UDP and the *sendto* Socket API

There are several mechanisms through which an application may use the socket API to initiate the transmission of a UDP datagram. We begin with *sendto()* because it does allow the user to pass an address structure, but it does not support scatter/gather operations via the *iovec* mechanism that is supported by *sendmsg()*. The parameters passed by the application to *sendto()* include:

fd	The file handle associated with the socket.
buff	A user-space pointer to the user data to be transmitted
len	The number of bytes of user data to be transmitted
addr	A user-space pointer to the *struct sockaddr_in* containing the destination address.
addr_len	The number of bytes of address data.
flags	Usually 0 but enumerated below (quoted from *man sendto*)

MSG_OOB: Sends out-of-band data on sockets that support this notion (e.g. SOCK_STREAM); the underlying protocol must also support out-of-band data. UDP does not.

MSG_DONTROUTE: Don't use a gateway to send out the packet, only send to hosts on directly connected networks. This is usually used only by diagnostic or routing programs. This is only defined for protocol families that route; packet sockets don't.

MSG_DONTWAIT: Enables non-blocking operation; if the operation would block, EAGAIN is returned (this can also be enabled using the O_NONBLOCK with the F_SETFL fcntl(2)).

MSG_NOSIGNAL: Requests not to send SIGPIPE on errors on stream oriented sockets when the other end breaks the connection. The EPIPE error is still returned.

MSG_MORE (Since Linux 2.4.4)

The caller has more data to send. This flag is used with TCP sockets to obtain the same effect as the TCP_CORK socket option (see tcp(7)), with the difference that this flag can be set on a per-call basis.

Since Linux 2.6, this flag is also supported for UDP sockets, and informs the kernel to package all of the data sent in calls with this flag set into a single datagram which is only transmitted when a call is performed that does not specify this flag. (See also the UDP_CORK socket option described in udp(7).)

MSG_CONFIRM (Linux 2.3+ only) Tell the link layer that forward process happened: you got a successful reply from the other side. If the link layer doesn't get this it'll regularly reprobe the neighbour (e.g. via a unicast ARP). Only valid on SOCK_DGRAM and SOCK_RAW sockets and currently only implemented for IPv4 and IPv6. See arp(7) for deatils

The *struct msghdr*

At this point it is useful to introduce some data structures that are used internally in the management of system calls that use the *sendto()* API. The *struct msghdr* and the *struct iovec* defined in *include/linux/socket.h* are used in the assembly and management of parameter information for *all* socket calls..

```
struct msghdr {
   void    *msg_name;          /* Socket name          */
   int      msg_namelen;       /* Length of name       */
   struct iovec *  msg_iov;    /* Data blocks          */
   __kernel_size_t msg_iovlen; /* Number of blocks     */

   void    *msg_control;  /* Per protocol magic
                   (eg BSD file descriptor passing) */
   __kernel_size_t msg_controllen; /* Length of cmsg list */
  unsigned        msg_flags;
};
```

msg_name — A pointer to the *struct sockaddr* passed by the application. For TCP/IP sockets this will always be a *struct sockaddr_in*.

msg_namelen — The length of the name structure passed in. For TCP/IP sockets this should be *sizeof(struct sockaddr_in)*.

msg_iov — A pointer to the IO vector.

msg_iovlen — The number of elements in the IO vector which is the number of disjoint fragments of memory comprising the message. For the *sendto()* API this value is necessarily 1.

msg_control — A pointer to *struct cmsghdr*. The use of control messages is related to the ability to pass *fds* through sockets.

msg_controllen — The size of the associated *cmsg* data.

msg_flags — These flags were documented on the first page of this section.

The *struct iovec*

The *iovec* mechanism is designed to support a general scatter/gather facility, but this is not supported by the *sendto* API. With the *sendmsg()* API the application program must provide both a *msghdr* and an *iovec,* but the *sys_sendto()* function constructs these structures when the *sendto()* API is used.

The *struct iovec* is defined in *include/linux/uio.h*. A single *iovec* element holds the user-space address and size of each block of user data. The *sendto()* API requires only a single element.

```
struct iovec
{
  void __user *iov_base;  /* BSD uses caddr_t (1003.1g
                             requires void *) */
  __kernel_size_t iov_len; /* Must be size_t (1003.1g) */
};
```

iov_base	A pointer to the *user space address* of the start of a message fragment.
iov_len	The length of the fragment.

The *sys_sendto()* front end

The *sys_sendto()* kernel function receives control from *sys_socketcall()* when the user-level *sendto()* function is invoked. This function is defined in *net/socket.c*. Its parameters are precisely those passed by the application program. The principle missions of this function are to *copy required parameters to kernel space* and *consolidate the parameter information in the iov and msg structures* that are allocated on the stack as shown below.

```
asmlinkage long sys_sendto(int fd, void __user * buff,
               size_t len, unsigned flags,
               struct sockaddr __user *addr, int addr_len)
{
        struct socket *sock;
        char address[MAX_SOCK_ADDR];
        int err;
        struct msghdr msg;
        struct iovec iov;
        int fput_needed;
        struct file *sock_file;
```

In standard fashion, the function commences by attempting to recover a pointer to the *struct socket* from the *fd* that was passed in. Failure here is fatal.

```

        sock_file = fget_light(fd, &fput_needed);
        if (!sock_file)
                return -EBADF;

        sock = sock_from_file(sock_file, &err);
        if (!sock)
                goto out_put;
```

Constructing the *msghdr* and the *iovec*

Here the *iov* and *msg* structures are filled in by *sys_sendo()* using the parameters provided by the user. Since the *sendto* API doesn't support scatter/gather, there will always be only a single element in the *iov*. The *control message* is a somewhat obscure facility by which an open *fd* may be passed from one process to another. This is not supported by the *sendto()* API and the control message pointer is set to NULL here.

```
        iov.iov_base=buff;
        iov.iov_len=len;

        msg.msg_name=NULL;
        msg.msg_iov=&iov;
        msg.msg_iovlen=1;
        msg.msg_control=NULL;
        msg.msg_controllen=0;
        msg.msg_namelen=0;
```

Copying the *struct sockaddr_in* to kernel space

Here *addr* should point to the *struct sockaddr_in* in *user space.* If it is NULL then no address structure was provided. The structure is copied to the local array *address* which is on this function's stack. The *msg_name* element of the *msg* structure is then set to point to the *kernel resident copy* of the *struct sockaddr_in.*

```
    if (addr) {
        err = move_addr_to_kernel(addr, addr_len, address);
        if (err < 0)
            goto out_put;
        msg.msg_name=address;
        msg.msg_namelen=addr_len;
    }
```

If the socket already carries the O_NONBLOCK attribute, the MSG_DONTWAIT bit is added to the *flags* passed in by the user.

```
    if (sock->file->f_flags & O_NONBLOCK)
        flags |= MSG_DONTWAIT;
    msg.msg_flags = flags;
```

With all the parameter data having been collected, *sock_sendmsg()* is invoked to do the work. Note that the *len* parameter appears to be redundant in this context since *len* has also been copied into the *iov*.

```
    err = sock_sendmsg(sock, &msg, len);

out_put:
    fput_light(sock_file, fput_needed);
    return err;
}

```

Asynchronous I/O

- One mechanims allows an application to request an I/O operation to be initiated.
- Another mechanism allows the application to wait for completion.
- Together with double buffering they make it possible to overlap I/O and processing
- This reduces elapsed time required and thus potentially increases throughput.

http://sourceforge.net/docman/display_doc.php?docid=12548&group_id=8875

1. Motivation

Asynchronous i/o overlaps application processing with i/o operations for improved utilization of CPU and devices, and improved application performance, in a dynamic/adaptive manner, especially under high loads involving large numbers of i/o operations.

