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)

```plaintext
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 details.
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.

```c
57 struct msghdr {
58    void   *msg_name;           /* Socket name           */
59    int     msg_namelen;        /* Length of name        */
60    struct iovec *  msg_iov;    /* Data blocks           */
61    __kernel_size_t msg_iovlen; /* Number of blocks      */
62    void    *msg_control;  /* Per protocol magic     */
63    __kernel_size_t msg_controllen; /* Length of cmsg list */
64    unsigned        msg_flags;  /* Number of blocks      */
65};
```

- **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.

```c
20 struct iovec  
21 {  
22   void __user *iov_base;  /* BSD uses caddr_t (1003.1g requires void *) */  
23   __kernel_size_t iov_len; /* Must be size_t (1003.1g) */  
24 };  
```

- *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.

```
1564 asmlinkage long sys_sendto(int fd, void __user * buff,
   size_t len, unsigned flags,
   struct sockaddr __user *addr, int addr_len)
1566{
1567    struct socket *sock;
1568    char address[MAX_SOCK_ADDR];
1569    int err;
1570    struct msghdr msg;
1571    struct iovec iov;
1572    int fput_needed;
1573    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.

1574 sock_file = fget_light(fd, &fput_needed);
1576 if (!sock_file)
1577    return -EBADF;
1578 sock = sock_from_file(sock_file, &err);
1580 if (!sock)
1581    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.

```c
1582        iov.iov_base=buff;
1583        iov.iov_len=len;
1584        msg.msg_name=NULL;
1585        msg.msg_iov=&iov;
1586        msg.msg_iovlen=1;
1587        msg.msg_control=NULL;
1588        msg.msg_controllen=0;
1589        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`.

```c
1590    if (addr) {
1591        err = move_addr_to_kernel(addr, addr_len, address);
1592        if (err < 0)
1593            goto out_put;
1594        msg.msg_name=address;
1595        msg.msg_namelen=addr_len;
1596    }
```

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

```c
1597    if (sock->file->f_flags & O_NONBLOCK)
1598        flags |= MSG_DONTWAIT;
1599    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`.

```c
1600    err = sock_sendmsg(sock, &msg, len);
1601
1602 out_put:
1603    fput_light(sock_file, fput_needed);
1604    return err;
1605 }  
1606
```
Asynchronous I/O

- One mechanism 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)
   (transferring/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**.

```c
85 struct kiocb {
86    struct list_head     ki_run_list;
87    long                 ki_flags;
88    int                  ki_users;
89    unsigned             ki_key;         /* id of this request */
90
91    struct file          *ki_filp;
92    struct kioctx       *ki_ctx;   /* may be NULL for sync ops */
93    int                  (*ki_cancel)(struct kiocb *, struct io_event *);
94    ssize_t             (*ki_retry)(struct kiocb *);
95    void                (*ki_dtor)(struct kiocb *);
96
97     union {
98         void __user         *user;
99         struct task_struct *tsk;
100    } ki_obj;
101
102   __u64              ki_user_data;   /* user's data for completion */
103   wait_queue_t        ki_wait;
104   loff_t             ki_pos;
105
106   void                *private;   /* State that we remember to be able to restart/retry */
107   /* State that we remember to be able to restart/retry */
108   unsigned short      ki_opcode;
109   size_t             ki_nbytes;    /* copy of iocb->aio_nbytes */
110   char                __user *ki_buf; /* remaining iocb->aio_buf */
111   size_t             ki_left;     /* remaining bytes */
112   long                ki_retried;    /* just for testing */
113   long                ki_kicked;    /* just for testing */
114   long                ki_queued;    /* just for testing */
115
116   struct list_head    ki_list;    /* the aio core uses this */
117   };
118 }
```
The *sock_iocb*

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

```c
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.

