Networking Implementation Notes
4.3BSD Edition

Samuel J. Leffler, William N. Joy, Robert S. Fabry, and Michael J. Karels

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

ABSTRACT

This report describes the internal structure of the networking facilities developed for the 4.3BSD version of the UNIX* operating system for the VAX†. These facilities are based on several central abstractions which structure the external (user) view of network communication as well as the internal (system) implementation.

The report documents the internal structure of the networking system. The ‘‘Berkeley Software Architecture Manual, 4.3BSD Edition’’ (PS1:6) provides a description of the user interface to the networking facilities.

Revised March 7, 1989

* UNIX is a trademark of Bell Laboratories.

† DEC, VAX, DECnet, and UNIBUS are trademarks of Digital Equipment Corporation.

TABLE OF CONTENTS

1. Introduction

This report describes the internal structure of facilities added to the 4.2BSD version of the UNIX operating system for the VAX, as modified in the 4.3BSD release. The system facilities provide a uniform user interface to networking within UNIX. In addition, the implementation introduces a structure for network communications which may be used by system implementors in adding new networking facilities. The internal structure is not visible to the user, rather it is intended to aid implementors of communication protocols and network services by providing a framework which promotes code sharing and minimizes implementation effort.

The reader is expected to be familiar with the C programming language and system interface, as described in the *Berkeley Software Architecture Manual, 4.3BSD Edition* [Joy86]. Basic understanding of network communication concepts is assumed; where required any additional ideas are introduced.

The remainder of this document provides a description of the system internals, avoiding, when possible, those portions which are utilized only by the interprocess communication facilities.

2. Overview

If we consider the International Standards Organization's (ISO) Open System Interconnection (OSI) model of network communication [ISO81] [Zimmermann80], the networking facilities described here correspond to a portion of the session layer (layer 3) and all of the transport and network layers (layers 2 and 1, respectively).

The network layer provides possibly imperfect data transport services with minimal addressing structure. Addressing at this level is normally host to host, with implicit or explicit routing optionally supported by the communicating agents.

At the transport layer the notions of reliable transfer, data sequencing, flow control, and service addressing are normally included. Reliability is usually managed by explicit acknowledgement of data delivered. Failure to acknowledge a transfer results in retransmission of the data. Sequencing may be handled by tagging each message handed to the network layer by a *sequence number* and maintaining state at the endpoints of communication to utilize received sequence numbers in reordering data which arrives out of order.

The session layer facilities may provide forms of addressing which are mapped into formats required by the transport layer, service authentication and client authentication, etc. Various systems also provide services such as data encryption and address and protocol translation.

The following sections begin by describing some of the common data structures and utility routines, then examine the internal layering. The contents of each layer and its interface are considered. Certain of the interfaces are protocol implementation specific. For these cases examples have been drawn from the Internet [Cerf78] protocol family. Later sections cover routing issues, the design of the raw socket interface and other miscellaneous topics.

3. Goals

The networking system was designed with the goal of supporting multiple *protocol families* and addressing styles. This required information to be "hidden" in common data structures which could be manipulated by all the pieces of the system, but which required interpretation only by the protocols which "controlled" it. The system described here attempts to minimize the use of shared data structures to those kept by a suite of protocols (a *protocol family*), and those used for rendezvous between "synchronous" and "asynchronous" portions of the system (e.g. queues of data packets are filled at interrupt time and emptied based on user requests).

A major goal of the system was to provide a framework within which new protocols and hardware could be easily be supported. To this end, a great deal of effort has been extended to create utility routines which hide many of the more complex and/or hardware dependent chores of networking. Later sections describe the utility routines and the underlying data structures they manipulate.

4. Internal address representation

Common to all portions of the system are two data structures. These structures are used to represent addresses and various data objects. Addresses, internally are described by the *sockaddr* structure,

```
struct sockaddr {
        short       sa_family;          /* data format identifier */
        char        sa_data[14];        /* address */
};
```

All addresses belong to one or more *address families* which define their format and interpretation. The *sa_family* field indicates the address family to which the address belongs, and the *sa_data* field contains the actual data value. The size of the data field, 14 bytes, was selected based on a study of current address formats.* Specific address formats use private structure definitions that define the format of the data field. The system interface supports larger address structures, although address-family-independent support facilities, for example routing and raw socket interfaces, provide only 14 bytes for address storage. Protocols that do not use those facilities (e.g, the current Unix domain) may use larger data areas.

5. Memory management

A single mechanism is used for data storage: memory buffers, or *mbuf*'s. An mbuf is a structure of the form:

```
struct mbuf {
        struct      mbuf *m_next;       /* next buffer in chain */
        u_long      m_off;              /* offset of data */
        short       m_len;              /* amount of data in this mbuf */
        short       m_type;             /* mbuf type (accounting) */
        u_char      m_dat[MLEN];        /* data storage */
        struct      mbuf *m_act;        /* link in higher-level mbuf list */
};
```

The *m_next* field is used to chain mbufs together on linked lists, while the *m_act* field allows lists of mbuf chains to be accumulated. By convention, the mbufs common to a single object (for example, a packet) are chained together with the *m_next* field, while groups of objects are linked via the *m_act* field (possibly when in a queue).

