TCP

CSCI780/420: Linux Kernel Internals
Spring 2000

Song Jiang

Outline:

- TCP and its features;
- 3-way handshake and State Transition Diagram;
- Major data structures and their relationship;
- How is a connetion established;
- How is data stream transmitted along the connection;
- TCP timer.

TCP and its features

The **Transmission Control Protocol**, or TCP, provides a connection-oriented, reliable, flow-controlled and ordered byte-stream service between the two end points of an application. This is completely different from UDP's connectionless, unreliable, datagram service.

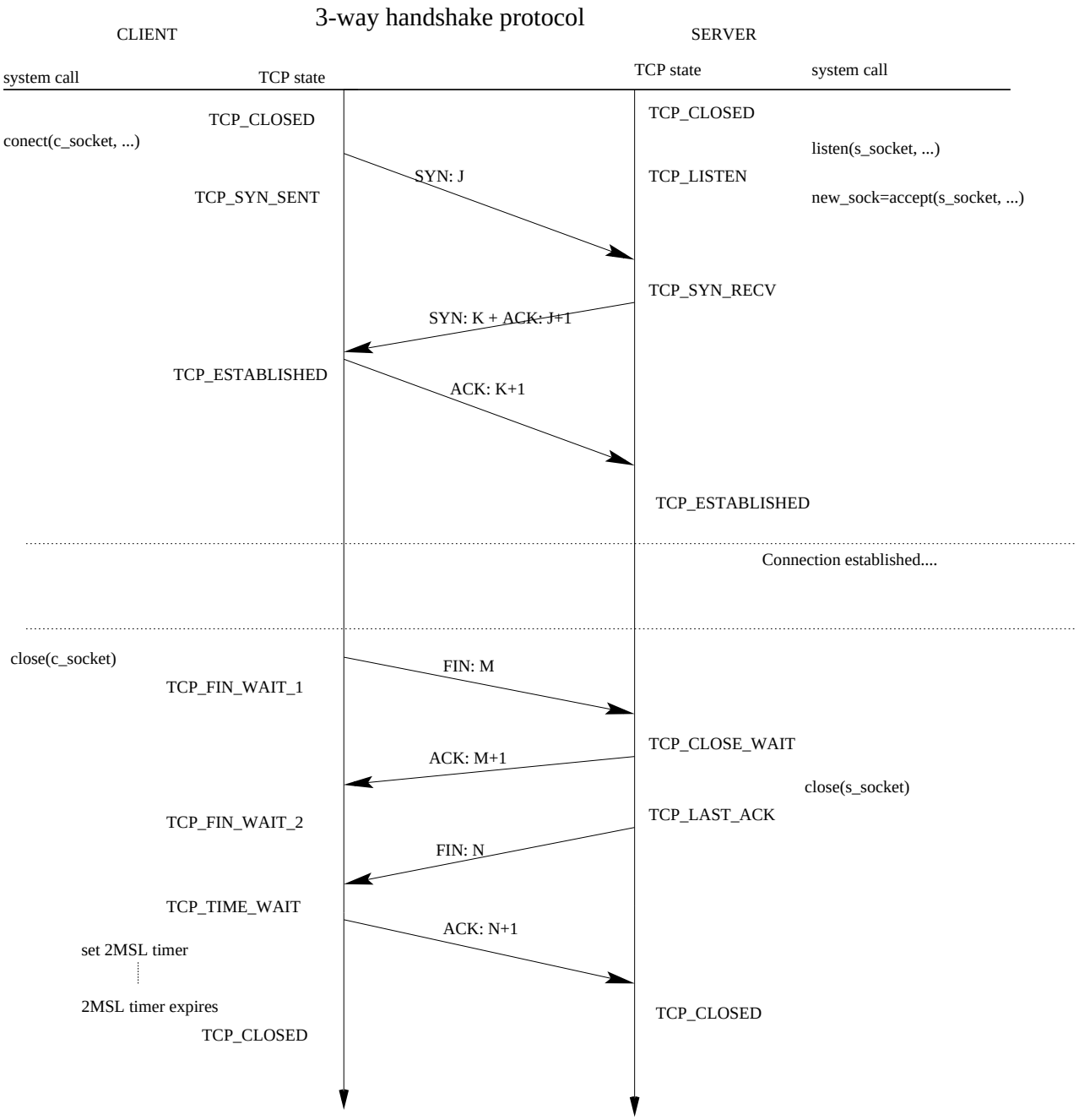
Under the virtual circuit model, the life of a connection in TCP is divided into three distinct phases: (1) opening the connection to create a full-duplex byte stream; (2) transferring data in one or both directions over this stream; and (3) closing the connection. Remote login and file transfer are examples of applications that are well suited to virtual-circuit service.

TCP is the most complex protocol in the suite of protocols. It uses the unreliable IP mechanism for communication. Like most reliable transport protocols, TCP uses `time with retransmission` to achieve reliability.

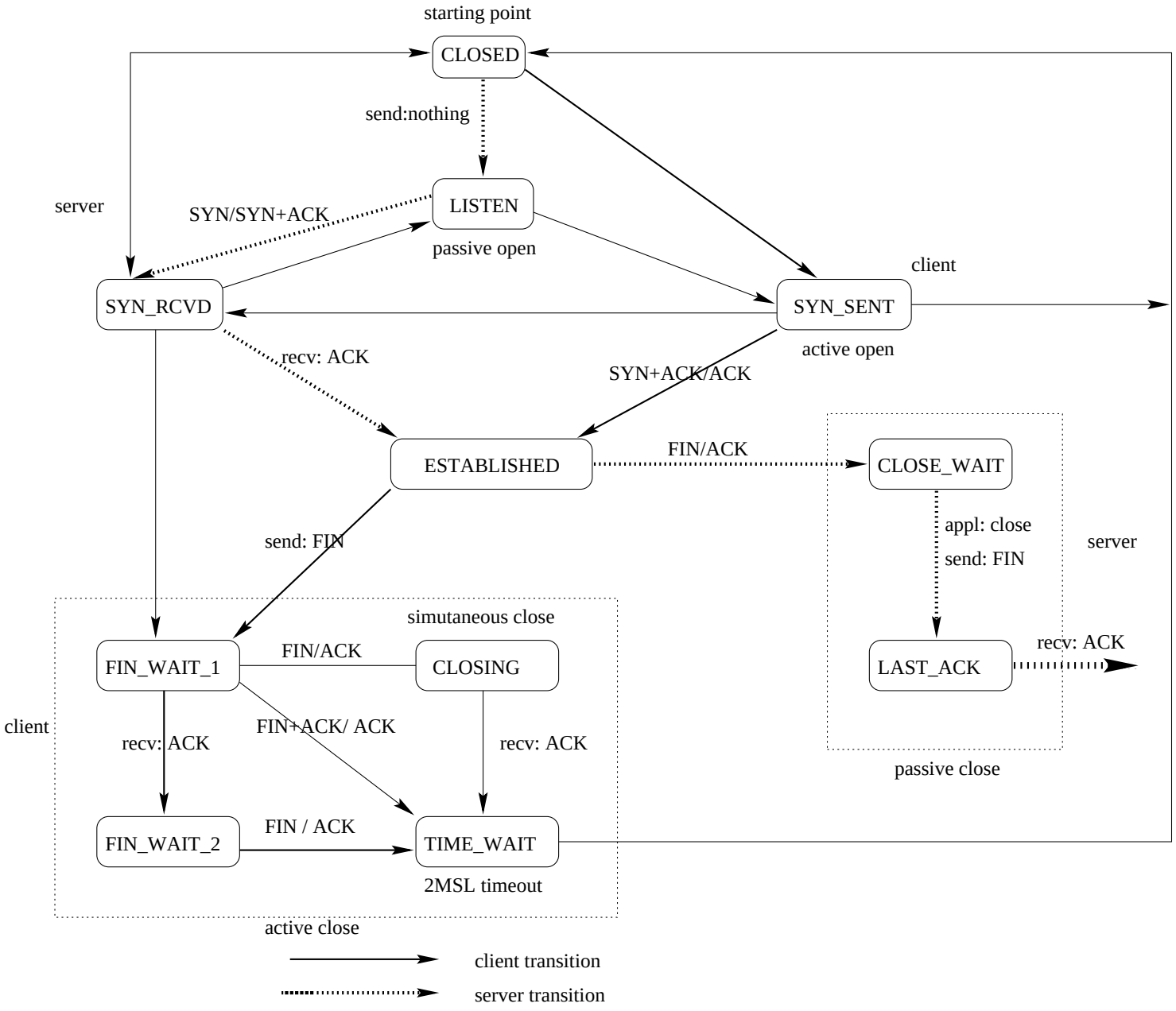
3-way handshake and State Transition Diagram