1.1 Where aio could be used:

Application performance and scalable connection management:

(a) Communications aio:
Web Servers, Proxy servers, LDAP servers, X-server

(b) Disk/File aio:
Databases, I/O intensive applications

(c) Combination
Streaming content servers (video/audio/web/ftp)
(transfering/serving data/files directly between disk and network)

If ki_retry returns -EIOCBQUEUED it has made a promise that aio_complete() will be called on the kiocb pointer in the future. The AIO core will not ask the method again -- ki_retry must ensure forward progress. aio_complete() must be called once and only once in the future, multiple calls may result in undefined behaviour.

If ki_retry returns -EIOCBRETRY it has made a promise that kick_iocb() will be called on the kiocb pointer in the future. This may happen through generic helpers that associate kiocb->ki_wait with a wait queue head that ki_retry uses via current->io_wait. It can also happen with custom tracking and manual calls to kick_iocb(), though that is discouraged. In either case, kick_iocb() must be called once and only once. ki_retry must ensure forward progress, the AIO core will wait indefinitely for kick_iocb() to be called.

The *kiocb*

The *kiocb* is the kernel level structure used to track a single AIO request. It is generic and applies to both block device I/O and socket I/O. The *private* field is used to link the *kiocb* to the *sock_iocb.*

```
struct kiocb {
   struct list_head  ki_run_list;
   long              ki_flags;
   int               ki_users;
   unsigned          ki_key;         /* id of this request */

   struct file       *ki_filp;
   struct kioctx     *ki_ctx;   /* may be NULL for sync ops */
   int     (*ki_cancel)(struct kiocb *, struct io_event *);
   ssize_t           (*ki_retry)(struct kiocb *);
   void              (*ki_dtor)(struct kiocb *);

    union {
       void __user         *user;
       struct task_struct  *tsk;
   } ki_obj;

   __u64      ki_user_data;   /* user's data for completion */
   wait_queue_t        ki_wait;
   loff_t              ki_pos;

   void                *private;
/* State that we remember to be able to restart/retry  */
   unsigned short      ki_opcode;
   size_t             ki_nbytes;/* copy of iocb->aio_nbytes */
   char          __user *ki_buf; /* remaining iocb->aio_buf */
   size_t              ki_left;        /* remaining bytes */
   long                ki_retried;     /* just for testing */
   long                ki_kicked;      /* just for testing */
   long                ki_queued;      /* just for testing */

   struct list_head    ki_list;    /* the aio core uses this
                                    * for cancellation */
};
```

The *sock_iocb*

This structure serves as a “container” for the parameters associated with a socket I/O request.

```
struct sock_iocb {
   struct list_head        list;

   int                     flags;
   int                     size;
   struct socket           *sock;
   struct sock             *sk;
   struct scm_cookie       *scm;
   struct msghdr           *msg, async_msg;
   struct iovec            async_iov;
   struct kiocb            *kiocb;
};
```

The *sock_sendmsg()* function

The *sock_sendmsg()* function defined in *net/socket.c* allocates and initializes *kiocb* and *sock_iocb* as local stack resident variables. These variables are not used at all on the UDP path. So presumably asynchronous I/O *is not in play and the wait_on_sync_kiocb()* never occurs for a UDP request.

```
int sock_sendmsg(struct socket *sock,
          struct msghdr *msg, size_t size)
{
   struct kiocb iocb;
   struct sock_iocb siocb;
   int ret;

   init_sync_kiocb(&iocb, NULL);
   iocb.private = &siocb;
   ret = __sock_sendmsg(&iocb, sock, msg, size);
   if (-EIOCBQUEUED == ret)
        ret = wait_on_sync_kiocb(&iocb);
   return ret;
}
```

```
#define init_sync_kiocb(x, filp)                       \
   do {                                               \
           struct task_struct *tsk = current;         \
           (x)->ki_flags = 0;                         \
           (x)->ki_users = 1;                         \
           (x)->ki_key = KIOCB_SYNC_KEY;              \
           (x)->ki_filp = (filp);                     \
           (x)->ki_ctx = NULL;                        \
           (x)->ki_cancel = NULL;                     \
           (x)->ki_retry = NULL;                      \
           (x)->ki_dtor = NULL;                       \
           (x)->ki_obj.tsk = tsk;                     \
           (x)->ki_user_data = 0;                     \
           init_wait((&(x)->ki_wait));                \
   } while (0)

```

The __*sock_sendmsg()* function

This function packages the socket call related parameters into the *sock_iocb* and then forwards them on to the AF_layer handler specified in the *proto_ops* structure linked to the *socket.*

```
static inline int __sock_sendmsg(struct kiocb *iocb,
                        struct socket *sock,
                        struct msghdr *msg, size_t size)
{
    struct sock_iocb *si = kiocb_to_siocb(iocb);
    int err;

    si->sock = sock;
    si->scm = NULL;
    si->msg = msg;
    si->size = size;

```

We have seen these calls before. They access *hooks* provided by the Security Enhanced Linux (SEL) facility. At present they all just seem to return 0!

```
    err = security_socket_sendmsg(sock, msg, size);
    if (err)
        return err;

```

This call maps to *inet_sendmsg()*

```
    return sock->ops->sendmsg(iocb, sock, msg, size);
}
```

The security system.

The “security” family of calls are part of the *security enhanced linux (SEL)* facility. It is a large framework with hooks in many places but it appears that at present it doesn't really do anything in the socket system. The *security_socket_sendmsg()* function just invokes the function pointed to by *socket_sendmsg()* element of the structure *security_operations* pointed to by *security_ops.*

As we shall see *security_ops->socket_sendmsg()* is bound to *cap_socket_sendmsg()* which does nothing.

```
int security_socket_sendmsg(struct socket *sock,
                            struct msghdr *msg, int size)
{
    return security_ops->socket_sendmsg(sock, msg, size);
}
```

The *security_ops* pointer is initially set to NULL.

```
struct security_operations *security_ops; /* Initialized to
                                               NULL */
```

An instance of the structure, the *default_security_ops* are also initially NULL.

```
struct security_operations default_security_ops = {
       .name   = "default",
};
```

Security initialization

Initialization takes place at boot time. The *security_fixup_ops()* function fills in the *default_security_ops* table and then the *security_ops* pointer is set to point to that table.

```
/**
 * security_init - initializes the security framework
 *
 * This should be called early in the kernel initialization
   sequence.
*/
int __init security_init(void)
{
   printk(KERN_INFO "Security Framework initialized\n");

   security_fixup_ops(&default_security_ops);
   security_ops = &default_security_ops;
   do_security_initcalls();

       return 0;
}
```