Each mbuf has a small data area for storing information, *m_dat*. The *m_len* field indicates the amount of data, while the *m_off* field is an offset to the beginning of the data from the base of the mbuf. Thus, for example, the macro *mtod*, which converts a pointer to an mbuf to a pointer to the data stored in the mbuf, has the form

```
#define   mtod(x,t)            ((t)((int)(x) + (x)->m_off))
```

(note the *t* parameter, a C type cast, which is used to cast the resultant pointer for proper assignment).

In addition to storing data directly in the mbuf's data area, data of page size may be also be stored in a separate area of memory. The mbuf utility routines maintain a pool of pages for this purpose and manipulate a private page map for such pages. An mbuf with an external data area may be recognized by the larger offset to the data area; this is formalized by the macro M_HASCL(*m*), which is true if the mbuf whose address is *m* has an external page cluster. An array of reference counts on pages is also maintained so that copies of pages may be made without core to core copying (copies are created simply by duplicating the reference to the data and incrementing the associated reference counts for the pages). Separate data pages are currently used only when copying data from a user process into the kernel, and when bringing data in at the hardware level. Routines which manipulate mbufs are not normally aware whether data is stored directly in the mbuf data array, or if it is kept in separate pages.

The following may be used to allocate and free mbufs:

* Later versions of the system may support variable length addresses.

m = m_get(wait, type);
MGET(m, wait, type);

The subroutine *m_get* and the macro *MGET* each allocate an mbuf, placing its address in *m*. The argument *wait* is either M_WAIT or M_DONTWAIT according to whether allocation should block or fail if no mbuf is available. The *type* is one of the predefined mbuf types for use in accounting of mbuf allocation.

MCLGET(m);

This macro attempts to allocate an mbuf page cluster to associate with the mbuf *m*. If successful, the length of the mbuf is set to CLSIZE, the size of the page cluster.

n = m_free(m);
MFREE(m,n);

The routine *m_free* and the macro *MFREE* each free a single mbuf, *m*, and any associated external storage area, placing a pointer to its successor in the chain it heads, if any, in *n*.

m_freem(m);

This routine frees an mbuf chain headed by *m*.

The following utility routines are available for manipulating mbuf chains:

m = m_copy(m0, off, len);

The *m_copy* routine create a copy of all, or part, of a list of the mbufs in *m0*. *Len* bytes of data, starting *off* bytes from the front of the chain, are copied. Where possible, reference counts on pages are used instead of core to core copies. The original mbuf chain must have at least *off* + *len* bytes of data. If *len* is specified as M_COPYALL, all the data present, offset as before, is copied.

m_cat(m, n);

The mbuf chain, *n*, is appended to the end of *m*. Where possible, compaction is performed.

m_adj(m, diff);

The mbuf chain, *m* is adjusted in size by *diff* bytes. If *diff* is non-negative, *diff* bytes are shaved off the front of the mbuf chain. If *diff* is negative, the alteration is performed from back to front. No space is reclaimed in this operation; alterations are accomplished by changing the *m_len* and *m_off* fields of mbufs.

m = m_pullup(m0, size);

After a successful call to *m_pullup*, the mbuf at the head of the returned list, *m*, is guaranteed to have at least *size* bytes of data in contiguous memory within the data area of the mbuf (allowing access via a pointer, obtained using the *mtod* macro, and allowing the mbuf to be located from a pointer to the data area using *dtom*, defined below). If the original data was less than *size* bytes long, *len* was greater than the size of an mbuf data area (112 bytes), or required resources were unavailable, *m* is 0 and the original mbuf chain is deallocated.

This routine is particularly useful when verifying packet header lengths on reception. For example, if a packet is received and only 8 of the necessary 16 bytes required for a valid packet header are present at the head of the list of mbufs representing the packet, the remaining 8 bytes may be ‘‘pulled up’’ with a single *m_pullup* call. If the call fails the invalid packet will have been discarded.

By insuring that mbufs always reside on 128 byte boundaries, it is always possible to locate the mbuf associated with a data area by masking off the low bits of the virtual address. This allows modules to store data structures in mbufs and pass them around without concern for locating the original mbuf when it comes time to free the structure. Note that this works only with objects stored in the internal data buffer of the mbuf. The *dtom* macro is used to convert a pointer into an mbuf’s data area to a pointer to the mbuf,

```
#define	dtom(x)	((struct mbuf *)((int)x & ~(MSIZE-1)))
```

Mbufs are used for dynamically allocated data structures such as sockets as well as memory allocated for packets and headers. Statistics are maintained on mbuf usage and can be viewed by users using the *netstat* (1) program.

6. Internal layering

The internal structure of the network system is divided into three layers. These layers correspond to the services provided by the socket abstraction, those provided by the communication protocols, and those provided by the hardware interfaces. The communication protocols are normally layered into two or more individual cooperating layers, though they are collectively viewed in the system as one layer providing services supportive of the appropriate socket abstraction.

The following sections describe the properties of each layer in the system and the interfaces to which each must conform.

6.1. Socket layer

The socket layer deals with the interprocess communication facilities provided by the system. A socket is a bidirectional endpoint of communication which is ‘‘typed’’ by the semantics of communication it supports. The system calls described in the *Berkeley Software Architecture Manual* [Joy86] are used to manipulate sockets.