Many of TCP's actions, in reponse to different types of segments arriving on a connection, can be summarized in a state transition digram.

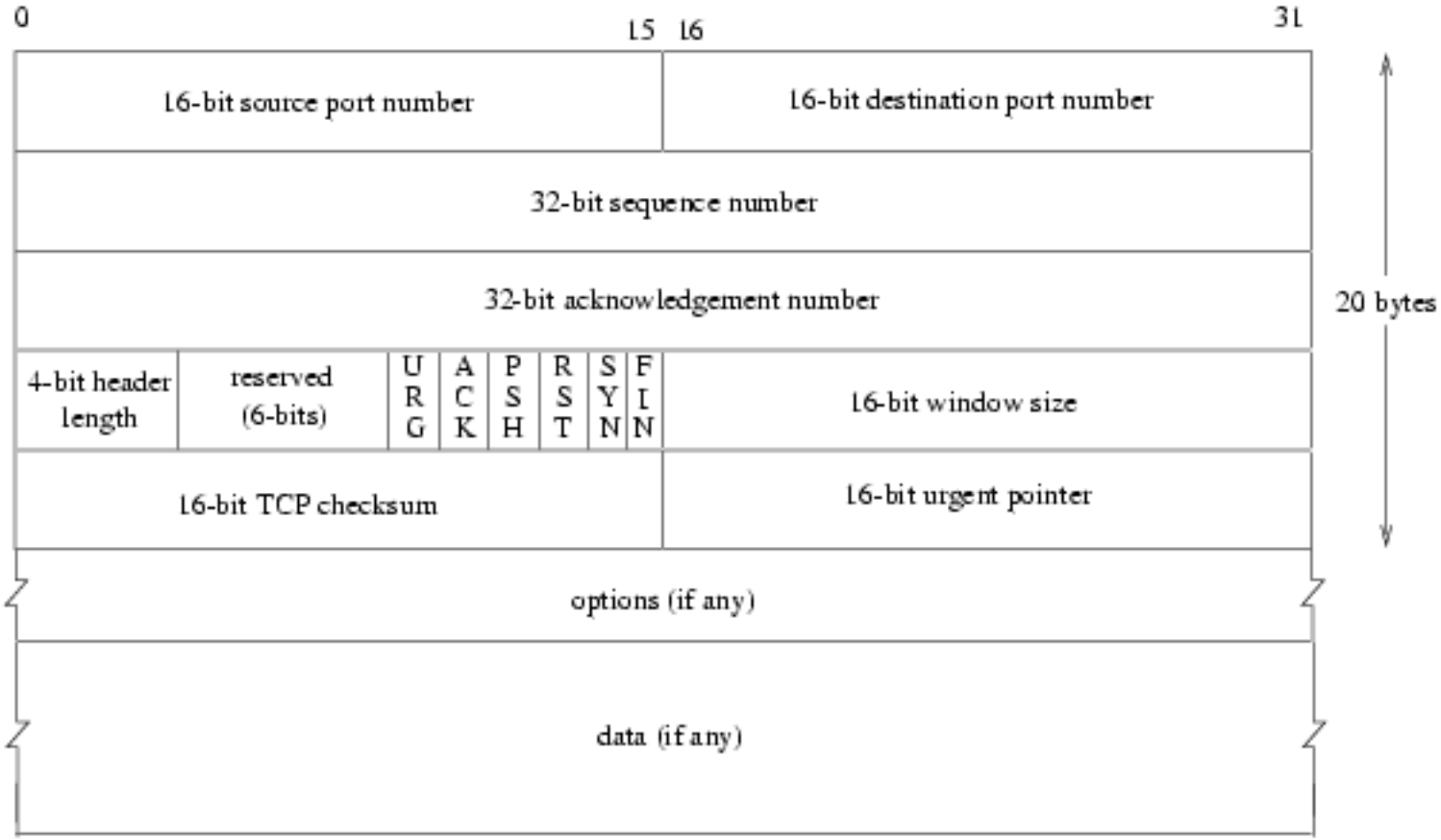
Three-way handshake protocol is untilized to faciltate to set up and close the connection.


3-way handshake protocol
CLIENT
SERVER
system call
TCP state
TCP state
system call
TCP_CLOSED
TCP_CLOSED
conect(c_socket, ...)
listen(s_socket, ...)
SYN: J
TCP_LISTEN
TCP_SYN_SENT
new_sock=accept(s_socket, ...)
TCP_SYN_RECV
SYN: K + ACK: J+1
TCP_ESTABLISHED
ACK: K+1
TCP_ESTABLISHED
Connection established....
close(c_socket)
FIN: M
TCP_FIN_WAIT_1
TCP_CLOSE_WAIT
ACK: M+1
close(s_socket)
TCP_FIN_WAIT_2
TCP_LAST_ACK
FIN: N
TCP_TIME_WAIT
ACK: N+1
set 2MSL timer
2MSL timer expires
TCP_CLOSED
TCP_CLOSED

