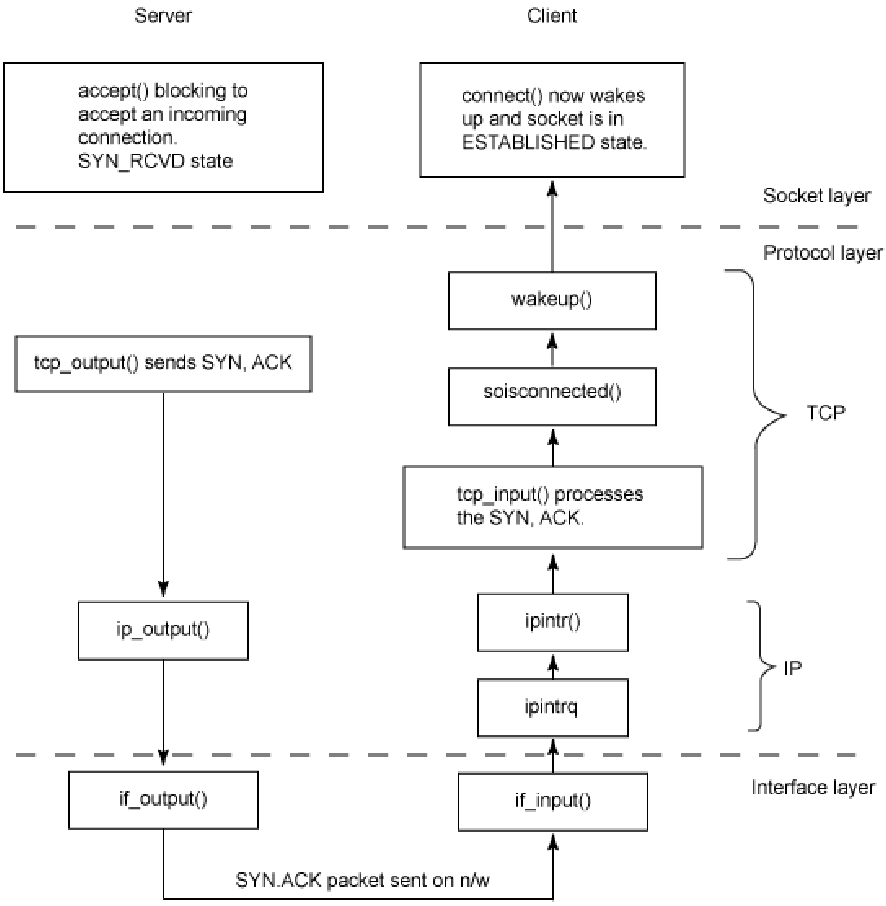
Figure 8. Flow sequence for a SYN packet



When the client issues `connect`, the `tcp_output()` function is called at the protocol layer, which outputs the SYN packet onto the interface. As shown in Figure 9 below, the `soconnect` now returns to the `connect()` function and sleeps. The socket state on the client side now is SYN_SENT. The interface layer calls `if_output()` (actually the interface-specific output function), which sends the packet onto the n/w.