A socket consists of the following data structure:

```
struct socket {
        short       so_type;             /* generic type */
        short       so_options;          /* from socket call */
        short       so_linger;           /* time to linger while closing */
        short       so_state;            /* internal state flags */
        caddr_t     so_pcb;              /* protocol control block */
        struct      protosw *so_proto;   /* protocol handle */
        struct      socket *so_head;     /* back pointer to accept socket */
        struct      socket *so_q0;       /* queue of partial connections */
        short       so_q0len;            /* partials on so_q0 */
        struct      socket *so_q;        /* queue of incoming connections */
        short       so_qlen;             /* number of connections on so_q */
        short       so_qlimit;           /* max number queued connections */
        struct      sockbuf so_rcv;      /* receive queue */
        struct      sockbuf so_snd;      /* send queue */
        short       so_timeo;            /* connection timeout */
        u_short     so_error;            /* error affecting connection */
        u_short     so_oobmark;          /* chars to oob mark */
        short       so_pgrp;             /* pgrp for signals */
};
```

Each socket contains two data queues, *so_rcv* and *so_snd*, and a pointer to routines which provide supporting services. The type of the socket, *so_type* is defined at socket creation time and used in selecting those services which are appropriate to support it. The supporting protocol is selected at socket creation time and recorded in the socket data structure for later use. Protocols are defined by a table of procedures, the *protosw* structure, which will be described in detail later. A pointer to a protocol-specific data structure, the ‘‘protocol control block,’’ is also present in the socket structure. Protocols control this data structure, which normally includes a back pointer to the parent socket structure to allow easy lookup when returning information to a user (for example, placing an error number in the *so_error* field). The other entries in the socket structure are used in queuing connection requests, validating user requests, storing socket characteristics (e.g. options supplied at the time a socket is created), and maintaining a socket's state.

Processes ‘‘rendezvous at a socket’’ in many instances. For instance, when a process wishes to extract data from a socket's receive queue and it is empty, or lacks sufficient data to satisfy the request, the process blocks, supplying the address of the receive queue as a ‘‘wait channel’ to be used in notification. When data arrives for the process and is placed in the socket's queue, the blocked process is identified by the fact it is waiting ‘‘on the queue.’’

6.1.1. Socket state

A socket's state is defined from the following:

```
#define SS_NOFDREF          0x001  /* no file table ref any more */
#define SS_ISCONNECTED      0x002  /* socket connected to a peer */
#define SS_ISCONNECTING     0x004  /* in process of connecting to peer */
#define SS_ISDISCONNECTING  0x008  /* in process of disconnecting */
#define SS_CANTSENDMORE     0x010  /* can't send more data to peer */
#define SS_CANTRCVMORE      0x020  /* can't receive more data from peer */
#define SS_RCVATMARK        0x040  /* at mark on input */

#define SS_PRIV             0x080  /* privileged */
#define SS_NBIO             0x100  /* non-blocking ops */
#define SS_ASYNC            0x200  /* async i/o notify */
```

The state of a socket is manipulated both by the protocols and the user (through system calls). When a socket is created, the state is defined based on the type of socket. It may change as control actions are performed, for example connection establishment. It may also change according to the type of input/output the user wishes to perform, as indicated by options set with *fcntl*. ''Non-blocking'' I/O implies that a process should never be blocked to await resources. Instead, any call which would block returns prematurely with the error EWOULDBLOCK, or the service request may be partially fulfilled, e.g. a request for more data than is present.

If a process requested ''asynchronous'' notification of events related to the socket, the SIGIO signal is posted to the process when such events occur. An event is a change in the socket's state; examples of such occurrences are: space becoming available in the send queue, new data available in the receive queue, connection establishment or disestablishment, etc.

A socket may be marked ''privileged'' if it was created by the super-user. Only privileged sockets may bind addresses in privileged portions of an address space or use ''raw'' sockets to access lower levels of the network.

6.1.2. Socket data queues

A socket's data queue contains a pointer to the data stored in the queue and other entries related to the management of the data. The following structure defines a data queue:

```
struct sockbuf {
        u_short   sb_cc;          /* actual chars in buffer */
        u_short   sb_hiwat;       /* max actual char count */
        u_short   sb_mbcnt;       /* chars of mbufs used */
        u_short   sb_mbmax;       /* max chars of mbufs to use */
        u_short   sb_lowat;       /* low water mark */
        short     sb_timeo;       /* timeout */
        struct    mbuf *sb_mb;    /* the mbuf chain */
        struct    proc *sb_sel;   /* process selecting read/write */
        short     sb_flags;       /* flags, see below */
};
```

Data is stored in a queue as a chain of mbufs. The actual count of data characters as well as high and low water marks are used by the protocols in controlling the flow of data. The amount of buffer space (characters of mbufs and associated data pages) is also recorded along with the limit on buffer allocation. The socket routines cooperate in implementing the flow control policy by blocking a process when it requests to send data and the high water mark has been reached, or when it requests to receive data and less than the low water mark is present (assuming non-blocking I/O has not been specified).*

* The low-water mark is always presumed to be 0 in the current implementation.