TCP State Transition Diagram



The TCP Packet Format

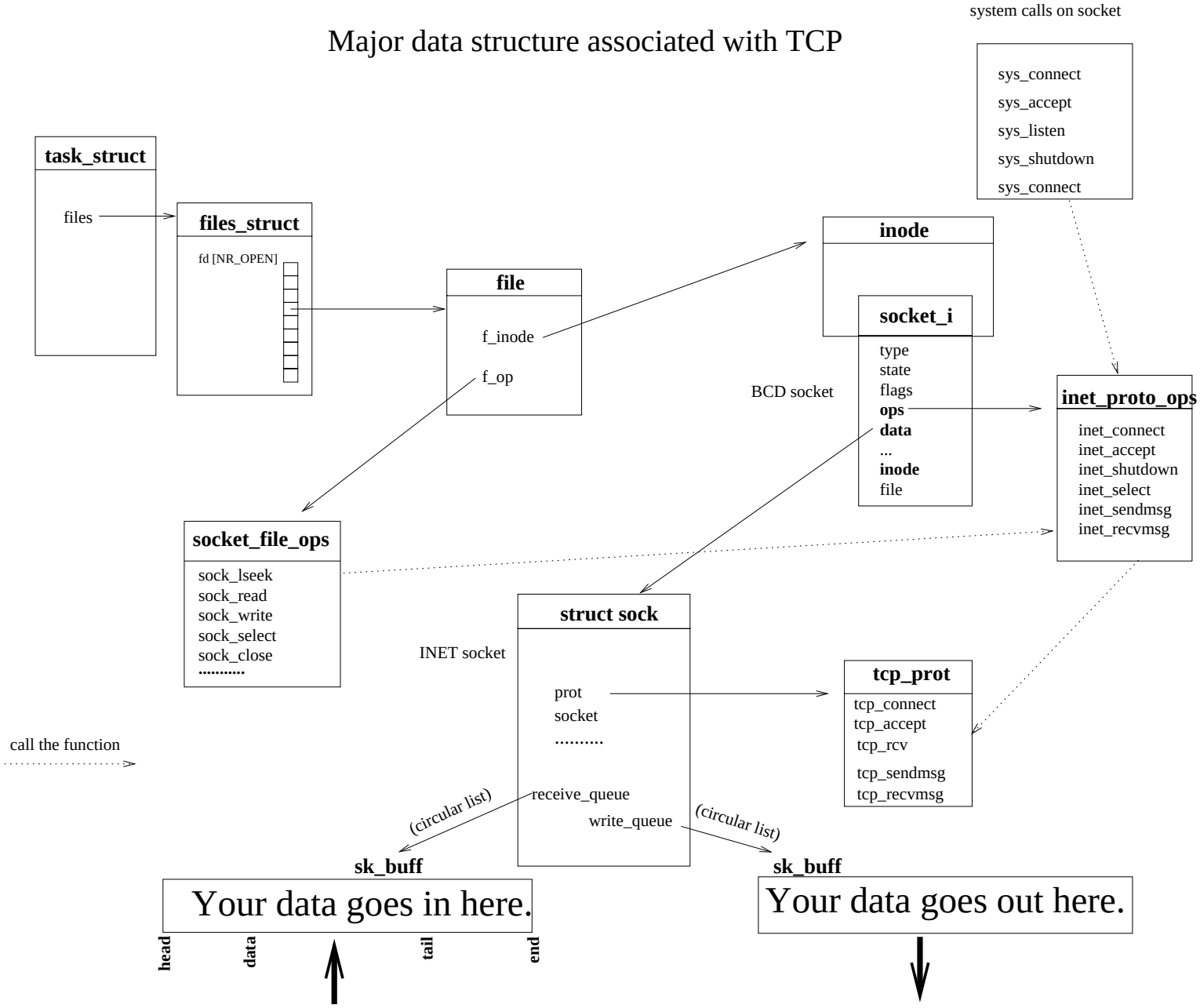


Major data structures and their relationship

Linux supports several classes of socket and these are known as address families. This is because each class has its own method of addressing its communications. INET (the Internet address family) is one of these families. At kernel initialization time, the address families built into the kernel register themselves with the BSD socket interface. These are several socket types and these represent the type of service that supports the connection. Linux BSD sockets support a number of socket types: TCP, UDP, RAW.

The following diagram gives a data structure frame for TCP to work.

Major data structure associated with TCP



The socket structure forms the basis for the implementation of the BSD socket interface. It is set up and initialized when the system call sys_socket.

Linux/include/linux/net.h

```
71 struct socket {
72   short                type;          /* SOCK_STREAM, SOCK_DGRAM,...  */
73   socket_state         state;         /* SS_CONNECTED, SS_UNCONNECTED...*/
74   long                 flags;
75   struct proto_ops     *ops;          /* protocols do most everything */
76   void                 *data;         /* protocol data                */
79   struct socket        *next;
81   struct inode         *inode;
82   struct fasync_struct *fasync_list;  /* Asynchronous wake up list    */
83   struct file          *file;         /* File back pointer for gc     */
84 };
```

ops points to the operation vector inet_proto_ops in AF_INET, where the specific operations for this address family are entered. The data points to the INET socket – substructure of the socket corresponding to the address family(Here is AF_INET).

```
 88 struct proto_ops {
 89   int   family;
 90
 91   int   (*create)       (struct socket *sock, int protocol);
 93   int   (*release)      (struct socket *sock, struct socket *peer);
 94   int   (*bind)         (struct socket *sock, struct sockaddr *umyaddr,
 95                          int sockaddr_len);
 96   int   (*connect)      (struct socket *sock, struct sockaddr *uservaddr,
 97                          int sockaddr_len, int flags);
 99   int   (*accept)       (struct socket *sock, struct socket *newsock,
107   int   (*listen)       (struct socket *sock, int len);
108   int   (*shutdown)     (struct socket *sock, int flags);
115   int   (*sendmsg)      (struct socket *sock, struct msghdr *m, int total_len, int nonblock,
116   int   (*recvmsg)      (struct socket *sock, struct msghdr *m, int total_len, int nonblock,
117 };
```

In the INET data structure the network-specific parts of the sockets are administered. This is required for TCP, UDP and RAW sockets.