The *set_to_cap_if_null* macro sets the security function for operation *x* to point to the actual function *cap_x*. Thus the security function for *socket_sendmsg* is *cap_socket_sendmsg*.

```
#define set_to_cap_if_null(ops,function)                        \
   do {                                                          \
       if (!ops->function) {                                     \
           ops->function = cap_##function;                       \
           pr_debug("Had to override the " #function             \
                  " security operation with the default.\n");\
       }                                                         \
   } while (0)
```

The *security_fixup_ops* function

This function fills in the *default_security_ops* table one entry at a time.

```
void security_fixup_ops(struct security_operations *ops)
{
        set_to_cap_if_null(ops, ptrace_may_access);
        set_to_cap_if_null(ops, ptrace_traceme);
        set_to_cap_if_null(ops, capget);
 :
 :
        set_to_cap_if_null(ops, socket_connect);
        set_to_cap_if_null(ops, socket_listen);
        set_to_cap_if_null(ops, socket_accept);
        set_to_cap_if_null(ops, socket_post_accept);
        set_to_cap_if_null(ops, socket_sendmsg);
        set_to_cap_if_null(ops, socket_recvmsg);
```

The actual security functions

At present all the socket functions just return 0!

```
static int cap_socket_sendmsg(struct socket *sock,
                             struct msghdr *msg, int size)
{
        return 0;
}

static int cap_socket_recvmsg(struct socket *sock,
                                  struct msghdr *msg,
                                 int size, int flags)
{
        return 0;
}
```

Control messages

Control messages may be used to pass *fds* from one unrelated process to another. They are not supported by *sendto* and we will not consider that facility in this course.

scm_cookie is defined in include/net/scm.h.

```
struct scm_cookie
{
     struct ucred creds;          /* Skb credentials      */
     struct scm_fp_list *fp;      /* Passed files         */
     unsigned long seq;           /* Connection seqno     */
};
```

The function, *scm_send(),* defined in include/net/scm.h is responsible for dispatching control messages. Recall that *sys_sendto()* unconditionally set the control elements of the *msg* structure to 0. However, it is possible that other drivers of this function would provide control elements. When invoked through *sys_sendto()* it simply saves the *uid, gid,* and *pid* in the *scm* structure. As we shall see, this data is discarded later in the path with no use having been made of it.

```
static __inline__ int scm_send(struct socket *sock,
struct msghdr *msg, struct scm_cookie *scm)
{
     memset(scm, 0, sizeof(*scm));
     scm->creds.uid = current->uid;
     scm->creds.gid = current->gid;
     scm->creds.pid = current->pid;
     if (msg->msg_controllen <= 0)
          return 0;
     return __scm_send(sock, msg, scm);
}
```

The *inet_sendmsg()* function

The protocol send routine for AF_INET is *inet_sendmsg()*. The value of *sk->num* is the local port number (or protocol number for sockets of type SOCK_RAW) in host byte order. If the socket has not been bound and this is the first transmission *the source port may be 0*. In this case it is necessary to call *inet_autobind()* (which was described in the discussion of UDP connect) to allocate an available source port.

```
int inet_sendmsg(struct kiocb *iocb, struct socket *sock,
                 struct msghdr *msg,
                 size_t size)
{
    struct sock *sk = sock->sk;

    /* We may need to bind the socket. */
    if (!inet_sk(sk)->num && inet_autobind(sk))
        return -EAGAIN;

```

The *sendmsg element* of the struct proto binding to the actual transport layer occurs here. For udp this maps to *udp_sendmsg*.

```
    return sk->sk_prot->sendmsg(iocb, sk, msg, size);
}
```

Data structures used by *udp_sendmsg*

The *ipcm_cookie* defined in include/net/ip.h holds the following information.

```
struct ipcm_cookie
{
    u32 addr;
    int oif;
    struct ip_options *opt;
};
```

addr	An IP address that is used at different times to store both the local and the remote IP address!
oif	Index of the output interface
opt	Pointer to the structure describing IP header options

IP Header options

As seen in CPSC 852 IP header options (1) do exist but (2) are very infrequently used. You will see in this course that their presence junks up the implementation in significant ways. You do not have to support them. This structure is used to map the standard IP header options during packet construction and decoding.

```
struct ip_options {
    __u32          faddr;     /* Saved first hop address */
    unsigned char  optlen;
    unsigned char  srr;
    unsigned char  rr;
    unsigned char  ts;
    unsigned char is_setbyuser:1, /* Set by setsockopt? */
         is_data:1, /* Options in __data, rather than skb */
         is_strictroute:1,  /* Strict source route      */
         srr_is_hit:1,   /* Packet dest addr was our one */
         is_changed:1,     /* IP checksum more not valid */
         rr_needaddr:1,   /* Need to record addr of outgoing
                             dev  */
         ts_needtime:1,   /* Need to record timestamp */
         ts_needaddr:1;   /* Need to record addr of outgoing
                             dev  */
         unsigned char router_alert;
         unsigned char __pad1;
         unsigned char __pad2;
         unsigned char __data[0];
};
```

The *udp_sock*

This structure is new to Linux 2.6 UDP. Its mission appears to be to support:

1. corking
2. encapsulation sockets
3. UDP Lite

```
struct udp_sock {
/* inet_sock has to be the first member */
   struct inet_sock inet;
   int              pending;      /* Any pending frames ? */
   unsigned int     corkflag;     /* Cork is required */
   __u16            encap_type;  /* Is this an Encapsocket? */
/*
 * Following member retains the infor to create a UDP header
 * when the socket is uncorked.
 */
   __u16            len;  /* total length of pending frames */
/*
 * Fields specific to UDP-Lite.
 */
   __u16            pcslen;
   __u16            pcrlen;
/* indicator bits used by pcflag: */
#define UDPLITE_BIT      0x1 /* set by udplite proto init */
#define UDPLITE_SEND_CC  0x2 /* set via udplite setsockopt */
#define UDPLITE_RECV_CC  0x4 /* set via udplite setsocktopt*/
   __u8             pcflag;  /* marks socket as UDP-Lite
                                    if > 0    */
   __u8             unused[3];
/*
 * For encapsulation sockets.
 */
  int (*encap_rcv)(struct sock *sk, struct sk_buff *skb);
};
```

The *cork* structure

“Corking” of a socket allow an application to call *sendto* or *write()* multiple times without any data actually being sent. For each call, a new *sk_buff* may or may not be allocated, and the data is copied from user space to kernel space. Eventually all of the *sk_buffs (frames/fragment)* are linked together to create a logical IP packet which is then sent.

The benefit of all this is a little mysterious. Possibly it is intended to make better use of GSO. The man page states that making use of it will make your code *non-portable.*

```
struct {
    unsigned int            flags;
    unsigned int            fragsize;
    struct ip_options       *opt;
    struct dst_entry        *dst;
    int     length; /* Total length of all frames */
    __be32                  addr;
    struct flowi            fl;
} cork;
```

dst — Routing information is determined when the first fragment is passed to a corked socket and the address of the route cache element is remembered here.

fl — The route key contains source/dest port/IP addresses. It is also filled in during processing of the first fragment and is eventually used to fill in transport and IP headers.