```c
599 int sock_sendmsg(struct socket *sock,
                      struct msghdr *msg, size_t size)
600 {
601    struct kiocb iocb;
602    struct sock_iocb siocb;
603    int ret;
604
605    init_sync_kiocb(&iocb, NULL);
606    iocb.private = &siocb;
607    ret = __sock_sendmsg(&iocb, sock, msg, size);
608    if (-EIOCBQUEUED == ret)
609        ret = wait_on_sync_kiocb(&iocb);
610    return ret;
121 #define init_sync_kiocb(x, filp)                        \  \
122    do {                                            \  \
123            struct task_struct *tsk = current;      \  \
124            (x)->ki_flags = 0;                      \  \
125            (x)->ki_users = 1;                      \  \
126            (x)->ki_key = KIOCB_SYNC_KEY;           \  \
127            (x)->ki_filp = (filp);                  \  \
128            (x)->ki_ctx = NULL;                     \  \
129            (x)->ki_cancel = NULL;                  \  \
130            (x)->ki_retry = NULL;                   \  \
131            (x)->ki_dtor = NULL;                    \  \
132            (x)->ki_obj.tsk = tsk;                   \  \
133            (x)->ki_user_data = 0;                 \  \
134            init_wait((&(x)->ki_wait));             \  
135        } while (0)
```

121 #define init_sync_kiocb(x, filp)                        \ 
122    do {                                            \ 
123        struct task_struct *tsk = current;        \ 
124        (x)->ki_flags = 0;                        \ 
125        (x)->ki_users = 1;                        \ 
126        (x)->ki_key = KIOCB_SYNC_KEY;            \ 
127        (x)->ki_filp = (filp);                    \ 
128        (x)->ki_ctx = NULL;                      \ 
129        (x)->ki_cancel = NULL;                   \ 
130        (x)->ki_retry = NULL;                    \ 
131        (x)->ki_dtor = NULL;                     \ 
132        (x)->ki_obj.tsk = tsk;                   \ 
133        (x)->ki_user_data = 0;                   \ 
134        init_wait((&(x)->ki_wait));             \ 
135    } while (0)
The \texttt{__sock\_sendmsg()} function

This function packages the socket call related parameters into the \texttt{sock\_iocb} and then forwards them on to the AF\_layer handler specified in the \texttt{proto\_ops} structure linked to the \texttt{socket}.

\begin{verbatim}
581 static inline int __sock_sendmsg(struct kiocb *iocb, 
      struct socket *sock, 
      struct msghdr *msg, size_t size) 
582 {
583     struct sock_iocb *si = kiocb_to_siocb(iocb);
584     int err;
585
586     si->sock = sock;
587     si->scm = NULL;
588     si->msg = msg;
589     si->size = size;
591
We have seen these calls before. They access \texttt{hooks} provided by the Security Enhanced Linux (SEL) facility. At present they all just seem to return 0!

592     err = security_socket_sendmsg(sock, msg, size);
593     if (err)
594         return err;
595
This call maps to \texttt{inet\_sendmsg()}

596     return sock->ops->sendmsg(iocb, sock, msg, size);
597 }
\end{verbatim}
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.

```c
933 int security_socket_sendmsg(struct socket *sock,  
    struct msghdr *msg, int size)
934 {  
    return security_ops->socket_sendmsg(sock, msg, size);
935 }
```

The security_ops pointer is initially set to NULL.

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

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

```c
799 struct security_operations default_security_ops = {
    .name   = "default",
801    
};
```
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.

```c
/**
 * 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`.

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

15
The **security_fixup_ops** function

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

```c
812 void security_fixup_ops(struct security_operations *ops) {
814     set_to_cap_if_null(ops, ptrace_may_access);
815     set_to_cap_if_null(ops, ptrace_traceme);
816     set_to_cap_if_null(ops, capget);
849     set_to_cap_if_null(ops, socket_connect);
850     set_to_cap_if_null(ops, socket_listen);
851     set_to_cap_if_null(ops, socket_accept);
852     set_to_cap_if_null(ops, socket_post_accept);
853     set_to_cap_if_null(ops, socket_sendmsg);
854     set_to_cap_if_null(ops, socket_recvmsg);
}
```

The actual security functions

At present all the socket functions just return 0!

```c
571 static int cap_socket_sendmsg(struct socket *sock, 
     struct msghdr *msg, int size) 
572 { 
573     return 0; 
574 }
575
576 static int cap_socket_recvmsg(struct socket *sock, 
     struct msghdr *msg, 
     int size, int flags) 
578 { 
579     return 0; 
580 }
```
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.