Linux/include/net/sock.h

```
161 struct sock
162 {
171         __u32                   write_seq, sent_seq, acked_seq, copied_seq, rcv_ack_seq;
176         unsigned short          rcv_ack_cnt;            /* count of same ack */
177         __u32                   window_seq, fin_seq, urg_seq, syn_seq;
203         struct device           * bound_device;
207         struct sock             *next;
211         struct sock             *pair;
213         struct sock             *prev;
221         struct sk_buff_head     write_queue, receive_queue;
223         struct proto            *prot;
224         struct wait_queue       **sleep;
225         __u32                   daddr;
226         __u32                   saddr;          /* Sending source */
227         __u32                   rcv_saddr;      /* Bound address */
264         unsigned char           protocol;
265         volatile unsigned char  state;
271         unsigned short          type;
310-        struct tcphdr           dummy_th;
338         struct socket           *socket;
    }
```

```
361 struct proto
362 {
367         void                    (*close)(...);
368         int                     (*build_header)(.....);
374         int                     (*connect)(.....);
376         struct sock *           (*accept) (.....);
377         void                    (*queue_xmit)(......);
383         int                     (*rcv)(....);
392         void                    (*shutdown)(....);
397         int                     (*sendmsg)(....);
399         int                     (*recvmsg)(....);
401         int                     (*bind)(....);
412         char                    name[32];
414 };
```

Every protocol must provide an `rcv` function, to which the packets received by the lower layers are passed. This function is entered in the associated `inet_protocol` structure for each of the IP-based protocols.

net/ipv4/tcp.c

```
2410 struct proto tcp_prot = {
```

```
2411        (struct sock *)&tcp_prot,       /* sklist_next */
2412        (struct sock *)&tcp_prot,       /* sklist_prev */
2413        tcp_close,                      /* close */
2414        ip_build_header,                /* build_header */
2415        tcp_connect,                    /* connect */
2416        tcp_accept,                     /* accept */
2417        ip_queue_xmit,                  /* queue_xmit */
2418        tcp_retransmit,                 /* retransmit */
2421        tcp_rcv,                        /* rcv */
2425        tcp_shutdown,                   /* shutdown */
2428        tcp_sendmsg,                    /* sendmsg */
2429        tcp_recvmsg,                    /* recvmsg */
2438        "TCP",                          /* name */
2441 };
2442
```

How is a connetion established?

Processes that communicate using sockets use a client server model. All the system calls specific to TCP/IP starts here:

```
1278 asmlinkage int sys_socketcall(int call, unsigned long *args)
1279 {       ... ...
1297         switch(call)
1298         {
1299                 case SYS_SOCKET:
1300                         return(sys_socket(a0,a1,get_user(args+2)));
1304                 case SYS_CONNECT:
1305                         return(sys_connect(a0, (struct sockaddr *)a1,
1306                                            get_user(args+2)));
1307                 case SYS_LISTEN:
1308                         return(sys_listen(a0,a1));
1309                 case SYS_ACCEPT:
1310                         return(sys_accept(a0,(struct sockaddr *)a1,
1311                                           (int *)get_user(args+2)));
1367         }
1368         return -EINVAL; /* to keep gcc happy */
1369 }
```

At first let's create a socket associated with an address family(AF_INET), a socket type(SOCK_STREAM) and a protocol(IPPROTO_TCP). The following function prepares a new socket and its associated data structure, from file descriptor to INET socket.

```
559 asmlinkage int sys_socket(int family, int type, int protocol)
560 {
561         int i, fd;
562         struct socket *sock;
563         struct proto_ops *ops;
564
565         /* Locate the correct protocol family. */
566         i = find_protocol_family(family);
584         ops = pops[i];
597 /*
598  *      Allocate the socket and allow the family to set things up. if
599  *      the protocol is 0, the family is instructed to select an appropriate
600  *      default.
601  */
603         if (!(sock = sock_alloc()))
               ... ...
609
610         sock->type = type;
```

```
611         sock->ops = ops;

            /*-- sock.data <-- an allocated INET socket
              -- initialize the inet socket state with 'TCP_CLOSE'
             */
612         if ((i = sock->ops->create(sock, protocol)) < 0)
               ... ...
617
            /*-- struct socket --> inode --> fd
             *-- Obtains the first available file descriptor and sets it up for use.
             */
618         if ((fd = get_fd(SOCK_INODE(sock))) < 0)
               ... ...
624         sock->file=current->files->fd[fd];
626         return(fd);
627 }
```

Then, bind socket address to the newly created socket by calling

```
asmlinkage int sys_bind(int fd, struct sockaddr *umyaddr, int addrlen)
```

(the socket's name or address is specified using the sockaddr data structure, An INET socket would have an IP port address bound to it. The registered port numbers can be seen in /etc/services)

Having bound an address to the socket, the server then listens for incoming connection requests specifying the bound address. In the function

```
asmlinkage int sys_listen(int fd, int backlog)
```

`SO_ACCEPTCON` is entered `sock->flags`(BSD socket), INET socket is set with state `TCP_LISTEN`. Now it is waiting for a connection request to come.

The originator of the request, the client, also creates a socket and makes a connection request on it by specifying the target address and its port number.

An outbound connection can only be made when it doesn't already have a connection established and is not being used for listening.

Client requests a connection by sending out TCP_SYN_SENT through calling `sys_connect`. This is the first packet in 3-way handshake.

```
828 asmlinkage int sys_connect(int fd, struct sockaddr *uservaddr, int addrlen)
829 {
830         struct socket *sock;
831         char address[MAX_SOCK_ADDR];
832         int err;
834         if (!(sock = sockfd_lookup(fd, &err))) <--- fetch the socket from inode
835                 return(err);
840         switch(sock->state)
841         {
842                 case SS_UNCONNECTED:/* This is ok... continue with connect */
845                 case SS_CONNECTED:  ... ...
851                 case SS_CONNECTING:  ... ...
863         }
864         err = sock->ops->connect(sock, (struct sockaddr *)address,
addrlen, sock->file->f_flags);
            ... ...
869         return err;
870 }
```

Now we use the AF_INET handler to do this from BCD socket.

```
673 static int inet_connect(struct socket *sock, struct sockaddr * uaddr,
674                   int addr_len, int flags)
675 {
676         struct sock *sk=(struct sock *)sock->data;
677         int err;
678         sock->conn = NULL;

692         if (sock->state != SS_CONNECTING) {
698                 err = sk->prot->connect(sk, (struct sockaddr_in *)uaddr, addr_len);
701                 sock->state = SS_CONNECTING;
702         }

712         cli(); /* avoid the race condition */
713         while(sk->state == TCP_SYN_SENT || sk->state == TCP_SYN_RECV) {
714                 interruptible_sleep_on(sk->sleep);
                  .......
726         }
727         sti();
728         sock->state = SS_CONNECTED; <-- Now the connection is done
729
```

```
730         if (sk->state != TCP_ESTABLISHED && sk->err) {
731                 sock->state = SS_UNCONNECTED;
732                 return sock_error(sk);
733         }
734         return(0);
735 }
```