The *udp_sendmsg()* function

In the case of UDP *sendto()* the *sk->prot* structure points to *udp_prot*, and the *sendmsg* element of the *struct proto* is the function *udp_sendmsg()* which is defined in *net/ipv4/udp.c*. This function is the UDP handler for both the *sendto()* and *sendmsg()* API (and possibly others). At entry *len* carries the length of *user data*.

```
int udp_sendmsg(struct kiocb *iocb, struct sock *sk,
                struct msghdr *msg,
                size_t len)
{
   struct inet_sock *inet = inet_sk(sk);
   struct udp_sock *up = udp_sk(sk);
   int ulen = len;
   struct ipcm_cookie ipc;
   struct rtable *rt = NULL;
   int free = 0;
   int connected = 0;
   u32 daddr, faddr, saddr;
   u16 dport;
   u8  tos;
   int err;
```

The *corkreq* flag will be set if and only if "corking" was previously specifed via *setsockopt()* or the MSG_MORE flag was set.

```
   int corkreq = up->corkflag || msg->msg_flags & MSG_MORE;

```

Cork management

This is the code from *setsockopt* where the cork flag is set or cleared. The call to *udp_push_pending_frames()* forces all previously corked up frames on to the IP layer.

```
switch(optname) {
case UDP_CORK:
   if (val != 0) {
      up->corkflag = 1;
   } else {
      up->corkflag = 0;
      lock_sock(sk);
      udp_push_pending_frames(sk, up);
      release_sock(sk);
   }
```

The value of *len* is checked first for validity. The use of unsigned short integer type for *len* in the UDP header limits the size of a UDP datagram to 64K. However, the existence of the UDP and IP headers should also limit it to 65507 bytes. At any rate, a length of more than 64K is clearly bad.

```
        if (len > 0xFFFF)
                return -EMSGSIZE;

```

Out-of-band data is not supported by any UDP API.

```
    /*
     *      Check the flags.
     */

    if (msg->msg_flags & MSG_OOB)  /* Mirror BSD error message
                                          compatibility */
        return -EOPNOTSUPP;

    ipc.opt = NULL;
```

Corked sockets / pending data

UDP supports and operational mode in which the results of multiple calls to *sendto/sendmsg* can create as single IP datagram. The *up->pending* flag indicates that this is *not* the first fragment/frame element of the datagram.

```
if (up->pending) {
    /*
     * There are pending frames.
     * The socket lock must be held while it's corked.
     */
     lock_sock(sk);
     if (likely(up->pending)) {
          if (unlikely(up->pending != AF_INET)) {
                release_sock(sk);
                 return -EINVAL;
           }
      goto do_append_data;   <--- This is a big jump
      }
      release_sock(sk);
 }
```

Constructing *destination* addresses from the *sockaddr_in* or the *struct sock*.

Arrival here implies this is the first or first and only fragment. The UDP header length is added to the length accumulator.

```
ulen += sizeof(struct udphdr);

```

The destination IP and port addresses *must* be specified via the *sockaddr_in* for a disconnected socket and *may* be specified for a connected socket. If the application provided a *struct sockaddr_in* the *msg_name* field points to it. If none was provided *msg_name* will be NULL.

COP will only support sending on connected sockets as indicated by

sk->sk_state == TCP_ESTABLISHED.

COP should silently ignore any *msg_name* passed to *cop_sendmsg()*

```
/*
 * Get and verify the address.
 */
if (msg->msg_name) {
   struct sockaddr_in * usin =
            (struct sockaddr_in*)msg->msg_name;
   if (msg->msg_namelen < sizeof(*usin))
      return -EINVAL;
   if (usin->sin_family != AF_INET) {
      if (usin->sin_family != AF_UNSPEC)
         return -EAFNOSUPPORT;
   }

```

Processing the *struct sockaddr_in*

If *struct sockaddr_in* was provided, the destination IP address and port number are extracted and saved. The *destination port address must be non-zero* in the *struct sockaddr_in* but *the destination IP address may be zero*.

```
        daddr = usin->sin_addr.s_addr;
        dport = usin->sin_port;
        if (dport == 0)
           return -EINVAL;
```

No *sockaddr_in* provided

If the pointer to the *sockaddr_in* structure is NULL, the socket must be already connected or the send process returns an error here. If the socket is connected the destination IP address and port are extracted from the *struct sock.*

```
     } else {
        if (sk->sk_state != TCP_ESTABLISHED)
           return -EDESTADDRREQ;
        daddr = inet->daddr;
        dport = inet->dport;
        /* Open fast path for connected socket.
           Route will not be used, if any options  are set.
         */
        connected = 1;
     }
```

Constructing the *source* IP address and port.

Since *inet_sendmsg()* called *inet_autobind()* if the source port in the socket was 0, the source port is guaranteed to be set here. The output device interface index is set from the *struct sock.* The *bound_dev_if* is set to NULL at socket creation time but may be set to a specific interface via *setsockopt()*. The source IP address to which the socket may be is bound is temporarily held in the *ipc* for unknown reasons.

```
ipc.addr = inet->saddr;

ipc.oif = sk->bound_dev_if;
```

The value of *msg_controllen* was set to NULL by *sys_sendto(),* but could presumably not be NULL when the *sendmsg()* API is used. Control messages are sent using the *ip_cmsg_send* function.

```
if (msg->msg_controllen) {
   err = ip_cmsg_send(msg, &ipc);
   if (err)
      return err;
   if (ipc.opt)
      free = 1;
   connected = 0;
}
```

IP header options may also be set by the application via *setsockopt()* and they are stored in the *inet_sock.* If present, a pointer to them is stored in the cookie.

```
if (!ipc.opt)
   ipc.opt = inet->opt;
```

This section here is something of an oddity. Note that *ipc.addr* was set to *inet->saddr* above. Here it is set to the destination address after the source address is saved in *saddr*. The inner if is related to source routed datagrams. It is replacing the destination address with the address of the first intermediate hop from the source route list. At this point *daddr* is the value specified in the struct *sockaddr_in* or if not *sockaddr_in* was provided *and* the socket was connected, then *daddr* was taken from *inet->daddr*.

```
saddr = ipc.addr;
ipc.addr = faddr = daddr;

if (ipc.opt && ipc.opt->srr) {
   if (!daddr)
      return -EINVAL;
   faddr = ipc.opt->faddr;
   connected = 0;
}
```

Routing options

The *RT_TOS macro* retrieves the low order 5 bits from the *tos* field of the *struct sock*. These will be 0 unless set by *setsockopt()*.

```
tos = RT_TOS(inet->tos);
```

The RTO_ONLINK bit forces the destination (or next hop in case of a *strict* source route) to be reachable in a single hop.

```
if (sock_flag(sk, SOCK_LOCALROUTE) ||
    (msg->msg_flags & MSG_DONTROUTE) ||
    (ipc.opt && ipc.opt->is_strictroute)) {
   tos |= RTO_ONLINK;
   connected = 0;
}
```

Multicasts

Recall that a multicast is always associated with a specific interface. If the *oif* or *saddr* is not already set here they are set using values that were specified when the multicast was set up.

```
bcopy((char *)hp->h_addr, &mreq.imr_interface, 4);
bcopy((char *)mgroup, &mreq.imr_multiaddr, 4);
status =  setsockopt(sock, 0, IP_ADD_MEMBERSHIP,
                           (char *)&mreq, sizeof(mreq));
```

```
if (MULTICAST(daddr)) {
   if (!ipc.oif)
      ipc.oif = inet->mc_index;
   if (!saddr)
      saddr = inet->mc_addr;
   connected = 0;
}
```