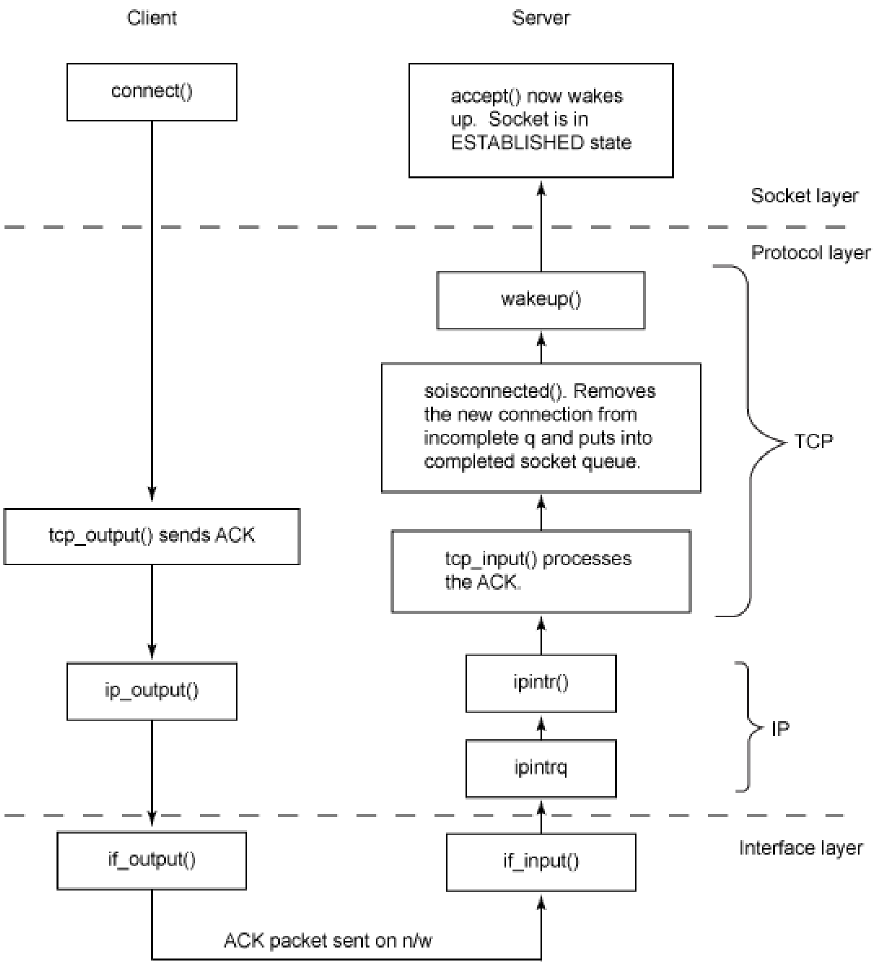
The interface on the destination (the server) receives the incoming SYN packet, places it on the ipintrq queue, and raises a software interrupt. The packet is then picked up by the `ipintr()`, which calls the `tcp_input` routine. `tcp_input()` executes at the s/w interrupt, and picks up the SYN packet from ipintrq, processes it, and places the partially completed socket connection in an incomplete socket queue. The socket state on the server side now is SYN_RCVD. After each processing, the `tcp_input()` routine calls `tcp_output()` if a response packet needs to be sent to the other end.

Figure 9. Flow sequence for a SYN, ACK Packet



The server, after processing the SYN, sends a SYN ACK packet using the `tcp_output ()`, `ip_output ()`, and `if_output ()` sequence. The n/w interface on the client receives this packet, places it on the ipintrq, and raises the s/w interrupt. Likewise, `ipintr ()` picks up the packet from the ipintrq, and passes it to the `tcp_input ()` routine on the client side TCP stack. The packet is now processed and `soisconnected ()` is called, which wakes up the connect call. The socket state on the client side now is established.

Figure 10. Flow sequence for ACK packet



The `tcp_input ()` routine on the client side processes the SYN ACK packet, and calls `tcp_output ()` to send an ACK packet back to the server. The `tcp_input ()` on the server side processes this ACK packet and calls `soisconnected ()`. This function removes the socket from the incomplete socket queue and puts it into the completed socket queue. `Wakeup ()` is then called to wake up the accept call. The socket on the server side now is established.

Shutdown

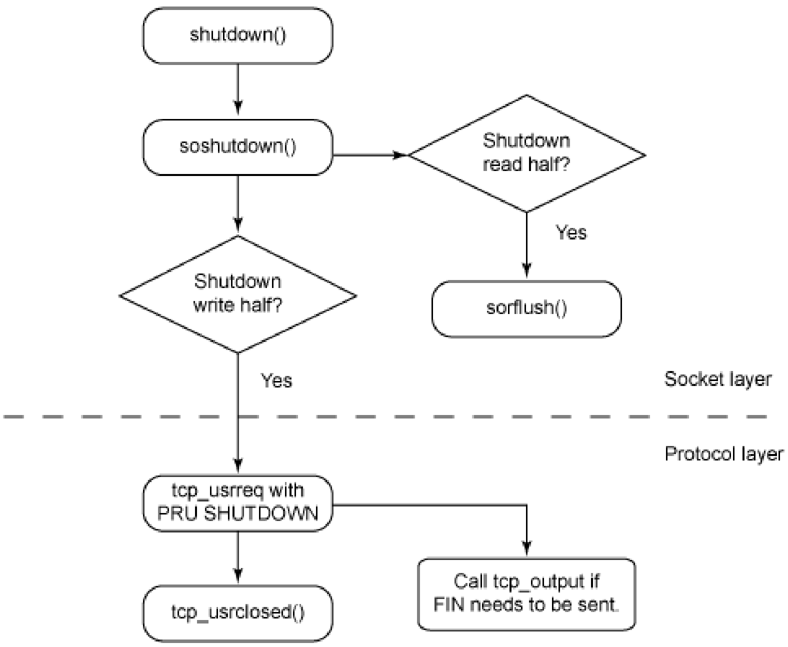
```
shutdown (struct proc *p, struct shutdown_args *uap, int *retval);
Struct shutdown_args
{
 int s;
 int how;
}
```