When a socket is created, the supporting protocol ‘‘reserves’’ space for the send and receive queues of the socket. The limit on buffer allocation is set somewhat higher than the limit on data characters to account for the granularity of buffer allocation. The actual storage associated with a socket queue may fluctuate during a socket’s lifetime, but it is assumed that this reservation will always allow a protocol to acquire enough memory to satisfy the high water marks.

The timeout and select values are manipulated by the socket routines in implementing various portions of the interprocess communications facilities and will not be described here.

Data queued at a socket is stored in one of two styles. Stream-oriented sockets queue data with no addresses, headers or record boundaries. The data are in mbufs linked through the *m_next* field. Buffers containing access rights may be present within the chain if the underlying protocol supports passage of access rights. Record-oriented sockets, including datagram sockets, queue data as a list of packets; the sections of packets are distinguished by the types of the mbufs containing them. The mbufs which comprise a record are linked through the *m_next* field; records are linked from the *m_act* field of the first mbuf of one packet to the first mbuf of the next. Each packet begins with an mbuf containing the ‘‘from’’ address if the protocol provides it, then any buffers containing access rights, and finally any buffers containing data. If a record contains no data, no data buffers are required unless neither address nor access rights are present.

A socket queue has a number of flags used in synchronizing access to the data and in acquiring resources:

```
#define  SB_LOCK   0x01  /* lock on data queue (so_rcv only) */
#define  SB_WANT   0x02  /* someone is waiting to lock */
#define  SB_WAIT   0x04  /* someone is waiting for data/space */
#define  SB_SEL    0x08  /* buffer is selected */
#define  SB_COLL   0x10  /* collision selecting */
```

The last two flags are manipulated by the system in implementing the select mechanism.

6.1.3. Socket connection queuing

In dealing with connection oriented sockets (e.g. SOCK_STREAM) the two ends are considered distinct. One end is termed *active*, and generates connection requests. The other end is called *passive* and accepts connection requests.

From the passive side, a socket is marked with SO_ACCEPTCONN when a *listen* call is made, creating two queues of sockets: *so_q0* for connections in progress and *so_q* for connections already made and awaiting user acceptance. As a protocol is preparing incoming connections, it creates a socket structure queued on *so_q0* by calling the routine *sonewconn*(). When the connection is established, the socket structure is then transferred to *so_q*, making it available for an *accept*.

If an SO_ACCEPTCONN socket is closed with sockets on either *so_q0* or *so_q*, these sockets are dropped, with notification to the peers as appropriate.

6.2. Protocol layer(s)

Each socket is created in a communications domain, which usually implies both an addressing structure (address family) and a set of protocols which implement various socket types within the domain (protocol family). Each domain is defined by the following structure:

```
struct   domain {
         int     dom_family;            /* PF_xxx */
         char    *dom_name;
         int     (*dom_init)();         /* initialize domain data structures */
         int     (*dom_externalize)();  /* externalize access rights */
         int     (*dom_dispose)();      /* dispose of internalized rights */
         struct  protosw *dom_protosw, *dom_protoswNPROTOSW;
         struct  domain *dom_next;
};
```

At boot time, each domain configured into the kernel is added to a linked list of domain. The initialization procedure of each domain is then called. After that time, the domain structure is used to locate protocols within the protocol family. It may also contain procedure references for externalization of access rights at the receiving socket and the disposal of access rights that are not received.

Protocols are described by a set of entry points and certain socket-visible characteristics, some of which are used in deciding which socket type(s) they may support.

An entry in the ‘‘protocol switch’’ table exists for each protocol module configured into the system. It has the following form:

```
struct protosw {
        short   pr_type;                /* socket type used for */
        struct  domain *pr_domain;      /* domain protocol a member of */
        short   pr_protocol;            /* protocol number */
        short   pr_flags;               /* socket visible attributes */
/* protocol-protocol hooks */
        int     (*pr_input)();          /* input to protocol (from below) */
        int     (*pr_output)();         /* output to protocol (from above) */
        int     (*pr_ctlinput)();       /* control input (from below) */
        int     (*pr_ctloutput)();      /* control output (from above) */
/* user-protocol hook */
        int     (*pr_usrreq)();         /* user request */
/* utility hooks */
        int     (*pr_init)();           /* initialization routine */
        int     (*pr_fasttimo)();       /* fast timeout (200ms) */
        int     (*pr_slowtimo)();       /* slow timeout (500ms) */
        int     (*pr_drain)();          /* flush any excess space possible */
};
```

A protocol is called through the *pr_init* entry before any other. Thereafter it is called every 200 milliseconds through the *pr_fasttimo* entry and every 500 milliseconds through the *pr_slowtimo* for timer based actions. The system will call the *pr_drain* entry if it is low on space and this should throw away any non-critical data.

Protocols pass data between themselves as chains of mbufs using the *pr_input* and *pr_output* routines. *Pr_input* passes data up (towards the user) and *pr_output* passes it down (towards the network); control information passes up and down on *pr_ctlinput* and *pr_ctloutput*. The protocol is responsible for the space occupied by any of the arguments to these entries and must either pass it onward or dispose of it. (On output, the lowest level reached must free buffers storing the arguments; on input, the highest level is responsible for freeing buffers.)