*sccm_cookie* is defined in include/net/scm.h.

```c
15 struct scm_cookie
16 {
17    struct ucred creds;          /* Skb credentials */
18    struct scm_fp_list *fp;      /* Passed files */
19    unsigned long seq;          /* Connection seqno */
20 }; // scm_cookie
```

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.

```c
33 static __inline__ int scm_send(struct socket *sock,
34             struct msghdr *msg, struct scm_cookie *scm)
35 {
36    memset(scm, 0, sizeof(*scm));
37    scm->creds.uid = current->uid;
38    scm->creds.gid = current->gid;
39    scm->creds.pid = current->pid;
40    if (msg->msg_controllen <= 0)
41        return 0;
42    return __scm_send(sock, msg, scm);
43 }
```
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.

```c
658 int inet_sendmsg(struct kiocb *iocb, struct socket *sock,
       struct msghdr *msg,
       size_t size)
659 {
660    struct sock *sk = sock->sk;
661    /* We may need to bind the socket. */
662    if (!inet_sk(sk)->num && inet_autobind(sk))
663       return -EAGAIN;
664
665    return sk->sk_prot->sendmsg(iocb, sk, msg, size);
666 }
```

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

```c
667    return sk->sk_prot->sendmsg(iocb, sk, msg, size);
668 }
```
Data structures used by `udp_sendmsg`

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

```c
51 struct ipcm_cookie
52 {
53    u32 addr;
54    int oif;
55    struct ip_options *opt;
56 }
```

- `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.

```
93 struct ip_options {
94     __u32 faddr;     /* Saved first hop address */
95     unsigned char optlen;
96     unsigned char srr;
97     unsigned char rr;
98     unsigned char ts;
99     unsigned char is_setbyuser:1, /* Set by setsockopt? */
100    is_data:1, /* Options in __data, rather than skb */
101    is_strictroute:1, /* Strict source route */
102    srr_is_hit:1,   /* Packet dest addr was our one */
103    is_changed:1,  /* IP checksum more not valid */
104    rr_needaddr:1, /* Need to record addr of outgoing dev */
105    ts_needtime:1, /* Need to record timestamp */
106    ts_needaddr:1; /* Need to record addr of outgoing dev */
107     unsigned char router_alert;
108     unsigned char __pad1;
109     unsigned char __pad2;
110     unsigned char __data[0];
111 };  
112  
```
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

```c
55 struct udp_sock {
56 /* inet_sock has to be the first member */
57   struct inet_sock inet;
58   int pending;          /* Any pending frames? */
59   unsigned int corkflag; /* Cork is required */
60 #__u16 encap_type;    /* Is this an Encapsocket? */
61 /*
62 * Following member retains the infor to create a UDP header
63 * when the socket is uncorked.
64 */
65 #__u16 len;            /* total length of pending frames */
66 /*
67 * Fields specific to UDP-Lite.
68 */
69 __u16 pcslen;
70 __u16 pcrlen;
71 /* indicator bits used by pcflag: */
72 #define UDPLITE_BIT    0x1 /* set by udplite proto init */
73 #define UDPLITE_SEND_CC 0x2 /* set via udplite setsockopt */
74 #define UDPLITE_RECV_CC 0x4 /* set via udplite setsockopt */
75 #__u8 pcflag;        /* marks socket as UDP-Lite */
76        if > 0 */
77 /*
78 * For encapsulation sockets.
79 */
80   int (*encap_rcv)(struct sock *sk, struct sk_buff *skb);
81 }
```
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.*

```
137    struct {
138        unsigned int            flags;
139        unsigned int            fragsize;
140        struct ip_options       *opt;
141        struct dst_entry        *dst;
142        int    length; /* Total length of all frames */
143        __be32                  addr;
144        struct flowi            fl;
145    } 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.