TCP prtocol is ready to process the connection requests. This time the argument is INET socket.

```
2174 static int tcp_connect(struct sock *sk, struct sockaddr_in *usin, int addr_len)
2175 {
2176         struct sk_buff *buff;
2177         struct device *dev=NULL;
2178         unsigned char *ptr;
2179         int tmp;
2180         int atype;
2181         struct tcphdr *t1;
2182         struct rtable *rt;
2183
2184         if (sk->state != TCP_CLOSE)
2185                 return(-EISCONN);
```

```
2196
2197        if (usin->sin_family && usin->sin_family != AF_INET) <--tcp is in the family ‘‘AF_I
2198                return(-EAFNOSUPPORT);
2199
2217        lock_sock(sk);
2218        sk->daddr = usin->sin_addr.s_addr;
2222        sk->dummy_th.dest = usin->sin_port;  <--- fill tcp header
2223
2224        buff = sock_wmalloc(sk,MAX_SYN_SIZE,0, GFP_KERNEL);
2230        buff->sk = sk;
2240
2241        tmp = sk->prot->build_header(buff, sk->saddr, sk->daddr, &dev,
2242                IPPROTO_TCP, sk->opt, MAX_SYN_SIZE,sk->ip_tos,sk->ip_ttl,&sk->ip_route_cach
2251        sk->rcv_saddr = sk->saddr;  /* Bound address */
2252
2262        t1 = (struct tcphdr *) skb_put(buff,sizeof(struct tcphdr));
2264        memcpy(t1,(void *)&(sk->dummy_th), sizeof(*t1));<-- copy tcp header from ’sk’ to sc
2269        t1->ack = 0;
2270        t1->window = 2;
2271        t1->syn = 1;  <--- SYN
2272        t1->doff = 6;
2313        tcp_set_state(sk,TCP_SYN_SENT); <-- aha! enter TCP_SYN_SENT via active open
```

```
 /* sent a connection request, waiting for ack. ip_queue_xmit() at
 internet protocol level really send out the TCP_SYN_SEND packet */
2329         sk->prot->queue_xmit(sk, dev, buff, 0);
2334         release_sock(sk);
2335         return(0);
2336 }
```

IP protocol is resposible to send out the SYN packet to the target address. On the target machine, when the packet arrives, the driver throws an interupt. A series of funtions on network and IP packet level will be invoked. Finally tcp protocol hanlder tcp_rcv will be called.

```
2300 int tcp_rcv(struct sk_buff *skb, struct device *dev, struct options *opt,
2301         __u32 daddr, unsigned short len,
2302         __u32 saddr, int redo, struct inet_protocol * protocol)
2303 {
2304         struct tcphdr *th;
2305         struct sock *sk;
2306         __u32 seq;
2307         int was_ack;
```

```
2391        if (sk->zapped || sk->state==TCP_CLOSE) {
2392                goto no_tcp_socket;  <-- connection fails!
2393        }
2425        if(sk->state!=TCP_ESTABLISHED)          /* Skip this lot for normal flow */
2426        {
2427
2432                if(sk->state==TCP_LISTEN)
2433                {
2507                        tcp_conn_request(sk, skb, daddr, saddr,
opt, dev, seq); <--- change to state TCP_SYN_RECV, and send out SYN+ACK
2520                        return 0;
2521                }
2823 no_tcp_socket:
2824        /*
2825         * No such TCB. If th->rst is 0 send a reset (checked in tcp_send_reset)
2826         */
2827        tcp_send_reset(daddr, saddr, th, &tcp_prot, opt,dev,0,255);
2829 discard_it:
2833        skb->sk = NULL;
2834        kfree_skb(skb, FREE_READ);
2835        return 0;
2836 }
```

```
526 /*
527  *      This routine handles a connection request.
528  *      It should make sure we haven't already responded.
529  *      Because of the way BSD works, we have to send a syn/ack now.
532  */
533
534 static void tcp_conn_request(struct sock *sk, struct sk_buff *skb,
535                 u32 daddr, u32 saddr, struct options *opt, struct device *dev, u32 seq)
536 {
537         struct sock *newsk;
538         struct tcphdr *th;
539         struct rtable *rt;
543
544         th = skb->h.th;

639         /*
640          * We need to build a new sock struct.
641          * It is sort of bad to have a socket without an inode attached
642          * to it, but the wake_up's will just wake up the listening socket,
643          * and if the listening socket is destroyed before this is taken
644          * off of the queue, this will take care of it.
645          */
```

```
646
647         newsk = (struct sock *) kmalloc(sizeof(struct sock), GFP_ATOMIC);
656         memcpy(newsk, sk, sizeof(*newsk));

            ... .../* do a lot of initialization */
729         newsk->state = TCP_SYN_RECV;
745         newsk->dummy_th.source = skb->h.th->dest;
746         newsk->dummy_th.dest = skb->h.th->source;
757         /*
758          *      Swap these two, they are from our point of view.
759          */
761         newsk->daddr = saddr;
762         newsk->saddr = daddr;
763         newsk->rcv_saddr = daddr;
772         {
773                 tcp_v4_hash(newsk);
774                 add_to_prot_sklist(newsk);
775         }
779         newsk->socket = NULL;
873         tcp_send_synack(newsk, sk, skb, 0); <-- send back SYN+ACK to source address
869 }
```

```
960 void tcp_send_synack(struct sock * newsk, struct sock * sk, struct sk_buff * skb, int destro
961 {
962         struct tcphdr *t1;
963         unsigned char *ptr;
964         struct sk_buff * buff;
965         struct device *ndev=newsk->bound_device;
966         int tmp;
967
968         buff = sock_wmalloc(newsk, MAX_SYN_SIZE, 1, GFP_ATOMIC);

977
978         buff->sk = newsk;
985         tmp = sk->prot->build_header(buff, newsk->saddr, newsk->daddr, &ndev,
986                                      IPPROTO_TCP, newsk->opt, MAX_SYN_SIZE,sk->ip_tos,sk->ip_ttl,&
1004         t1 =(struct tcphdr *)skb_put(buff,sizeof(struct tcphdr));
1005
1006         memcpy(t1, skb->h.th, sizeof(*t1));

1009         /*
1010          *      Swap the send and the receive.
1011          */
1012         t1->dest = skb->h.th->source;
```

```
1013          t1->source = newsk->dummy_th.source;
1014          t1->seq = ntohl(buff->seq);
1015          newsk->sent_seq = newsk->write_seq;
              ... ... /* set other fields of newsk */
              /* send out SYN+ACK packet */
1050                  newsk->prot->queue_xmit(newsk, ndev, buff, 0);
1078 }
```

Now it is client's turn to process SYN+ACK from server. The same as server did while it receives SYN packet, eventually tcp_rcv() is invoked.

```
2300 int tcp_rcv(struct sk_buff *skb, struct device *dev, struct options *opt,
2301          __u32 daddr, unsigned short len,
2302          __u32 saddr, int redo, struct inet_protocol * protocol)
2303 {
           ... ...
2425          if(sk->state!=TCP_ESTABLISHED)          /* Skip this lot for normal flow */
2426          {
                  ....
2544                  if(sk->state==TCP_SYN_SENT)
2545                  {
2546                          /* Crossed SYN or previous junk segment */
```

```
2547                    if(th->ack)
2548                    {

2568                            if(!th->syn)
2569                            {
                                        ... ...
2576                                    return 0;
2577                            }
2578
2579                            /* process the ACK, get the SYN packet out
2580                             * of the send queue, do other initial
2581                             * processing stuff. [We know it's good, and
2582                             * we know it's the SYN,ACK we want.]
2583                             */
2584                            tcp_ack(sk,th,skb->ack_seq,len);
                                <-- change state into TCP_ESTABLISHED
2601                            tcp_send_ack(sk);  <-- send out ACK
2602                            tcp_set_state(sk, TCP_ESTABLISHED);
2625                    }
2656            }
```