The *pr_usrreq* routine interfaces protocols to the socket code and is described below.

The *pr_flags* field is constructed from the following values:

```
#define PR_ATOMIC          0x01    /* exchange atomic messages only */
#define PR_ADDR            0x02    /* addresses given with messages */
#define PR_CONNREQUIRED    0x04    /* connection required by protocol */
#define PR_WANTRCVD        0x08    /* want PRU_RCVD calls */
#define PR_RIGHTS          0x10    /* passes capabilities */
```

Protocols which are connection-based specify the PR_CONNREQUIRED flag so that the socket routines will never attempt to send data before a connection has been established. If the PR_WANTRCVD flag is set, the socket routines will notify the protocol when the user has removed data from the socket’s receive queue. This allows the protocol to implement acknowledgement on user receipt, and also update windowing information based on the amount of space available in the receive queue. The PR_ADDR field indicates that any data placed in the socket’s receive queue will be preceded by the address of the sender. The PR_ATOMIC flag specifies that each *user* request to send data must be performed in a single *protocol* send request; it is the protocol’s responsibility to maintain record boundaries on data to be sent. The

PR_RIGHTS flag indicates that the protocol supports the passing of capabilities; this is currently used only by the protocols in the UNIX protocol family.

When a socket is created, the socket routines scan the protocol table for the domain looking for an appropriate protocol to support the type of socket being created. The *pr_type* field contains one of the possible socket types (e.g. SOCK_STREAM), while the *pr_domain* is a back pointer to the domain structure. The *pr_protocol* field contains the protocol number of the protocol, normally a well-known value.

6.3. Network-interface layer

Each network-interface configured into a system defines a path through which packets may be sent and received. Normally a hardware device is associated with this interface, though there is no requirement for this (for example, all systems have a software ‘‘loopback’’ interface used for debugging and performance analysis). In addition to manipulating the hardware device, an interface module is responsible for encapsulation and decapsulation of any link-layer header information required to deliver a message to its destination. The selection of which interface to use in delivering packets is a routing decision carried out at a higher level than the network-interface layer. An interface may have addresses in one or more address families. The address is set at boot time using an *ioctl* on a socket in the appropriate domain; this operation is implemented by the protocol family, after verifying the operation through the device *ioctl* entry.

An interface is defined by the following structure,

```
struct ifnet {
        char    *if_name;               /* name, e.g. ‘‘en’’ or ‘‘lo’’ */
        short   if_unit;                /* sub-unit for lower level driver */
        short   if_mtu;                 /* maximum transmission unit */
        short   if_flags;               /* up/down, broadcast, etc. */
        short   if_timer;               /* time ’til if_watchdog called */
        struct  ifaddr *if_addrlist;    /* list of addresses of interface */
        struct  ifqueue if_snd;         /* output queue */
        int     (*if_init)();           /* init routine */
        int     (*if_output)();         /* output routine */
        int     (*if_ioctl)();          /* ioctl routine */
        int     (*if_reset)();          /* bus reset routine */
        int     (*if_watchdog)();       /* timer routine */
        int     if_ipackets;            /* packets received on interface */
        int     if_ierrors;             /* input errors on interface */
        int     if_opackets;            /* packets sent on interface */
        int     if_oerrors;             /* output errors on interface */
        int     if_collisions;          /* collisions on csma interfaces */
        struct  ifnet *if_next;
};
```

Each interface address has the following form:

```
struct ifaddr {
        struct  sockaddr ifa_addr;      /* address of interface */
        union {
                struct  sockaddr ifu_broadaddr;
                struct  sockaddr ifu_dstaddr;
        } ifa_ifu;
        struct  ifnet *ifa_ifp;         /* back-pointer to interface */
        struct  ifaddr *ifa_next;       /* next address for interface */
};
#define ifa_broadaddr   ifa_ifu.ifu_broadaddr   /* broadcast address */
#define ifa_dstaddr     ifa_ifu.ifu_dstaddr     /* other end of p-to-p link */
```

The protocol generally maintains this structure as part of a larger structure containing additional information concerning the address.

Each interface has a send queue and routines used for initialization, *if_init*, and output, *if_output*. If the interface resides on a system bus, the routine *if_reset* will be called after a bus reset has been performed. An interface may also specify a timer routine, *if_watchdog*; if *if_timer* is non-zero, it is decremented once per second until it reaches zero, at which time the watchdog routine is called.