```c
483 int udp_sendmsg(struct kiocb *iocb, struct sock *sk,
                      struct msghdr *msg,
                      size_t len)
484 {
485    struct inet_sock *inet = inet_sk(sk);
486    struct udp_sock *up = udp_sk(sk);
487    int ulen = len;
488    struct ipcm_cookie ipc;
489    struct rtable *rt = NULL;
490    int free = 0;
491    int connected = 0;
492    u32 daddr, faddr, saddr;
493    u16 dport;
494    u8  tos;
495    int err;
496
The corkreq flag will be set if and only if "corking" was previously specified via *setsockopt()* or the MSG_MORE flag was set.

497    int corkreq = up->corkflag || msg->msg_flags & MSG_MORE;
498```

23
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.

```
  1296    switch(optname) {
  1232    case UDP_CORK:
  1233       if (val != 0) {
  1234          up->corkflag = 1;
  1235       } else {
  1236          up->corkflag = 0;
  1237          lock_sock(sk);
  1238          udp_push_pending_frames(sk, up);
  1239          release_sock(sk);
  1240       }
```
The value of \textit{len} is checked first for validity. The use of unsigned short integer type for \textit{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.

\begin{verbatim}
    if (len > 0xFFFF)
        return -EMSGSIZE;
\end{verbatim}

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

\begin{verbatim}
    /*
     * Check the flags.
     */
    if (msg->msg_flags & MSG_OOB) /* Mirror BSD error message compatibility */
        return -EOPNOTSUPP;

    ipc.opt = NULL;
\end{verbatim}
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.

```c
511    if (up->pending) {
512        /*
513         * There are pending frames.
514         * The socket lock must be held while it's corked.
515         */
516        lock_sock(sk);
517        if (likely(up->pending)) {
518            if (unlikely(up->pending != AF_INET)) {
519                release_sock(sk);
520                return -EINVAL;
521            }
522            goto do_append_data;  <-- This is a big jump
523        }
524        release_sock(sk);
525    }
```
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.

```c
526    ulen += sizeof(struct udphdr);
527```

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

```
530     sk->sk_state == TCP_ESTABLISHED.
```

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

```c
540       daddr = usin->sin_addr.s_addr;
541       dport = usin->sin_port;
542       if (dport == 0)
543          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*.

```c
544   } else {
545       if (sk->sk_state != TCP_ESTABLISHED)
546          return -EDESTADDRREQ;
547       daddr = inet->daddr;
548       dport = inet->dport;
549       /* Open fast path for connected socket.
550          Route will not be used, if any options  are set.
551        */
552       connected = 1;
553   }
```
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 bound is temporarily held in the `ipc` for unknown reasons.

```c
554    ipc.addr = inet->saddr;
555
556    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.

```c
557    if (msg->msg_controllen) {
558       err = ip_cmsg_send(msg, &ipc);
559       if (err)
560          return err;
561       if (ipc.opt)
562          free = 1;
563       connected = 0;
564    }
```

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.