Routing the datagram

If the socket is connected there may be a valid route cache element already associated with the *struct sock*. The function *sk_dst_check* actually returns a pointer to *struct dst_entry*, but since the *struct rtable* is defined as a union of a *struct dst_entry* with a *struct rtable **, it is safe and correct to cast the pointer to *struct dst_entry* to a pointer to *struct rtable*. If the route cache entry is no longer valid, 0 will be returned by *sk_dst_check()*. Your protocol must verify the route for each packet that is sent.

```
if (connected)
   rt = (struct rtable*)sk_dst_check(sk, 0);

```

For connected sockets with an obsolete *dst_entry* and for unconnected sockets, *rt* will be NULL here. In these cases it is necessary to call *ip_route_output_flow()* which will first try to resolve the route via route cache and will invoke *ip_route_output_slow()* to resolve the route from the FIB if it cannot be found in the cache. Your protocol must deal with this situation.

```
if (rt == NULL) {
    struct flowi fl = {.oif = ipc.oif,
                       .nl_u = { .ip4_u =
                               { .daddr = faddr,
                                 .saddr = saddr,
                                 .tos = tos } },
                       .proto = IPPROTO_UDP,
                       .uli_u = { .ports =
                              { .sport = inet->sport,
                                .dport = dport } } };
    err = ip_route_output_flow(&rt, &fl, sk, !
            (msg->msg_flags&MSG_DONTWAIT));
    if (err)
       goto out;

    err = -EACCES;
```

This appears to be checking to see if the broadcast attributes of the route and the *struct sock* are mutually incompatible with respect to the *broadcast* attribute.

```
      err = -EACCES;
      if ((rt->rt_flags & RTCF_BROADCAST) &&
          !sock_flag(sk, SOCK_BROADCAST))
         goto out;
```

If the socket is connected but the existing *dst_cache* entry was obsolete, then it is updated here to point to the element returned by *ip_route_output_flow*.   You need to do this as well.

```
      if (connected)
         sk_dst_set(sk, dst_clone(&rt->u.dst));
   }  //endif rt was NULL
```

UGH... the “confirm facility” is ugly --- the jump out of line and back even uglier.

```
   if (msg->msg_flags&MSG_CONFIRM)
      goto do_confirm;
back_from_confirm:
```

Final choice of IP address

Source and destination IP addresses are finalized here. The source is taken from the route. If a destination was previously stored in the *ipc* it takes precedence over the route.

```
saddr = rt->rt_src;
if (!ipc.addr)
   daddr = ipc.addr = rt->rt_dst;
```

Way back at the start *up->pending* was tested and if true, all of this code was jumped over via the *goto do_append_data.* If somehow data has become pending in the meantime it appears to be a fatal error.

```
lock_sock(sk);
if (unlikely(up->pending)) {
   /* The socket is already corked while preparing it. */
   /* ... which is an evident application bug. --ANK */
   release_sock(sk);

   LIMIT_NETDEBUG(KERN_DEBUG "udp cork app bug 2\n");
   err = -EINVAL;
   goto out;
}
```

Setting up the *cork*

Since it may be possible to add more user data to the logical IP packet being constructed, it is necessary to remember where the packet is going and how long it is. The addresses are kept in the cork which is part of the *inet_sock* and the length is in the *udp_sock()*.

```
/*
 * Now cork the socket to pend data.
 */
inet->cork.fl.fl4_dst = daddr;
inet->cork.fl.fl_ip_dport = dport;
inet->cork.fl.fl4_src = saddr;
inet->cork.fl.fl_ip_sport = inet->sport;
up->pending = AF_INET;

```

Convergence of the *first and not first* fragments.

For a not first fragment all of the code involving control messages, address checking and routing was jumped over. The two paths converge here.

Here the length of the addtional user data is added to the length maintained in the *udp_sock.*

```
do_append_data:
   up->len += ulen;
```

Allocating the *sk_buff* and copying data

The *ip_append_data* function is responsible of allocating the *struct sk_buff* and copying the data to it. The *ip_generic_getfrag()* function does the actual copying of data from user space into the *sk_buff.* You will do this in line in a more sane way.

```
   err = ip_append_data(sk, ip_generic_getfrag,
                             msg->msg_iov, ulen,
                        sizeof(struct udphdr), &ipc, rt,
                        corkreq ? msg->msg_flags|MSG_MORE :
                        msg->msg_flags);
```

Sending the packet

Recall that the *corkreq* flag will be set if and only if "corking" was previously specifed via setsockopt() or the MSG_MORE flag was set. So that will almost never be true and the *udp_push_pending_frames()* will trigger the transmission of the single frame that was just constructed.

```
    if (err)
       udp_flush_pending_frames(sk);
    else if (!corkreq)
       err = udp_push_pending_frames(sk, up);
    release_sock(sk);

```

The exit from *udp_sendmsg()*

On return *udp_sendmsg, ip_rt_put()* is called to decrement the reference count of the packet's route cache element structure. This was incremented by the call to *sk_dst_check()* or *sk_dst_set().* It is also incremented in *sk_dst_clone()* when the pointer to the route cache element is stored in the *sk_buff. (Which hasn't happened yet!)* You need to be careful to properly handle route reference counting.

```
out:
    ip_rt_put(rt);
    if (free)
       kfree(ipc.opt);
    if (!err) {
       UDP_INC_STATS_USER(UDP_MIB_OUTDATAGRAMS);
       return len;
    }
    return(err);
```

The jump to *back_from_confirm* will be taken *unless* both the value of *len* is *0* and the MSG_PROBE flag is 0.

```
do_confirm:
   dst_confirm(&rt->u.dst);
   if (!(msg->msg_flags&MSG_PROBE) || len)
      goto back_from_confirm;
   err = 0;
   goto out;
}
```

The *udp_push_pending_frames* function

This function has two missions:

- Fill in the UDP header
- Compute the checksum

```
static int udp_push_pending_frames(struct sock *sk,
                                   struct udp_sock *up)
{
   struct inet_sock *inet = inet_sk(sk);
   struct flowi *fl = &inet->cork.fl;
   struct sk_buff *skb;
   struct udphdr *uh;
   int err = 0;

```

The *ip_append_data()* function leaves the *skb(s)* on the *sk's* write queue. So if per chance the queue is empty, there is nothing to do.

```
/* Grab the skbuff where UDP header space exists. */
   if ((skb = skb_peek(&sk->sk_write_queue)) == NULL)
      goto out;

```

UDP header creation

This function is trusting that *ip_append_data()* has properly set up *skb->h.uh.* You can't depend on that! Recall that the cork contained a flow information (route key) structure in which the address data was saved and the length was saved in the *udpsock* structure.

```
/*
 * Create a UDP header
 */
uh = skb->h.uh;
uh->source = fl->fl_ip_sport;
uh->dest = fl->fl_ip_dport;
uh->len = htons(up->len);
uh->check = 0;

```

If checksumming is disabled, skip to the send code.

```
if (sk->sk_no_check == UDP_CSUM_NOXMIT) {
   skb->ip_summed = CHECKSUM_NONE;
   goto send;
}