The state of an interface and certain characteristics are stored in the *if_flags* field. The following values are possible:

```
#define  IFF_UP            0x1    /* interface is up */
#define  IFF_BROADCAST     0x2    /* broadcast is possible */
#define  IFF_DEBUG         0x4    /* turn on debugging */
#define  IFF_LOOPBACK      0x8    /* is a loopback net */
#define  IFF_POINTOPOINT   0x10   /* interface is point-to-point link */
#define  IFF_NOTRAILERS    0x20   /* avoid use of trailers */
#define  IFF_RUNNING       0x40   /* resources allocated */
#define  IFF_NOARP         0x80   /* no address resolution protocol */
```

If the interface is connected to a network which supports transmission of *broadcast* packets, the IFF_BROADCAST flag will be set and the *ifa_broadaddr* field will contain the address to be used in sending or accepting a broadcast packet. If the interface is associated with a point-to-point hardware link (for example, a DEC DMR-11), the IFF_POINTOPOINT flag will be set and *ifa_dstaddr* will contain the address of the host on the other side of the connection. These addresses and the local address of the interface, *if_addr*, are used in filtering incoming packets. The interface sets IFF_RUNNING after it has allocated system resources and posted an initial read on the device it manages. This state bit is used to avoid multiple allocation requests when an interface's address is changed. The IFF_NOTRAILERS flag indicates the interface should refrain from using a *trailer* encapsulation on outgoing packets, or (where per-host negotiation of trailers is possible) that trailer encapsulations should not be requested; *trailer* protocols are described in section 14. The IFF_NOARP flag indicates the interface should not use an "address resolution protocol" in mapping internetwork addresses to local network addresses.

Various statistics are also stored in the interface structure. These may be viewed by users using the *netstat*(1) program.

The interface address and flags may be set with the SIOCSIFADDR and SIOCSIFFLAGS *ioctl*s. SIOCSIFADDR is used initially to define each interface's address; SIOGSIFFLAGS can be used to mark an interface down and perform site-specific configuration. The destination address of a point-to-point link is set with SIOCSIFDSTADDR. Corresponding operations exist to read each value. Protocol families may also support operations to set and read the broadcast address. In addition, the SIOCGIFCONF *ioctl* retrieves a list of interface names and addresses for all interfaces and protocols on the host.

6.3.1. UNIBUS interfaces

All hardware related interfaces currently reside on the UNIBUS. Consequently a common set of utility routines for dealing with the UNIBUS has been developed. Each UNIBUS interface utilizes a structure of the following form:

```
struct  ifubinfo {
        short     iff_uban;                  /* uba number */
        short     iff_hlen;                  /* local net header length */
        struct    uba_regs *iff_uba;         /* uba regs, in vm */
        short     iff_flags;                 /* used during uballoc's */
};
```

Additional structures are associated with each receive and transmit buffer, normally one each per interface; for read,

```
struct  ifrw {
        caddr_t   ifrw_addr;                      /* virt addr of header */
        short     ifrw_bdp;                       /* unibus bdp */
        short     ifrw_flags;                     /* type, etc. */
#define IFRW_W    0x01                            /* is a transmit buffer */
        int       ifrw_info;                      /* value from ubaalloc */
        int       ifrw_proto;                     /* map register prototype */
        struct    pte *ifrw_mr;                   /* base of map registers */
};
```

and for write,

```
struct  ifxmt {
        struct    ifrw ifrw;
        caddr_t   ifw_base;                       /* virt addr of buffer */
        struct    pte ifw_wmap[IF_MAXNUBAMR];     /* base pages for output */
        struct    mbuf *ifw_xtofree;              /* pages being dma'd out */
        short     ifw_xswapd;                     /* mask of clusters swapped */
        short     ifw_nmr;                        /* number of entries in wmap */
};
#define ifw_addr  ifrw.ifrw_addr
#define ifw_bdp   ifrw.ifrw_bdp
#define ifw_flags ifrw.ifrw_flags
#define ifw_info  ifrw.ifrw_info
#define ifw_proto ifrw.ifrw_proto
#define ifw_mr    ifrw.ifrw_mr
```

One of each of these structures is conveniently packaged for interfaces with single buffers for each direction, as follows:

```
struct  ifuba {
        struct    ifubinfo ifu_info;
        struct    ifrw ifu_r;
        struct    ifxmt ifu_xmt;
};
#define ifu_uban      ifu_info.iff_uban
#define ifu_hlen      ifu_info.iff_hlen
#define ifu_uba       ifu_info.iff_uba
#define ifu_flags     ifu_info.iff_flags
#define ifu_w         ifu_xmt.ifrw
#define ifu_xtofree   ifu_xmt.ifw_xtofree
```

The *if_ubinfo* structure contains the general information needed to characterize the I/O-mapped buffers for the device. In addition, there is a structure describing each buffer, including UNIBUS resources held by the interface. Sufficient memory pages and bus map registers are allocated to each buffer upon initialization according to the maximum packet size and header length. The kernel virtual address of the buffer is held in *ifrw_addr*, and the map registers begin at *ifrw_mr*. UNIBUS map register *ifrw_mr*[−1] maps the local network header ending on a page boundary. UNIBUS data paths are reserved for read and for write, given by *ifrw_bdp*. The prototype of the map registers for read and for write is saved in *ifrw_proto*.

When write transfers are not at least half-full pages on page boundaries, the data are just copied into the pages mapped on the UNIBUS and the transfer is started. If a write transfer is at least half a page long and on a page boundary, UNIBUS page table entries are swapped to reference the pages, and then the initial pages are remapped from *ifw_wmap* when the transfer completes. The mbufs containing the mapped pages are placed on the *ifw_xtofree* queue to be freed after transmission.

When read transfers give at least half a page of data to be input, page frames are allocated from a network page list and traded with the pages already containing the data, mapping the allocated pages to replace the input pages for the next UNIBUS data input.