```c
565    if (!ipc.opt)
566        ipc.opt = inet->opt;
```
This section here is something of an oddity. Note that \textit{ipc.addr} was set to \textit{inet->saddr} above. Here it is set to the destination address after the source address is saved in \textit{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 \textit{daddr} is the value specified in the struct \textit{sockaddr_in} or if not \textit{sockaddr_in} was provided and the socket was connected, then \textit{daddr} was taken from \textit{inet->daddr}.

\begin{verbatim}
568  saddr = ipc.addr;
569  ipc.addr = faddr = daddr;
570
571  if (ipc.opt && ipc.opt->srr) {
572     if (!daddr)
573        return -EINVAL;
574     faddr = ipc.opt->faddr;
575     connected = 0;
576    }
\end{verbatim}

\textbf{Routing options}

The \textit{RT\_TOS macro} retrieves the low order 5 bits from the \textit{tos} field of the \textit{struct sock}. These will be 0 unless set by \textit{setsockopt()}.

\begin{verbatim}
577  tos = RT\_TOS(inet->tos);
\end{verbatim}

The RTO\_ONLINK bit forces the destination (or next hop in case of a \textit{strict} source route) to be reachable in a single hop.

\begin{verbatim}
578  if (sock_flag(sk, SOCK\_LOCALROUTE) ||
579      (msg->msg_flags & MSG\_DONTRoute) ||
580      (ipc.opt && ipc.opt->is_strictroute)) {
581     tos |= RTO\_ONLINK;
582     connected = 0;
583  }
\end{verbatim}
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.

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

585   if (MULTICAST(daddr)) {
586       if (!ipc.oif)
587          ipc.oif = inet->mc_index;
588       if (!saddr)
589          saddr = inet->mc_addr;
590       connected = 0;
591   }
```
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.

593    if (connected)
594       rt = (struct rtable*)sk_dst_check(sk, 0);
595
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.

596    if (rt == NULL) {
597       struct flowi fl = {.oif = ipc.oif,
598          .nl_u = { .ip4_u =
599              { .daddr = faddr,
600                .saddr = saddr,
601                .tos = tos } },
602          .proto = IPPROTO_UDP,
603          .uli_u = { .ports =
604              { .sport = inet->sport,
605                .dport = dport } } } );
606       err = ip_route_output_flow(&rt, &fl, sk, ! (msg->msg_flags&MSG_DONTWAIT));
607       if (err)
608          goto out;
609       err = -EACCES;
This appears to be checking to see if the broadcast attributes of the route and the \textit{struct sock} are mutually incompatible with respect to the \textit{broadcast} attribute.

\begin{verbatim}
err = -EACCES;
if ((rt->rt_flags & RTCF_BROADCAST) && !sock_flag(sk, SOCK_BROADCAST))
goto out;
\end{verbatim}

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

\begin{verbatim}
if (connected)
    sk_dst_set(sk, dst_clone(&rt->u.dst));
} //endif rt was NULL
\end{verbatim}

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

\begin{verbatim}
if (msg->msg_flags&MSG_CONFIRM)
goto do_confirm;
\end{verbatim}

\begin{verbatim}
back_from_confirm:
\end{verbatim}
**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.

```c
  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.

```c
  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()`.

```c
/*
 * 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 additional user data is added to the length maintained in the udp_sock.

```c
645 do_append_data:
646    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.

```c
647    err = ip_append_data(sk, ip_generic_getfrag,
648                   msg->msg_iov, ulen,
649                   sizeof(struct udphdr), &ipc, rt,
650                   corkreq ? msg->msg_flags|MSG_MORE :
651                       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.

```c
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.

```c
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.

```c
666 do_confirm:
667    dst_confirm(&rt->u.dst);
668    if (!(msg->msg_flags&MSG_PROBE) || len)
669        goto back_from_confirm;
670    err = 0;
671    goto out;
672 }
```
The `udp_push_pending_frames` function

This function has two missions:

- Fill in the UDP header
- Compute the checksum

```c
402 static int udp_push_pending_frames(struct sock *sk,
                                        struct udp_sock *up)
403 {
404    struct inet_sock *inet = inet_sk(sk);
405    struct flowi *fl = &inet->cork.fl;
406    struct sk_buff *skb;
407    struct udphdr *uh;
408    int err = 0;
409
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.