```

Checksumming

If checksumming is not disabled then it must be addressed here. The "easy case" is when there is only one *sk_buff* on the *write queue.*

```
if (skb_queue_len(&sk->sk_write_queue) == 1) {
   /*
    * Only one fragment on the socket.
    */
   if (skb->ip_summed == CHECKSUM_HW) {
      skb->csum = offsetof(struct udphdr, check);
      uh->check = ~csum_tcpudp_magic(fl->fl4_src,
                                fl->fl4_dst,
                                 up->len, IPPROTO_UDP, 0);
   } else {
      skb->csum = csum_partial((char *)uh,
            sizeof(struct udphdr), skb->csum);
      uh->check = csum_tcpudp_magic(fl->fl4_src,
                                fl->fl4_dst,
                                up->len, IPPROTO_UDP, skb->csum);
      if (uh->check == 0)
         uh->check = -1;
   }
```

More than one *sk_buff* on the write queue.

```
} else {
   unsigned int csum = 0;
   /*
    * HW-checksum won't work as there are two or more
    * fragments on the socket so that all csums of sk_buffs
    * should be together.
    */
   if (skb->ip_summed == CHECKSUM_HW) {
      int offset = (unsigned char *)uh - skb->data;
      skb->csum = skb_checksum(skb, offset, skb->len -
                                       offset, 0);

      skb->ip_summed = CHECKSUM_NONE;
   } else {
      skb->csum = csum_partial((char *)uh,
            sizeof(struct udphdr), skb->csum);
   }

   skb_queue_walk(&sk->sk_write_queue, skb) {
      csum = csum_add(csum, skb->csum);
   }
   uh->check = csum_tcpudp_magic(fl->fl4_src, fl->fl4_dst,
         up->len, IPPROTO_UDP, csum);
   if (uh->check == 0)
      uh->check = -1;
}
```

Sending the datagram

The *ip_push_pending_frames()* function builds the *ip header* and passes the logical packet on to the net filter layer.

```
send:
   err = ip_push_pending_frames(sk);
out:
   up->len = 0;
   up->pending = 0;
   return err;
}
```

The *ip_push_pending_frames()* function

The mission of this function is to combine all of the fragment *sk_buffs* on the write queue into a single logical *sk_buff* structure and pass it on the the *netfilter* layer for processing.

```
/*
 * Combined all pending frags on the socket as one IP datagram
 * and push them out.
 */
int ip_push_pending_frames(struct sock *sk)
{
   struct sk_buff *skb, *tmp_skb;
   struct sk_buff **tail_skb;
   struct inet_sock *inet = inet_sk(sk);
   struct ip_options *opt = NULL;
   struct rtable *rt = inet->cork.rt;
   struct iphdr *iph;
   __be16 df = 0;
   __u8 ttl;
   int err = 0;

```

The *ip_append_data()* function leaves the *sk_buff*(s) on the struct sock's write queue. So if per chance the queue is empty, there is nothing to do. The first fragment in the queue carries the UDP header.

- The pointer *skb* will always point to the first fragment.
- The pointer *tail_skb* will move along the list pointing to the place where the next link is to be stored as the fragments are logically linked together. For the ONLY the first packet will *tail_skb* point to the *frag_list.*

```
   if ((skb = __skb_dequeue(&sk->sk_write_queue)) == NULL)
      goto out;

   tail_skb = &(skb_shinfo(skb)->frag_list);

```

The *skb_pull()* function increments the *skb->data* pointer and decrements the value of *skb->len* effectively removing data from the head of a buffer and returning it to the headroom. This code assumes that nh.raw is set properly and forces data to point to the same spot.

```
/* move skb->data to ip header from ext header */
   if (skb->data < skb->nh.raw)
      __skb_pull(skb, skb->nh.raw - skb->data);
```

Constructing a single IP packet from the fragments

This loop processes the remainder of the write queue removing *sk_buffs* which remain write queue linking them, on the fragment list and accumulating the total length. All of these fragments evidently held references to the *sk* and since all of these fragments are being converted here in to a single *struct sk_buff* their references are dropped and there destructors nullified.

In a properly constructed fragmented packet the *frag_list* pointer of the first fragment points to the head of the fragment chain. The remainder of the packets are linked together using the *skb->next* pointers and *not the frag_list.* Recall that *skb->data_len* keeps track of the amount of data in the fragment chain.

Fragments do not carry headers. The call to *skb_pull()* is evidently trying to ensure that the *data* pointer points to the user data. Hence it must have been the case that *skb->h.raw* pointed there.

```
   while ((tmp_skb = __skb_dequeue(&sk->sk_write_queue))
                                 != NULL) {
      __skb_pull(tmp_skb, skb->h.raw - skb->nh.raw);
      *tail_skb = tmp_skb;
      tail_skb = &(tmp_skb->next);
      skb->len += tmp_skb->len;
      skb->data_len += tmp_skb->len;
      skb->truesize += tmp_skb->truesize;
      __sock_put(tmp_skb->sk);
      tmp_skb->destructor = NULL;
      tmp_skb->sk = NULL;
   }
```

Path MTU discovery and TTL processing

You should just set *df* to htons(IP_DF), and use *ip_select_ttl()* to initialize the *ttl* back when the socket was initially connected.

```

/* Unless user demanded real pmtu discovery (IP_PMTUDISC_DO), we allow
 * to fragment the frame generated here. No matter, what transforms
 * how transforms change size of the packet, it will come out.
 */
  if (inet->pmtudisc != IP_PMTUDISC_DO)
     skb->local_df = 1;

/* DF bit is set when we want to see DF on outgoing frames.
 * If local_df is set too, we still allow to fragment this
    * frame locally. */
  if (inet->pmtudisc == IP_PMTUDISC_DO ||
      (skb->len <= dst_mtu(&rt->u.dst) &&
       ip_dont_fragment(sk, &rt->u.dst)))
     df = htons(IP_DF);

  if (inet->cork.flags & IPCORK_OPT)
     opt = inet->cork.opt;

  if (rt->rt_type == RTN_MULTICAST)
     ttl = inet->mc_ttl;
  else
     ttl = ip_select_ttl(inet, &rt->u.dst);

```

Building the IP header

You will need to do something like this. However, you can do most of it only once at connect time and then *memcpy()* it into place and only have to update the *id, length*, and *checksum.*

```
iph = (struct iphdr *)skb->data;
iph->version = 4;
iph->ihl = 5;
if (opt) {
   iph->ihl += opt->optlen>>2;
   ip_options_build(skb, opt, inet->cork.addr, rt, 0);
}

iph->tos = inet->tos;
iph->tot_len = htons(skb->len);
iph->frag_off = df;
ip_select_ident(iph, &rt->u.dst, sk);

iph->ttl = ttl;
iph->protocol = sk->sk_protocol;
iph->saddr = rt->rt_src;
iph->daddr = rt->rt_dst;
```

The *ip_send_check()* function is an inline function that computes the header checksum. You will need to do this.

```
ip_send_check(iph);

skb->priority = sk->sk_priority;
skb->dst = dst_clone(&rt->u.dst);
```

```
/* Generate a checksum for an outgoing IP datagram. */
__inline__ void ip_send_check(struct iphdr *iph)
{
   iph->check = 0;
   iph->check = ip_fast_csum((unsigned char *)iph, iph->ihl);
}
```

Sending the packet

Packets are sent by passing them to the netfilter layer that is responsible for such things as firewalls and NAT. This is the mechanism that you will use.