In the `shutdown` system call:

- `s` is the socket descriptor.
- `how` specifies which half of the connection is to be closed. A value of zero, one, and two for `how` specifies to close the read half, write half, and both halves of the connection, respectively.

The `shutdown` system call closes either one or both ends of the connection. If it is the read half that needs to be closed, then any data existing in the receive buffer is discarded and that end of the connection is closed. For the write half, TCP sends any leftover data and then terminates the write end of the connection.

Figure 11. Call sequence for shutdown system call



If the read half of the connection needs to be shutdown, the `soshutdown()` function calls `sorflush()`. `sorflush()` marks the socket to reject any incoming packets and any system resources held are released.

If the write half of the connection needs to be closed, then `tcp_usrreq()` is called with PRU_SHUTDOWN as the request. The switch case for PRU_SHUTDOWN calls the `tcp_usrclosed()` function, which updates the state of the socket, depending on the current state. The TCP/IP state diagram is helpful in understanding the different states a socket can exist in at any given time. If a FIN needs to be sent upon returning from `tcp_usrclosed()`, then `tcp_output()` is called, which sends it on to the interface.

Close

```
soo_close(struct file *fp , struct proc *p);
```

In the `close` system call:

- `fp` is the pointer to the file structure.
- `p` is the pointer to the proc structure of the calling process.

The `close` system call closes or aborts any pending connections on the socket.