410 /* Grab the skb where UDP header space exists. */
411 if ((skb = skb_peek(&sk->sk_write_queue)) == NULL)
412     goto out;
413```

```c
```
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.

```c
414    /*
415     * Create a UDP header
416     */
417    uh = skb->h.uh;
418    uh->source = fl->fl_ip_sport;
419    uh->dest = fl->fl_ip_dport;
420    uh->len = htons(up->len);
421    uh->check = 0;
422
```

If checksumming is disabled, skip to the send code.

```c
423    if (sk->sk_no_check == UDP_CSUM_NOXMIT) {
424       skb->ip_summed = CHECKSUM_NONE;
425       goto send;
426    }
427```
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.

```c
428   if (skb_queue_len(&sk->sk_write_queue) == 1) {
429       /*
430        * Only one fragment on the socket.
431        */
432       if (skb->ip_summed == CHECKSUM_HW) {
433          skb->csum = offsetof(struct udphdr, check);
434          uh->check = ~csum_tcpudp_magic(fl->fl4_src,
435                  fl->fl4_dst,
436                  up->len, IPPROTO_UDP, 0);
437       } else {
438          skb->csum = csum_partial((char *)uh,
439                  sizeof(struct udphdr), skb->csum);
440          uh->check = csum_tcpudp_magic(fl->fl4_src,
441                  fl->fl4_dst,
442                  up->len, IPPROTO_UDP, skb->csum);
443          if (uh->check == 0)
444              uh->check = -1;
445       }
```
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_bufs
         * 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.

```
  469  send:
  470    err = ip_push_pending_frames(sk);
  471  out:
  472    up->len = 0;
  473    up->pending = 0;
  474    return err;
  475  }
```
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_buf* structure and pass it on the the *netfilter* layer for processing.

```
1188 /*
1189  * Combined all pending frags on the socket as one IP datagram
1190  * and push them out.
1191 */
1192 int ip_push_pending_frames(struct sock *sk)
1193 {
1194    struct sk_buff *skb, *tmp_skb;
1195    struct sk_buff **tail_skb;
1196    struct inet_sock *inet = inet_sk(sk);
1197    struct ip_options *opt = NULL;
1198    struct rtable *rt = inet->cork.rt;
1199    struct iphdr *iph;
1200    __be16 df = 0;
1201    __u8 ttl;
1202    int err = 0;
1203
```

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*.

```
1204    if ((skb = __skb_dequeue(&sk->sk_write_queue)) == NULL)
1205        goto out;
1206    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.

1208  /* move skb->data to ip header from ext header */
1209  if (skb->data < skb->nh.raw)
1210     __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.

1211  while ((tmp_skb = __skb_dequeue(&sk->sk_write_queue))
1212      != NULL) {
1213      __skb_pull(tmp_skb, skb->h.raw - skb->nh.raw);
1214      *tail_skb = tmp_skb;
1215      tail_skb = &tmp_skb->next;
1216      skb->len += tmp_skb->len;
1217      skb->data_len += tmp_skb->len;
1218      skb->truesize += tmp_skb->truesize;
1219      __sock_put(tmp_skb->sk);
1220      tmp_skb->destructor = NULL;
1221      tmp_skb->sk = NULL;
1221  }

45
Path MTU discovery and TTL processing

You should just set \texttt{df} to \texttt{htons(IP\_DF)}, and use \texttt{ip\_select\_ttl()} to initialize the \texttt{ttl} back when the socket was initially connected.