The following utility routines are available for use in writing network interface drivers; all use the structures described above.

if_ubaminit(ifubinfo, uban, hlen, nmr, ifr, nr, ifx, nx);
if_ubainit(ifuba, uban, hlen, nmr);

> *if_ubaminit* allocates resources on UNIBUS adapter *uban*, storing the information in the *ifubinfo*, *ifrw* and *ifxmt* structures referenced. The *ifr* and *ifx* parameters are pointers to arrays of *ifrw* and *ifxmt* structures whose dimensions are *nr* and *nx*, respectively. *if_ubainit* is a simpler, backwards-compatible interface used for hardware with single buffers of each type. They are called only at boot time or after a UNIBUS reset. One data path (buffered or unbuffered, depending on the *ifu_flags* field) is allocated for each buffer. The *nmr* parameter indicates the number of UNIBUS mapping registers required to map a maximal sized packet onto the UNIBUS, while *hlen* specifies the size of a local network header, if any, which should be mapped separately from the data (see the description of trailer protocols in chapter 14). Sufficient UNIBUS mapping registers and pages of memory are allocated to initialize the input data path for an initial read. For the output data path, mapping registers and pages of memory are also allocated and mapped onto the UNIBUS. The pages associated with the output data path are held in reserve in the event a write requires copying non-page-aligned data (see *if_wubaput* below). If *if_ubainit* is called with memory pages already allocated, they will be used instead of allocating new ones (this normally occurs after a UNIBUS reset). A 1 is returned when allocation and initialization are successful, 0 otherwise.

m = if_ubaget(ifubinfo, ifr, totlen, off0, ifp);
m = if_rubaget(ifuba, totlen, off0, ifp);

> *if_ubaget* and *if_rubaget* pull input data out of an interface receive buffer and into an mbuf chain. The first interface passes pointers to the *ifubinfo* structure for the interface and the *ifrw* structure for the receive buffer; the second call may be used for single-buffered devices. *totlen* specifies the length of data to be obtained, not counting the local network header. If *off0* is non-zero, it indicates a byte offset to a trailing local network header which should be copied into a separate mbuf and prepended to the front of the resultant mbuf chain. When the data amount to at least a half a page, the previously mapped data pages are remapped into the mbufs and swapped with fresh pages, thus avoiding any copy. The receiving interface is recorded as *ifp*, a pointer to an *ifnet* structure, for the use of the receiving network protocol. A 0 return value indicates a failure to allocate resources.

if_wubaput(ifubinfo, ifx, m);
if_wubaput(ifuba, m);

> *if_ubaput* and *if_wubaput* map a chain of mbufs onto a network interface in preparation for output. The first interface is used by devices with multiple transmit buffers. The chain includes any local network header, which is copied so that it resides in the mapped and aligned I/O space. Page-aligned data that are page-aligned in the output buffer are mapped to the UNIBUS in place of the normal buffer page, and the corresponding mbuf is placed on a queue to be freed after transmission. Any other mbufs which contained non-page-sized data portions are copied to the I/O space and then freed. Pages mapped from a previous output operation (no longer needed) are unmapped.

7. Socket/protocol interface

The interface between the socket routines and the communication protocols is through the *pr_usrreq* routine defined in the protocol switch table. The following requests to a protocol module are possible:

```
#define PRU_ATTACH       0   /* attach protocol */
#define PRU_DETACH       1   /* detach protocol */
#define PRU_BIND         2   /* bind socket to address */
#define PRU_LISTEN       3   /* listen for connection */
#define PRU_CONNECT      4   /* establish connection to peer */
#define PRU_ACCEPT       5   /* accept connection from peer */
#define PRU_DISCONNECT   6   /* disconnect from peer */
#define PRU_SHUTDOWN     7   /* won't send any more data */
#define PRU_RCVD         8   /* have taken data; more room now */
#define PRU_SEND         9   /* send this data */
#define PRU_ABORT        10  /* abort (fast DISCONNECT, DETATCH) */
#define PRU_CONTROL      11  /* control operations on protocol */
#define PRU_SENSE        12  /* return status into m */
#define PRU_RCVOOB       13  /* retrieve out of band data */
#define PRU_SENDOOB      14  /* send out of band data */
#define PRU_SOCKADDR     15  /* fetch socket's address */
#define PRU_PEERADDR     16  /* fetch peer's address */
#define PRU_CONNECT2     17  /* connect two sockets */
/* begin for protocols internal use */
#define PRU_FASTTIMO     18  /* 200ms timeout */
#define PRU_SLOWTIMO     19  /* 500ms timeout */
#define PRU_PROTORCV     20  /* receive from below */
#define PRU_PROTOSEND    21  /* send to below */
```

A call on the user request routine is of the form,

```
error = (*protosw[].pr_usrreq)(so, req, m, addr, rights);
int error; struct socket *so; int req; struct mbuf *m, *addr, *rights;
```

The mbuf data chain *m* is supplied for output operations and for certain other operations where it is to receive a result. The address *addr* is supplied for address-oriented requests such as PRU_BIND and PRU_CONNECT. The *rights* parameter is an optional pointer to an mbuf chain containing user-specified capabilities (see the *sendmsg* and *recvmsg* system calls). The protocol is responsible for disposal of the data mbuf chains on output operations. A non-zero return value gives a UNIX error number which should be passed to higher level software. The following paragraphs describe each of the requests possible.