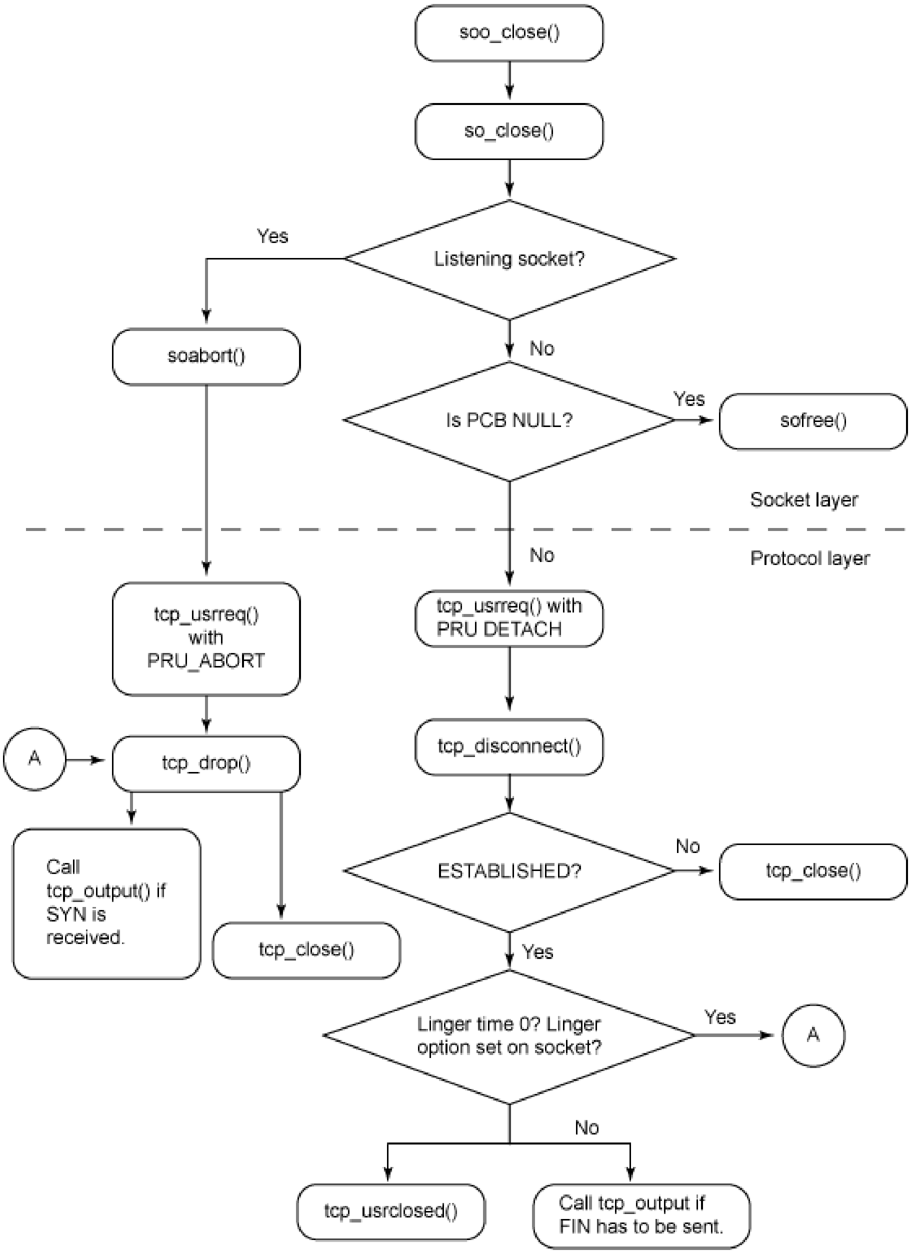
The `soo_close()` simply calls the `so_close()` function, which first checks if the socket to be closed is a listening socket (socket that's accepting incoming connections). If it is, then the two socket queues are traversed to check for any pending connections. For each pending connection, `soabort()` is called, which issues `tcp_usrreq()` with PRU_ABORT as the request. This switch case calls `tcp_drop()`, which checks the state of the socket.

If the state is SYN_RCVD, then a RST segment is sent by setting the state to CLOSED and calling `tcp_output()`. The socket is then closed by the `tcp_close()` function. The `tcp_close` function updates three variables of the routing metrics structure, and then releases the resources held by the socket.

If the socket is not a listening socket, then control falls to `soclose()` to check if there is already a control block attached to the socket. If not, the socket is freed by `sofree()`. If yes, `tcp_usrreq()` with PRU_DETACH is called to detach the protocol from the socket. The switch case for PRU_DETACH calls `tcp_disconnect()` to check if a connection state is ESTABLISHED. If not, `tcp_disconnect()` calls `tcp_close()`, which releases the Internet and control blocks. `tcp_disconnect()` otherwise checks for the linger time and linger socket option. It calls `tcp_drop()` if the option is set and the linger time is zero. If not, `tcp_usrclosed()` is called, which sets the state of the socket and calls `tcp_output()` if a FIN segment needs to be sent.

Figure 12 shows the important calls that occur when the close system call is issued by a TCP application.

Figure 12. Call sequence for close system call



Send

```
sendmsg ( struct proc*p, struct sendmsg_args *uap, int retval);
struct sendmsg_args
{
   int s;
   caddr_t msg;
   int flags;
};
```