1222
1223 /* Unless user demanded real pmtu discovery (IP\_PMTUDISC\_DO),
1224  * we allow
1225  * to fragment the frame generated here. No matter, what
1226  * how transforms change size of the packet, it will come out.
1226 */
1227    if (inet->pmtudisc != IP\_PMTUDISC\_DO)
1228       skb->local_df = 1;
1229
1230 /* DF bit is set when we want to see DF on outgoing frames.
1231  * If local_df is set too, we still allow to fragment this
1232  * frame locally. */
1233    if (inet->pmtudisc == IP\_PMTUDISC\_DO ||
1234        (skb->len <= dst_mtu(&rt->u.dst) &&
1235         ip\_dont\_fragment(sk, &rt->u.dst)))
1236       df = htons(IP\_DF);
1237
1238    if (inet->cork.flags & IPCORK\_OPT)
1239       opt = inet->cork.opt;
1240
1241    if (rt->rt_type == RTN\_MULTICAST)
1242       ttl = inet->mc_ttl;
1243 else
1244       ttl = ip\_select\_ttl(inet, &rt->u.dst);
1245
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`.

```
1246   iph = (struct iphdr *) skb->data;
1247   iph->version = 4;
1248   iph->ihl = 5;
1249   if (opt) {
1250       iph->ihl += opt->optlen>>2;
1251       ip_options_build(skb, opt, inet->cork.addr, rt, 0);
1252   }
1253   iph->tos = inet->tos;
1254   iph->tot_len = htons(skb->len);
1255   iph->frag_off = df;
1256   ip_select_ident(iph, &rt->u.dst, sk);
1257   iph->ttl = ttl;
1258   iph->protocol = sk->sk_protocol;
1259   iph->saddr = rt->rt_src;
1260   iph->daddr = rt->rt_dst;
```

The `ip_send_check()` function is an inline function that computes the header checksum.

```
1261   ip_send_check(iph);
1262   skb->priority = sk->sk_priority;
1263   skb->dst = dst_clone(&rt->u.dst);
```

```c
88 /* Generate a checksum for an outgoing IP datagram. */
89 __inline__ void ip_send_check(struct iphdr *iph)
90 {
91     iph->check = 0;
92     iph->check = ip_fast_csum((unsigned char *)iph, iph->ihl);
93 }
```
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.

```c
1266 /* Netfilter gets whole the not fragmented skb. */
1267    err = NF_HOOK(PF_INET, NF_IP_LOCAL_OUT, skb, NULL,
1268                  skb->dst->dev, dst_output);
1269    if (err) {
1270       if (err > 0)
1271          err = inet->recverr ? net_xmit_errno(err) : 0;
1272       if (err)
1273          goto error;
1274    }
1275
1276    inet->cork.flags &= ~IPCORK_OPT;
1277    kfree(inet->cork.opt);
1278    inet->cork.opt = NULL;
1279    if (inet->cork.rt) {
1280       ip_rt_put(inet->cork.rt);
1281       inet->cork.rt = NULL;
1282    }
1283    return err;
1284
1285    error:
1286    IP_INC_STATS(IPSTATS_MIB_OUTDISCARDS);
1287    goto out;
1288  }
1289 }
```
The \textit{dst\_output\_function}

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

\begin{verbatim}
225 static inline int dst_output(struct sk_buff *skb) {
226    return skb->dst->output(skb);
228 }
\end{verbatim}
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 modularized 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()`.

```c
771 int ip_append_data(struct sock *sk,
772       int getfrag(void *from, char *to, int offset, int len,
773       int odd, struct sk_buff *skb),
774       void *from, int length, int transhdrlen,
775       struct ipcm_cookie *ipc, struct rtable *rt,
776       unsigned int flags)
777 {
778    struct inet_sock *inet = inet_sk(sk);
779    struct sk_buff *skb;
780
781    struct ip_options *opt = NULL;
782    int hh_len;
783    int exthdrlen;
784    int mtu;
785    int copy;
786    int err;
787    int offset = 0;
788    unsigned int maxfraglen, fragheaderlen;
789    int csummode = CHECKSUM_NONE;
790
791    if (flags & MSG_PROBE)
792       return 0;
    793```
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`.

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

Header options are copied to the cork here.

```c
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.

```c
dst_hold(&rt->u.dst);
inetcork.fragsize = mtu = dst_mtu(rt->u.dst.path);
inetcork.rt = rt;
inetcork.length = 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 fragment.

```c
819    } else {  // write queue not empty
820       rt = inet->cork.rt;
821       if (inet->cork.flags & IPCORK_OPT)
822          opt = inet->cork.opt;
823
824       transhdrlen = 0;
825       exthdrlen = 0;
826       mtu = inet->cork.fragsize;
827    }
```

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.