Now it is the server who should get this ACK and enter state TCP_ESTABLISHED:

```
2300 int tcp_rcv(struct sk_buff *skb, struct device *dev, struct options *opt,
2301         __u32 daddr, unsigned short len,
2302         __u32 saddr, int redo, struct inet_protocol * protocol)
2303 {
             ......
2425         if(sk->state!=TCP_ESTABLISHED)          /* Skip this lot for normal flow */
2426         {
2528                 if (sk->state == TCP_SYN_RECV)
2529                 {
2530                         if(th->syn && skb->seq+1 == sk->acked_seq)
2531                         {
2532                                 kfree_skb(skb, FREE_READ);
2533                                 return 0;
2534                         }
2535                         goto rfc_step4;
2536                 }
2714 rfc_step4:
2757         if(!th->ack)
2758         {
2759                 kfree_skb(skb, FREE_WRITE);
```

```
2760                    return 0;
2761            }
2762
2763            if(!tcp_ack(sk,th,skb->ack_seq,len))  <-- change state to TCP_ESTABLISHED
2764            {
2774                    return 0;
2775            }
```

Till now the connection has been built up, at least on the INET socket, no matter whether 'accept' has been issued from application on server (though a 'listen' is required).

Once the server has received the incoming request it either accepts or rejects it. If the incoming request is to be accepted, the server must create a new socket to accept it on. Once a socket has been used for listening for incoming connection requests, it can't be used to support a connection.

Let us see how an BCD socket is hooked on the exsiting INET socket via 'accept'

```
758 /*
759  *      For accept, we attempt to create a new socket, set up the link
760  *      with the client, wake up the client, then return the new
```

```
761  *      connected fd. We collect the address of the connector in kernel
762  *      space and move it to user at the very end.
764  */
765
766 asmlinkage int sys_accept(int fd, struct sockaddr *upeer_sockaddr, int *upeer_addrlen)
767 {
768         struct socket *sock, *newsock;
769         int i;
770         char address[MAX_SOCK_ADDR];
771         int len;
772
                /* Go from a file number to its socket slot */
773         if (!(sock = sockfd_lookup(fd, &i)))
774                 return i;
781         if (!(newsock = sock_alloc()))
782         {
                  ... ...
787         }
788         newsock->type = sock->type;
789         newsock->ops = sock->ops;
797         i = newsock->ops->accept(sock, newsock, sock->file->f_flags);
805         if ((fd = get_fd(SOCK_INODE(newsock))) < 0)
```

```
806         {
                    ... ...
810         }
811         newsock->file=current->files->fd[fd];
812
813         if (upeer_sockaddr)
814         {
815                 newsock->ops->getname(newsock, (struct sockaddr *)address, &len, 1);
816                 move_addr_to_user(address,len, upeer_sockaddr, upeer_addrlen);
817         }
818         sockfd_put(sock);
819         return(fd);
820 }
```

Now from BCD socket call 'accept' specific to INET address family.

```
744 /*
745  *      Accept a pending connection. The TCP layer now gives BSD semantics.
746  */
748 static int inet_accept(struct socket *sock, struct socket *newsock, int flags)
749 {
750         struct sock *sk1, *sk2;
```

```
751         int err;
753         sk1 = (struct sock *) sock->data;
774         if (sk1->pair != NULL) {
775                 sk2 = sk1->pair;
776                 sk1->pair = NULL;
777         } else {
778                 sk2 = sk1->prot->accept(sk1,flags);
779                 if (sk2 == NULL)
780                         return sock_error(sk1);
781         }
782         newsock->data = (void *)sk2;
784         sk2->socket = newsock;
789         cli(); /* avoid the race. */
790         while(sk2->state == TCP_SYN_RECV) {
791                 interruptible_sleep_on(sk2->sleep);
                  ... ...
799                 }
800         }
801         sti();
815         newsock->state = SS_CONNECTED;
816         return(0);
817 }
```

```
2099 static struct sock *tcp_accept(struct sock *sk, int flags)
2100 {
2101         int error;
2102         struct sk_buff *skb;
2103         struct sock *newsk = NULL;
2104
2105   /*
2106    * We need to make sure that this socket is listening,
2107    * and that it has something pending.
2108    */
2109
2110         error = EINVAL;
2111         if (sk->state != TCP_LISTEN)
2112                 goto no_listen;
2114         lock_sock(sk);start_bh_atomic();
2116         skb = tcp_find_established(sk);
2117         if (skb) {
2118 got_new_connect:
2119                 __skb_unlink(skb, &sk->receive_queue);
2120                 newsk = skb->sk;
2121                 kfree_skb(skb, FREE_READ);
2122                 sk->ack_backlog--;
```

```
2123                    error = 0;
2124 out:
2125                    end_bh_atomic();
2126                    release_sock(sk);
2127 no_listen:
2128                    sk->err = error;
2129                    return newsk;
2130            }
2131
2132            error = EAGAIN;
2133            if (flags & O_NONBLOCK)
2134                    goto out;
2135            skb = wait_for_connect(sk);<--placed on the 'sleep' waiting queue
                                               until a eligible 'skb' found
2136            if (skb)
2137                    goto got_new_connect;
2138            error = ERESTARTSYS;
2139            goto out;
2140 }
```

```
659 /*
660  *      Find someone to 'accept'. Must be called with
661  *      the socket locked or with interrupts disabled
662  */
663
664 static struct sk_buff *tcp_find_established(struct sock *s)
665 {
666         struct sk_buff *p=skb_peek(&s->receive_queue);
667         if(p==NULL)
668                 return NULL;
669         do
670         {
671                 if(p->sk->state == TCP_ESTABLISHED || p->sk->state >= TCP_FIN_WAIT1)
672                         return p;
673                 p=p->next;
674         }
675         while(p!=(struct sk_buff *)&s->receive_queue);
676         return NULL;
677 }
```

```
 57 enum {
```

```
58   TCP_ESTABLISHED = 1,
59   TCP_SYN_SENT,
60   TCP_SYN_RECV,
61   TCP_FIN_WAIT1,
62   TCP_FIN_WAIT2,
63   TCP_TIME_WAIT,
64   TCP_CLOSE,
65   TCP_CLOSE_WAIT,
66   TCP_LAST_ACK,
67   TCP_LISTEN,
68   TCP_CLOSING   /* now a valid state */
69 };
```

Till now, a newly generated socket has been hooked onto a received socket buffer. The whole connection is avaible between 2 BCD socket, and the send/recv data stream can flow along the connection.

How is data stream transmitted along the connection?

With the connection established both ends are free to send and receive data. Let's see how a read is implemented on a TCP established connection.