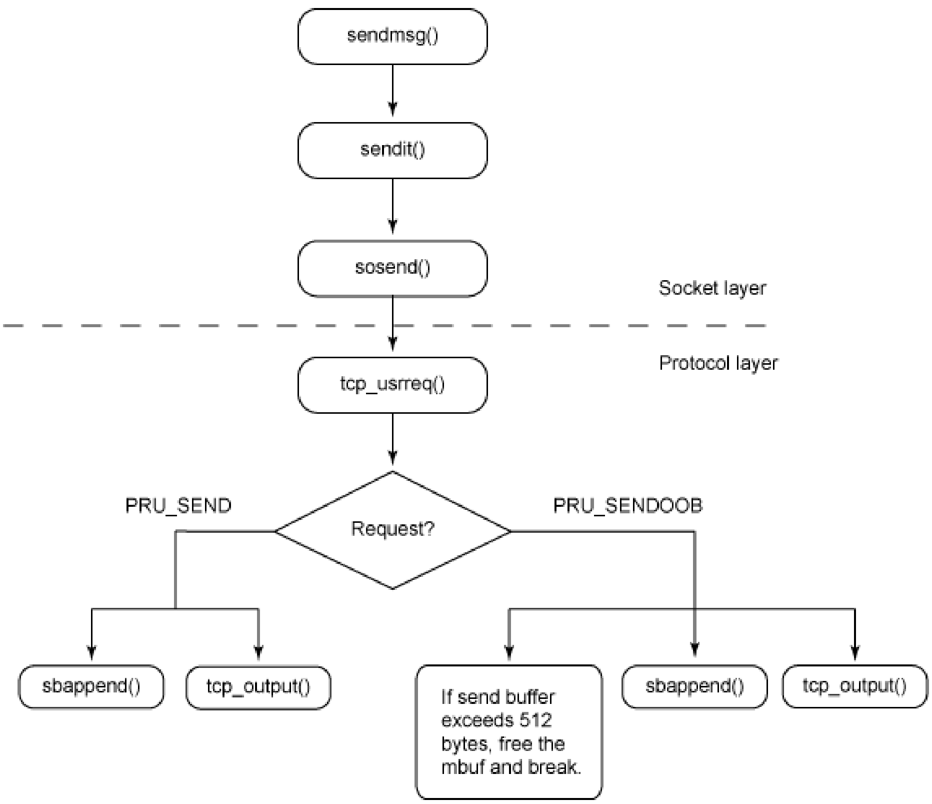
In the `send` system call:

- `s` is the socket descriptor.

- `msg` is a pointer to the `msghdr` structure.
- `flags` is the control information.

There are four system calls to send data on the n/w interface: `write`, `writev`, `sendto`, and `sendmsg`. This article discusses only the `sendmsg()` system call. All four send calls eventually call `sosend()`. While `send` (a library function called by the process), `sendto`, and `sendmsg` system calls can operate only on the socket descriptor, the `write` and `writev` system calls can operate on any kind of descriptor.

Figure 13. Call Sequence for sendmsg



The `sendmsg` system call copies the message to be sent from the process to the kernel space and calls `sendit()`. In `sendit()`, a structure is initialized to collect the output from the process into memory buffers in the kernel. The address and control information are also copied from the process to the kernel. `sosend()` is then called, which performs four tasks:

- Initializes various parameters based on the values passed by the `sendit()` function.
- Validates the conditions of the socket and the state of the connection, and determines the space needed to pass on the message and report errors.
- Allocates memory and copies the data from the process.
- Makes the protocol specific call to send the data on to the network.

The `tcp_usrreq()` is then called and, depending on the flags specified by the process, the control switches to PRU_SEND or PRU_SENDOOB (to send out of band data). In the case of

PRU_SENDOOB, the send buffer size can exceed up to 512 bytes more in which any allocated memory is freed and control breaks. Otherwise the `sbappend()` and `tcp_output()` functions are called by both PRU_SEND and PRU_SENDOOB cases. `sbappend()` adds the data at the end of send buffer and `tcp_output()` sends the segment onto the interface.

Receive

```
recvmsg(struct proc *p, struct recvmsg_args *uap , int *retval);
struct recvmsg_args
{
 int s,
 struct msghdr *msg,
 int flags,
};
```

In the `receive` system call:

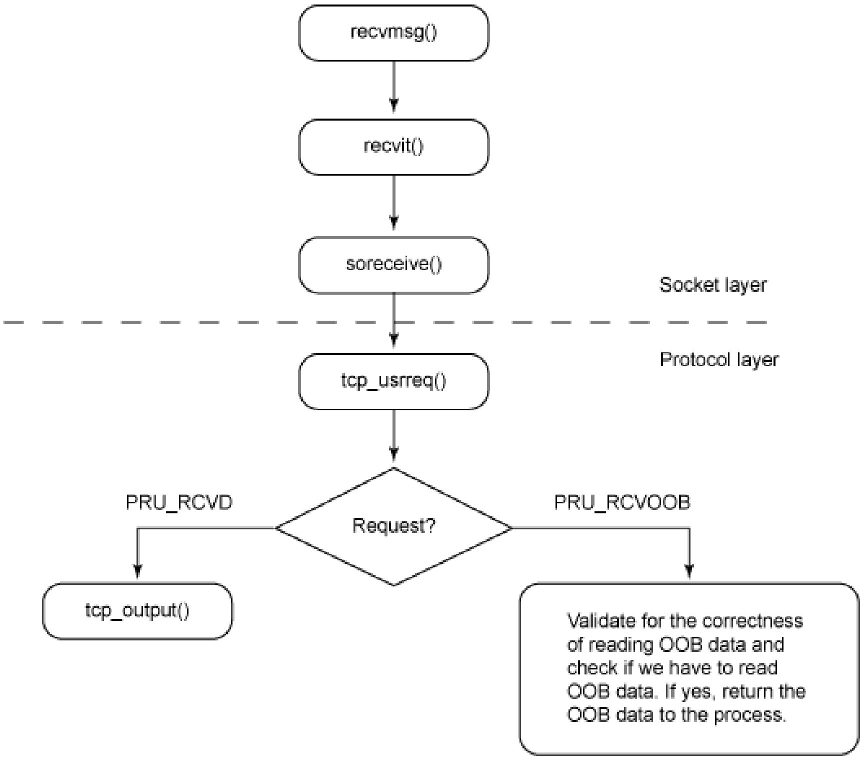
- `s` is the socket descriptor.
- `msg` is a pointer to the `msghdr` structure.
- `flags` specify the control information.

There are four system calls that can be used to receive data from a connection: `read`, `readv`, `recvfrom`, and `recvmsg`. While `recv` (a library function used by the process), `recvfrom`, and `recvmsg` operate only on socket descriptor, `read` and `readv` can operate on any kind of descriptor. All the `read` system calls eventually call `soreceive()`.

Figure 14 shows the sequence of calls for the `recvmsg` system call. The `recvmsg()` and `recvit()` functions initialize various arrays and structures to send the received data to the process from the kernel. `recvit()` calls `soreceive()`, which transfers received data from socket buffers to the receive buffers process. The `soreceive()` function performs various checks, such as:

- Whether the MSG_OOB flag is set.
- If the process is attempting to receive data.
- Should it block until enough data has arrived.
- Transfer the read data to the process.
- Check if the data is out of band data or regular, and handle accordingly.
- Notify the protocol when the data reception is complete.

Figure 14. Call sequence for recvmsg



The `soreceive()` function makes the protocol-dependant requests when either the MSG_OOB flag is set or when the data reception is complete. In the case of receiving out-of-band data, the protocol layer checks for different conditions to validate that the received data is OOB, and then returns it to the socket layer. In the latter case, the protocol layer calls `tcp_output()`, which sends a window update segment on to the network. This notifies the other end about any space that is available to receive data.

Conclusion

In this article, you learned about the most important TCP function calls that trigger low-level calls to accomplish certain things. The call sequences in the figures showed a broad overview of the kernel-level TCP calls. Use this article a good starting point in understanding the organization of the FreeBSD TCP/IP stack.

Related topics

- *TCP/IP Illustrated, Volume 2, The Implementation*, ISBN-10: 0-201-63354-X, by Gary R. Wright and W. Richard Stevens: This book discusses protection keys used in application space. For more detailed information on inpcb and tcpcb structures, see Chapter 22.
- Search the AIX and UNIX library by topic:
  - System administration
  - Application development
  - Performance
  - Porting
  - Security
  - Tips
  - Tools and utilities
  - Java™ technology
  - Linux®
  - Open source
- AIX and UNIX: The AIX and UNIX developerWorks zone provides a wealth of information relating to all aspects of AIX systems administration and expanding your UNIX skills.
- AIX Wikis: Discover a collaborative environment for technical information related to AIX.
- IBM trial software: Build your next development project with software for download directly from developerWorks.
- developerWorks technical events and webcasts: Stay current with developerWorks technical events and webcasts.
- Podcasts: Tune in and catch up with IBM technical experts.