```c
828    hh_len = LL_RESERVED_SPACE(rt->u.dst.dev);
829
830    fragheaderlen = sizeof(struct iphdr) +
831     (opt ? opt->optlen : 0);
832    maxfraglen = ((mtu - fragheaderlen) & ~7) +
833         fragheaderlen;
834
835    if (inet->cork.length+length > 0xFFFF - fragheaderlen) {
836       ip_local_error(sk, EMSGSIZE, rt->rt_dst, inet->dport,
837            mtu-exthdrlen);
838       return -EMSGSIZE;
839    }
```
This section is trying to take advantage of UDP checksum offload if it exts.

837
838  /*
839  * transhdrlen > 0 means that this is the first fragment
840  * and we wish
841  * it won't be fragmented in the future.
842  */

843  if (transhdrlen &&
844      length + fragheaderlen <= mtu &&
845      rt->u.dst.dev->features & NETIF_F_ALL_CSUM &&
846      !exthdrlen)
847      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.

848  inet->cork.length += length;
849  if (((length > mtu) && (sk->sk_protocol == IPPROTO_UDP))
850     && (rt->u.dst.dev->features & NETIF_F_UFO)) {
851      err = ip_ufo_append_data(sk, getfrag, from,
852                              length, hh_len,
853                              fragheaderlen, transhdrlen, mtu,
854                              flags);
855      if (err)
856         goto error;
857      return 0;
858  }
So what's going on in the rest of this function.... I give up!

860 /* So, what's going on in the loop below?
861 *
862 * We use calc fragment length to generate chained skb,
863 * each of segments is IP fragment ready for sending to
864 * network after
865 * adding appropriate IP header.
866 */
867 if ((skb = skb_peek_tail(&sk->sk_write_queue)) == NULL)
868     goto alloc_new_skb;
869
870 while (length > 0) {
871    /* Check if the remaining data fits into current packet. */
872    copy = mtu - skb->len;
873    if (copy < length)
874        copy = maxfraglen - skb->len;
875    if (copy <= 0) {
876        char *data;
877        unsigned int datalen;
878        unsigned int fraglen;
879        unsigned int fraggap;
880        unsigned int alloclen;
881        struct sk_buff *skb_prev;
Buffer allocation

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

882 \texttt{alloc_new_skb:}
883     skb_prev = skb;
884     if (skb_prev)
885         fraggap = skb_prev->len - maxfraglen;
886     else
887         fraggap = 0;
888
889     /*
890     * If remaining data exceeds the mtu,
891     * we know we need more fragment(s).
892     */
893     datalen = length + fraggap;
894     if (datalen > mtu - fragheaderlen)
895         datalen = maxfraglen - fragheaderlen;
896     fraglen = datalen + fragheaderlen;
897
898     if ((flags & MSG_MORE) &&
899         !(rt->u.dst.dev->features&NETIF_F_SG))
900         allocalen = mtu;
901     else
902         allocalen = datalen + fragheaderlen;
903
904     /* The last fragment gets additional space at tail.
905     * Note, with MSG_MORE we overalloc on fragments,
906     * because we have no idea what fragment will be
907     * the last.
908     */
909     if (datalen == length + fraggap)
910         allocalen += 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.

```c
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 *hdrlen are 0.

```c
/*
 * 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()

```c
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 {
    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];
        }
    }
}
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.

```c
677 int
678 ip_generic_getfrag(void *from, char *to, int offset,
679     int len, int odd, struct sk_buff *skb)
680 {
681     struct iovec *iov = from;
682     if (skb->ip_summed == CHECKSUM_HW) {
683         if (memcpy_fromiovecend(to, iov, offset, len) < 0)
684             return -EFAULT;
685     } else {
686         unsigned int csum = 0;
687         if (csum_partial_copy_fromiovecend(to, iov,
688             offset, len, &csum) < 0)
689             return -EFAULT;
690     }
691     skb->csum = csum_block_add(skb->csum, csum, odd);
692     return 0;
693 }
```