Linux/net/socket.c

```
354 /*
355  *      Read data from a socket. ubuf is a user mode pointer. We make sure the user
356  *      area ubuf...ubuf+size-1 is writable before asking the protocol.
357  */
358
359 static int sock_read(struct inode *inode, struct file *file, char *ubuf, int size)
360 {
361         struct socket *sock;
362         int err;
363         struct iovec iov;
364         struct msghdr msg;
365
366         sock = socki_lookup(inode); <--   inode ==> socket
374         if ((err=verify_area(VERIFY_WRITE,ubuf,size))<0)
375                 return err;
```

```
376         msg.msg_name=NULL;
377         msg.msg_iov=&iov;
378         msg.msg_iovlen=1;
380         iov.iov_base=ubuf;
381         iov.iov_len=size;
383         return(sock->ops->recvmsg(sock, &msg, size,(file->f_flags & O_NONBLOCK), 0,&msg.msg_
384 }
```

Here msghdr is used to tranfer data.

```
 25 struct msghdr
 26 {
 27         void    *       msg_name;       /* Socket name                  */
 28         int             msg_namelen;    /* Length of name               */
 29         struct iovec *  msg_iov;        /* Data blocks                  */
 30         int             msg_iovlen;     /* Number of blocks             */
 34 };
```

```
inet_recvmsg(struct socket *sock, struct msghdr *ubuf, int size, int noblock,
int flags, int *addr_len )
```

is called, then tcp_recvmsg() is invoked to receive data via INET socket.

```
1584 /*
1585  *      This routine copies from a sock struct into the user buffer.
1586  */
1588 static int tcp_recvmsg(struct sock *sk, struct msghdr *msg,
1589         int len, int nonblock, int flags, int *addr_len)
1590 {
1591         struct wait_queue wait = { current, NULL };

1618         seq = &sk->copied_seq;
1622         add_wait_queue(sk->sleep, &wait);
1623         lock_sock(sk);
1624         while (len > 0)
1625         {
1626                 struct sk_buff * skb;
1627                 u32 offset;

1653                 current->state = TASK_INTERRUPTIBLE;
1655                 skb = sk->receive_queue.next;
1656                 while (skb != (struct sk_buff *)&sk->receive_queue)
1657                 {
1658                         if (before(*seq, skb->seq))
1659                                 break;
```

```
1660                    offset = *seq - skb->seq;
1661                    if (skb->h.th->syn)
1662                            offset--;
1663                    if (offset < skb->len)
1664                            goto found_ok_skb;
1669                    skb = skb->next;
1670            }
                ... ...
1704            cleanup_rbuf(sk);
1705            release_sock(sk);
1706            sk->socket->flags |= SO_WAITDATA;
1707            schedule();
1708            sk->socket->flags &= ~SO_WAITDATA;
1709            lock_sock(sk);
1710            continue;
1712    found_ok_skb:
1765            memcpy_toiovec(msg->msg_iov,((unsigned char *)skb->h.th) +
1766                    skb->h.th->doff*4 + offset, used);
                ... ...
1809    }
1810
1811    if(copied>0 && msg->msg_name)
```

```
1812         {
1813                 struct sockaddr_in *sin=(struct sockaddr_in *)msg->msg_name;
1814                 sin->sin_family=AF_INET;
1815                 sin->sin_addr.s_addr=sk->daddr;
1816                 sin->sin_port=sk->dummy_th.dest;
1817         }
1818         if(addr_len)
1819                 *addr_len=sizeof(struct sockaddr_in);
1820
1821         remove_wait_queue(sk->sleep, &wait);
1822         current->state = TASK_RUNNING;
1823
1824         /* Clean up data we have read: This will do ACK frames */
1825         cleanup_rbuf(sk);
1826         release_sock(sk);
1827         return copied;
1828 }
```

What tcp_recvmsg() found here is placed by tcp_rcv(), let's see it:

```
2300 int tcp_rcv(struct sk_buff *skb, struct device *dev, struct options *opt,
2301         __u32 daddr, unsigned short len,
2302         __u32 saddr, int redo, struct inet_protocol * protocol)
2303 {
2304         struct tcphdr *th;
2305         struct sock *sk;
2306         __u32 seq;
2307         int was_ack;

2318         th = skb->h.th;
2319         was_ack = th->ack; /* Remember for later when we've freed the skb */
2320         sk = skb->sk;

2714 rfc_step4:              /* I'll clean this up later */
2715
2716         /*
2717          *      We are now in normal data flow (see the step list in the RFC)
2718          *      Note most of these are inline now. I'll inline the lot when
2719          *      I have time to test it hard and look at what gcc outputs
2720          */
```

```
2721          /* checks to see if the tcp header is actually acceptable. */
2722          if (!tcp_sequence(sk, skb->seq, skb->end_seq-th->syn))
2723          {
2724                  bad_tcp_sequence(sk, th, skb->end_seq-th->syn, dev);
2725                  kfree_skb(skb, FREE_READ);
2726                  return 0;
2727          }
2777 rfc_step6:
2789
2790          /*
2791           *      Process the encapsulated data
2792           */
2793
2794          if(tcp_data(skb,sk, saddr, len))
2795                  kfree_skb(skb, FREE_READ);
2796
2821          return 0;
2836 }
```

We need to move the skb into the receive_queue of sk, so that the application can fetch it through reading socket.

```
2034 /*
2035  *      This routine handles the data.  If there is room in the buffer,
2036  *      it will be have already been moved into it.  If there is no
2037  *      room, then we will just have to discard the packet.
2038  */
2039
2040 static int tcp_data(struct sk_buff *skb, struct sock *sk,
2041          unsigned long saddr, unsigned int len)
2042 {
2043         struct tcphdr *th;
2044         u32 new_seq, shut_seq;
2045
2046         th = skb->h.th;
2047         skb_pull(skb,th->doff*4);
2048         skb_trim(skb,len-(th->doff*4));
2049
2050         /*
2051          *      The bytes in the receive read/assembly queue has increased. Needed for the
2052          *      low memory discard algorithm
2053          */
2054
2055         sk->bytes_rcv += skb->len;
```

```
2132         tcp_queue(skb, sk, th);
2134         return(0);
2135 }

1922 static void tcp_queue(struct sk_buff * skb, struct sock * sk, struct tcphdr *th)
1923 {
1924         u32 ack_seq;
1925
1926         tcp_insert_skb(skb, &sk->receive_queue);
             ... ...
1988                 /*
1989                  *      Tell the user we have some more data.
1990                  */
1991
1992                 if (!sk->dead)
1993                         sk->data_ready(sk,0);
2031 }
```

On calling `sk->data_ready()`, callback function will wake up tasks waiting on this socket.

```
 68 static void smb_data_callback(struct sock *sk, int len)
 69 {
 70         struct socket *sock = sk->socket;
 71
 72         if (!sk->dead)
 73         {
 74                 unsigned char peek_buf[4];
 75                 int result;
 76                 unsigned short fs;
 77
                    ... ...

103                 if (result != -EAGAIN)
104                 {
105                         wake_up_interruptible(sk->sleep);
106                 }
107         }
108 }
```

TCP Timers

TCP must manage seven different timers for each connection. Assume a very simple connection in which one process (Sender) only sends data and the other process(Receiver) only receives data. The following timers are maintained by Sender only.