The *dst_output()* function is known as an "OK function". The packet will be passed to the that function if it is not dropped by the *netfilter* layer. The OK function used here just passes the packet on using the *skb->dst->output()* function.

```
/* Netfilter gets whole the not fragmented skb. */
   err = NF_HOOK(PF_INET, NF_IP_LOCAL_OUT, skb, NULL,
                 skb->dst->dev, dst_output);
   if (err) {
      if (err > 0)
         err = inet->recverr ? net_xmit_errno(err) : 0;
      if (err)
         goto error;
   }

out:
   inet->cork.flags &= ~IPCORK_OPT;
   kfree(inet->cork.opt);
   inet->cork.opt = NULL;
   if (inet->cork.rt) {
      ip_rt_put(inet->cork.rt);
      inet->cork.rt = NULL;
   }
   return err;

error:
   IP_INC_STATS(IPSTATS_MIB_OUTDISCARDS);
   goto out;
}
```

The *dst_output_function*

This function uses the indirect binding established in routing to determine the next output function to handle the packet. The *skb->dst->output* function points to *ip_output()* if and only if the packet is going to be sent to another host.

```
static inline int dst_output(struct sk_buff *skb)
{
        return skb->dst->output(skb);
}
```

Summary

You should create a *ntp_sock_t* structure at socket creation time. It should contain an *ntp_hdr_t* and a *struct iphdr*. These should be initialized to the extent possible at connect time.

If failure is detected at any point, bail out but *be sure* to release held resources and references. Items highlighted in *green* have been covered in this section.

1 - If the sock is not in the TCP_ESTABLISHED state, return -ENOTCONN.
2 - Use *sk_dst_check()* to verify the route. Return -ENOTCONN if it doesn't work.
3 - Allocate an *sk_buff*, set up the header pointers correctly, and attach the route cache pointer to it.
4 - Copy the user data to the *buffer*
5 - Copy the *cop_hdr* to the *buffer* and fill in some missing elements
6 - Copy the *iphdr* to the buffer and fill in missing elements
7 - Invoke NF_HOOK() to dispatch the packet
8 - Provide an OK function that will pass the packet on to the *output* function in the *dst_entry* of the *sk_buff.*

As we proceed with the project it will be necessary to support internal callers of *send* (for example the receive code will eventually have to call send to send acknowledgements and to do retransmissions. A properly modularlized version will *not require* duplication of massive amounts of code.

The *ip_append_data()* function.

This is an unbelievably messy function. It has way too many parameters indicating an undesirable level of coupling with its caller. The "getfrag" function is a callback that actually points to *ip_generic_getfrag()* whose mission is to actually copy data from user space into the *sk_buff().*

```
int ip_append_data(struct sock *sk,
      int getfrag(void *from, char *to, int offset, int len,
      int odd, struct sk_buff *skb),
      void *from, int length, int transhdrlen,
      struct ipcm_cookie *ipc, struct rtable *rt,
      unsigned int flags)
{
   struct inet_sock *inet = inet_sk(sk);
   struct sk_buff *skb;

   struct ip_options *opt = NULL;
   int hh_len;
   int exthdrlen;
   int mtu;
   int copy;
   int err;
   int offset = 0;
   unsigned int maxfraglen, fragheaderlen;
   int csummode = CHECKSUM_NONE;

   if (flags & MSG_PROBE)
      return 0;

```

Even if corking is not explicitly enabled, the cork mechanism is unconditionally set up whenever *ip_append_data* is called with an empty *write_queue*.

```
if (skb_queue_empty(&sk->sk_write_queue)) {
   /*
    * setup for corking.
    */
   opt = ipc->opt;
```

Header options are copied to the cork here.

```
   if (opt) {
      if (inet->cork.opt == NULL) {
         inet->cork.opt = kmalloc(sizeof(struct
        ip_options) + 40, sk->sk_allocation);
         if (unlikely(inet->cork.opt == NULL))
            return -ENOBUFS;
      }
      memcpy(inet->cork.opt, opt, sizeof(struct
                  ip_options)+opt->optlen);
      inet->cork.flags |= IPCORK_OPT;
      inet->cork.addr = ipc->addr;
   }
```

The fragsize holds the routing system's view of path mtu. The *transhdrlen* is passed by the caller and represents the size of the UDP header here.

```
   dst_hold(&rt->u.dst);
   inet->cork.fragsize = mtu = dst_mtu(rt->u.dst.path);
   inet->cork.rt = rt;
   inet->cork.length = 0;
   sk->sk_sndmsg_page = NULL;
   sk->sk_sndmsg_off = 0;
   if ((exthdrlen = rt->u.dst.header_len) != 0) {
      length += exthdrlen;
      transhdrlen += exthdrlen;
   }
```

If the write queue is not empty then the cork is already setup and we don't have to worry about transport header length in this fragrament.

```
} else {  // write queue not empty
   rt = inet->cork.rt;
   if (inet->cork.flags & IPCORK_OPT)
      opt = inet->cork.opt;

   transhdrlen = 0;
   exthdrlen = 0;
   mtu = inet->cork.fragsize;
}
```

This is attempting to ensure that total length including all headers and data remains less than the 64K limit on an IP packet. The LL_RESERVED_SPACE macro appears to be the new recommended method for retrieving the MAC header length.

```
hh_len = LL_RESERVED_SPACE(rt->u.dst.dev);

fragheaderlen = sizeof(struct iphdr) +
            (opt ? opt->optlen : 0);
maxfraglen = ((mtu - fragheaderlen) & ~7) +
                            fragheaderlen;

if (inet->cork.length+length > 0xFFFF - fragheaderlen) {
   ip_local_error(sk, EMSGSIZE, rt->rt_dst, inet->dport,
                            mtu-exthdrlen);
   return -EMSGSIZE;
}
```

This section is trying to take advantage of UDP checksum offload if it exts.

```

    /*
     * transhdrlen > 0 means that this is the first fragment
     * and we wish
     * it won't be fragmented in the future.
     */

    if (transhdrlen &&
        length + fragheaderlen <= mtu &&
        rt->u.dst.dev->features & NETIF_F_ALL_CSUM &&
        !exthdrlen)
       csummode = CHECKSUM_HW;

```

UFO = UDP fragmentation offload. If it is supported it means that the NIC can consume a 64Kb datagram and resegment (one hopes not fragment) into multiple IP packets.

```
    inet->cork.length += length;
    if (((length > mtu) && (sk->sk_protocol == IPPROTO_UDP))
        &&  (rt->u.dst.dev->features & NETIF_F_UFO)) {

       err = ip_ufo_append_data(sk, getfrag, from,
                   length, hh_len,
                   fragheaderlen, transhdrlen, mtu,
                   flags);
       if (err)
          goto error;
       return 0;
    }
```