- Connection-Established: The connection-establishment timer starts when a SYN is sent to establish a new connection. If the reponse is not received within 75 seconds, the connection establishment is aborted.
- Retransmission: The retransmission-timer is set when TCP sends data. If the data is not acknowledged by the other end when this expires, TCP retransmits the data. The value of this timer is calculated dynamically but must be tetween 1 and 64 seconds.
- Persist: The persist timer is used when the Receiver has throttled the Sender. Throttle packets(called window advertisements) are not acknowledged, so a lost window advertisement could effectively stop all communication. When this timer goes off, the Sender queries the Receiver.
- FIN_WAIT2: The FIN_WAIT2 timer is used when the sender executes a close. If the Receiver crashes, it will never acknowledge the close. When this time goes off, the

Sender assumes the Receiver crashed and closes the connection.

- 2MSL: The 2MSL(2 times maximum segment lifetime) timer is set when the Receiver enters the TIME_WAIT state. At this point, both sides have closed their sockets.

A single timer is used by the Receiver only:

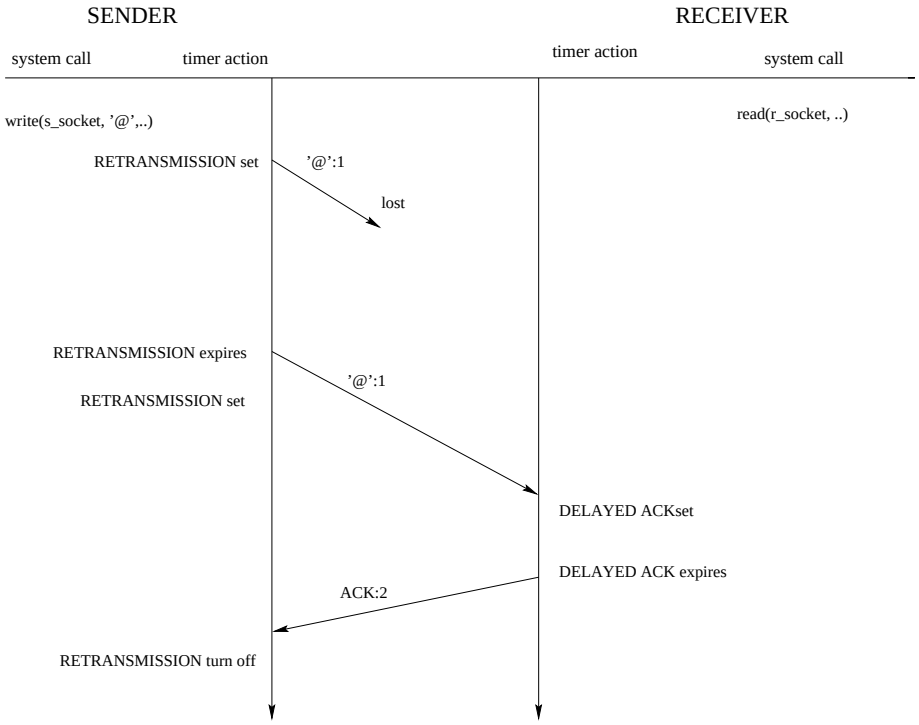
- Delayed ACK: The delayed ACK timer is set when TCP receives data that must be acknowledged, but need not be acknowledged immediately. Needed because TCP uses a sliding window protocol.

Both the Sender and the Receiver implement:

- Keepalive: The keepalive timer is optional and can be turned on by the user. Since TCP is connection oriented, even idle connections need to be maintained. If the connection is idle for 2 hours, a special segemnt is sent to the other end, forcing it to respond. Both ends can thus determine if their peer have crashed/rebooted.

A simple example

Assume a window and segment size of one byte. The Sender wants to transmit one byte of data and stop.



To write, sock_write() is called. Then inet_sendmsg(), tcp_sendmsg, tcp_do_sendmsg, at last tcp_send_skb() is called one by one:

```
124 void tcp_send_skb(struct sock *sk, struct sk_buff *skb)
125 {
126         int size;
127         struct tcphdr * th = skb->h.th;

209         {
210                 /*
211                  *      This is going straight out
212                  */
213                 clear_delayed_acks(sk);
214                 th->ack_seq = htonl(sk->acked_seq);
215                 th->window = htons(tcp_select_window(sk));
216
217                 tcp_send_check(th, sk->saddr, sk->daddr, size, skb);
218
219                 sk->sent_seq = sk->write_seq;

227                 sk->prot->queue_xmit(sk, skb->dev, skb, 0);
228
```

```
229                 /*
230                  *      Set for next retransmit based on expected ACK time
231                  *      of the first packet in the resend queue.
232                  *      This is no longer a window behind.
233                  */
234
235                 tcp_reset_xmit_timer(sk, TIME_WRITE, sk->rto);
236         }
237 }
```

Assume the Retransmit Timer has gone off:

```
 82 static void tcp_retransmit_time(struct sock *sk, int all)
 83 {
 84         /*
 85          * record how many times we've timed out.
 86          * This determines when we should quite trying.
 87          * This needs to be counted here, because we should not be
 88          * counting one per packet we send, but rather one per round
 89          * trip timeout.
 90          */
 91         sk->retransmits++;
```

```
 92
 93         tcp_do_retransmit(sk, all);
 94
 95         /*
 96          * Increase the timeout each time we retransmit.  Note that
 97          * we do not increase the rtt estimate.  rto is initialized
 98          * from rtt, but increases here.  Jacobson (SIGCOMM 88) suggests
 99          * that doubling rto each time is the least we can get away with.
100          * In KA9Q, Karn uses this for the first few times, and then
101          * goes to quadratic.  netBSD doubles, but only goes up to *64,
102          * and clamps at 1 to 64 sec afterwards.  Note that 120 sec is
103          * defined in the protocol as the maximum possible RTT.  I guess
104          * we'll have to use something other than TCP to talk to the
105          * University of Mars.
106          *
107          * PAWS allows us longer timeouts and large windows, so once
108          * implemented ftp to mars will work nicely. We will have to fix
109          * the 120 second clamps though!
110          */
111
112         sk->backoff++;
113         sk->rto = min(sk->rto << 1, 120*HZ);
```

```
114
115         /* be paranoid about the data structure... */
116         if (sk->send_head)
117                 tcp_reset_xmit_timer(sk, TIME_WRITE, sk->rto);
118         else
119                 /* This should never happen! */
120                 printk(KERN_ERR "send_head NULL in tcp_retransmit_time\n");
121 }
122
```

Receiver just needs to read the packet, set the ACK timer and hang out until it goes off. The following routine is to do this.

```
void tcp_send_delayed_ack(struct sock * sk, int max_timeout, unsigned long timeout)
```

References:

1. Linux kernel Internels, second edition, M. Beck, H. Bohme, etc.
2. TCP/IP Illustrated, Vol2 – The implementation, G.R.Wright W.R. Stevens
3. Bob Matthews's TCP presentation
4. Eric Koskinen's IP presentation