So what's going on in the rest of this function.... I give up!

```
/* So, what's going on in the loop below?
 *
 * We use calc fragment length to generate chained skb,
 * each of segments is IP fragment ready for sending to
   network after
 * adding appropriate IP header.
 */

   if ((skb = skb_peek_tail(&sk->sk_write_queue)) == NULL)
      goto alloc_new_skb;

   while (length > 0) {
 /* Check if the remaining data fits into current packet. */
      copy = mtu - skb->len;
      if (copy < length)
         copy = maxfraglen - skb->len;
      if (copy <= 0) {
         char *data;
         unsigned int datalen;
         unsigned int fraglen;
         unsigned int fraggap;
         unsigned int alloclen;
         struct sk_buff *skb_prev;
```

Buffer allocation

A *go_to* target nested inside a loop and an if is always a *bad sign* but this is where a new *sk_buff* is allocated.

```
alloc_new_skb:
         skb_prev = skb;
         if (skb_prev)
            fraggap = skb_prev->len - maxfraglen;
         else
            fraggap = 0;


         /*
          * If remaining data exceeds the mtu,
          * we know we need more fragment(s).
          */
         datalen = length + fraggap;
         if (datalen > mtu - fragheaderlen)
            datalen = maxfraglen - fragheaderlen;
         fraglen = datalen + fragheaderlen;

         if ((flags & MSG_MORE) &&
             !(rt->u.dst.dev->features&NETIF_F_SG))
            alloclen = mtu;
         else
            alloclen = datalen + fragheaderlen;

         /* The last fragment gets additional space at tail.
          * Note, with MSG_MORE we overalloc on fragments,
          * because we have no idea what fragment will be
          * the last.
          */
         if (datalen == length + fraggap)
            alloclen += rt->u.dst.trailer_len;

```

New *sk_buff* allocation

Amongst all of this insanity, here is a typical and correct way to allocate a send *sk_buff*. The call will block on *sndbuf* quota exceeded unless MSG_DONTWAIT is set. The value of *transhdrlen* will be non-zero only for the first fragment.

Otherwise, the *non-blocking sock_wmalloc()* is called as long as the socket is not 2x over quota. The 1 parameter following the length is the *force* flag that allows *sock_wmalloc()* to ignore quota overflow.

```
        if (transhdrlen) {
           skb = sock_alloc_send_skb(sk,
                 alloclen + hh_len + 15,
                 (flags & MSG_DONTWAIT), &err);
        } else {
           skb = NULL;
           if (atomic_read(&sk->sk_wmem_alloc) <=
               2 * sk->sk_sndbuf)
              skb = sock_wmalloc(sk,
                       alloclen + hh_len + 15, 1,
                       sk->sk_allocation);
           if (unlikely(skb == NULL))
              err = -ENOBUFS;
        }

        if (skb == NULL)
           goto error;

        /*
         * Fill in the control structures
         */
        skb->ip_summed = csummode;
        skb->csum = 0;
        skb_reserve(skb, hh_len);

```

If this is not the first fragment then all of the **hdrlens* are 0.

```
/*
 * Find where to start putting bytes.
 */
data = skb_put(skb, fraglen);
skb->nh.raw = data + exthdrlen;
data += fragheaderlen;
skb->h.raw = data + exthdrlen;

if (fraggap) {
   skb->csum = skb_copy_and_csum_bits(
      skb_prev, maxfraglen,
      data + transhdrlen, fraggap, 0);
   skb_prev->csum = csum_sub(skb_prev->csum,
              skb->csum);
   data += fraggap;
   pskb_trim_unique(skb_prev, maxfraglen);
}

```

The actual copy from user occurs in the indirect call to *getfrag()*

```
   copy = datalen - transhdrlen - fraggap;
   if (copy > 0 && getfrag(from, data + transhdrlen,
     offset, copy, fraggap, skb) <
      err = -EFAULT;
      kfree_skb(skb);
   offset += copy;
   length -= datalen - fraggap;
   transhdrlen = 0;
   exthdrlen = 0;
   csummode = CHECKSUM_NONE;

   /*
    * Put the packet on the pending queue.
    */
   __skb_queue_tail(&sk->sk_write_queue, skb);
   continue;
}

```

```
if (copy > length)
   copy = length;

if (!(rt->u.dst.dev->features & NETIF_F_SG)) {
   unsigned int off;

   off = skb->len;
   if (getfrag(from, skb_put(skb, copy),
         offset, copy, off, skb) < 0) {
      __skb_trim(skb, off);
      err = -EFAULT;
      goto error;
   }

} else {
   int i = skb_shinfo(skb)->nr_frags;
   skb_frag_t *frag = &skb_shinfo(skb)->frags[i-1];
   struct page *page = sk->sk_sndmsg_page;
   int off = sk->sk_sndmsg_off;
   unsigned int left;

   if (page && (left = PAGE_SIZE - off) > 0) {
      if (copy >= left)
         copy = left;
      if (page != frag->page) {
         if (i == MAX_SKB_FRAGS) {
            err = -EMSGSIZE;
            goto error;
         }
         get_page(page);
         skb_fill_page_desc(skb, i, page, sk
                        ->sk_sndmsg_off, 0);
         frag = &skb_shinfo(skb)->frags[i];
      }
```

```
         } else if (i < MAX_SKB_FRAGS) {
            if (copy > PAGE_SIZE)
               copy = PAGE_SIZE;
            page = alloc_pages(sk->sk_allocation, 0);
            if (page == NULL)  {
               err = -ENOMEM;
               goto error;
            }
            sk->sk_sndmsg_page = page;
            sk->sk_sndmsg_off = 0;

            skb_fill_page_desc(skb, i, page, 0, 0);
            frag = &skb_shinfo(skb)->frags[i];
            skb->truesize += PAGE_SIZE;
            atomic_add(PAGE_SIZE, &sk->sk_wmem_alloc);
         } else {
            err = -EMSGSIZE;
            goto error;
         }
         if (getfrag(from, page_address(frag->page)+
                    frag->page_offset+frag->size, offset,
                     copy, skb->len, skb) < 0) {
            err = -EFAULT;
            goto error;
         }
         sk->sk_sndmsg_off += copy;
         frag->size += copy;
         skb->len += copy;
         skb->data_len += copy;
      }
      offset += copy;
      length -= copy;
   }

   return 0;

error:
   inet->cork.length -= length;
   IP_INC_STATS(IPSTATS_MIB_OUTDISCARDS);
   return err;
}
```

The *ip_generic_getfrag()* function

Formerally called, *udp_getfrag(),* this function is a callback function provided to *ip_append_data().* Its mission is to copy fragments of the datagram from user space into *sk_buffs* that are allocated by *ip_append_data()* and to compute the UDP checksum. You may want to use the *memcpy_from_iovecend()* function.

```
int
ip_generic_getfrag(void *from, char *to, int offset,
                   int len, int odd, struct sk_buff *skb)
{
   struct iovec *iov = from;

   if (skb->ip_summed == CHECKSUM_HW) {
      if (memcpy_fromiovecend(to, iov, offset, len) < 0)
         return -EFAULT;
   } else {
      unsigned int csum = 0;
      if (csum_partial_copy_fromiovecend(to, iov,
                                  offset, len, &csum) < 0)
         return -EFAULT;
      skb->csum = csum_block_add(skb->csum, csum, odd);
   }
   return 0;
}
```