PRU_ATTACH
: When a protocol is bound to a socket (with the *socket* system call) the protocol module is called with this request. It is the responsibility of the protocol module to allocate any resources necessary. The ‘‘attach’’ request will always precede any of the other requests, and should not occur more than once.

PRU_DETACH
: This is the antithesis of the attach request, and is used at the time a socket is deleted. The protocol module may deallocate any resources assigned to the socket.

PRU_BIND
: When a socket is initially created it has no address bound to it. This request indicates that an address should be bound to an existing socket. The protocol module must verify that the requested address is valid and available for use.

PRU_LISTEN
: The ‘‘listen’’ request indicates the user wishes to listen for incoming connection requests on the associated socket. The protocol module should perform any state changes needed to carry out this request (if possible). A ‘‘listen’’ request always precedes a request to accept a connection.

PRU_CONNECT

The ‘‘connect’’ request indicates the user wants to a establish an association. The *addr* parameter supplied describes the peer to be connected to. The effect of a connect request may vary depending on the protocol. Virtual circuit protocols, such as TCP [Postel81b], use this request to initiate establishment of a TCP connection. Datagram protocols, such as UDP [Postel80], simply record the peer’s address in a private data structure and use it to tag all outgoing packets. There are no restrictions on how many times a connect request may be used after an attach. If a protocol supports the notion of *multi-casting*, it is possible to use multiple connects to establish a multi-cast group. Alternatively, an association may be broken by a PRU_DISCONNECT request, and a new association created with a subsequent connect request; all without destroying and creating a new socket.

PRU_ACCEPT

Following a successful PRU_LISTEN request and the arrival of one or more connections, this request is made to indicate the user has accepted the first connection on the queue of pending connections. The protocol module should fill in the supplied address buffer with the address of the connected party.

PRU_DISCONNECT

Eliminate an association created with a PRU_CONNECT request.

PRU_SHUTDOWN

This call is used to indicate no more data will be sent and/or received (the *addr* parameter indicates the direction of the shutdown, as encoded in the *soshutdown* system call). The protocol may, at its discretion, deallocate any data structures related to the shutdown and/or notify a connected peer of the shutdown.

PRU_RCVD

This request is made only if the protocol entry in the protocol switch table includes the PR_WANTRCVD flag. When a user removes data from the receive queue this request will be sent to the protocol module. It may be used to trigger acknowledgements, refresh windowing information, initiate data transfer, etc.

PRU_SEND

Each user request to send data is translated into one or more PRU_SEND requests (a protocol may indicate that a single user send request must be translated into a single PRU_SEND request by specifying the PR_ATOMIC flag in its protocol description). The data to be sent is presented to the protocol as a list of mbufs and an address is, optionally, supplied in the *addr* parameter. The protocol is responsible for preserving the data in the socket’s send queue if it is not able to send it immediately, or if it may need it at some later time (e.g. for retransmission).

PRU_ABORT

This request indicates an abnormal termination of service. The protocol should delete any existing association(s).

PRU_CONTROL

The ‘‘control’’ request is generated when a user performs a UNIX *ioctl* system call on a socket (and the ioctl is not intercepted by the socket routines). It allows protocol-specific operations to be provided outside the scope of the common socket interface. The *addr* parameter contains a pointer to a static kernel data area where relevant information may be obtained or returned. The *m* parameter contains the actual *ioctl* request code (note the non-standard calling convention). The *rights* parameter contains a pointer to an *ifnet* structure if the *ioctl* operation pertains to a particular network interface.

PRU_SENSE

The ‘‘sense’’ request is generated when the user makes an *fstat* system call on a socket; it requests status of the associated socket. This currently returns a standard *stat* structure. It typically contains only the optimal transfer size for the connection (based on buffer size, windowing information and maximum packet size). The *m* parameter contains a pointer to a static kernel data area where the status buffer should be placed.

PRU_RCVOOB

Any ‘‘out-of-band’’ data presently available is to be returned. An mbuf is passed to the protocol module, and the protocol should either place data in the mbuf or attach new mbufs to the one supplied if there is insufficient space in the single mbuf. An error may be returned if out-of-band data is not (yet) available or has already been consumed. The *addr* parameter contains any options such as MSG_PEEK to examine data without consuming it.

PRU_SENDOOB

Like PRU_SEND, but for out-of-band data.

PRU_SOCKADDR

The local address of the socket is returned, if any is currently bound to it. The address (with protocol specific format) is returned in the *addr* parameter.

PRU_PEERADDR

The address of the peer to which the socket is connected is returned. The socket must be in a SS_ISCONNECTED state for this request to be made to the protocol. The address format (protocol specific) is returned in the *addr* parameter.

PRU_CONNECT2

The protocol module is supplied two sockets and requested to establish a connection between the two without binding any addresses, if possible. This call is used in implementing the system call.

The following requests are used internally by the protocol modules and are never generated by the socket routines. In certain instances, they are handed to the *pr_usrreq* routine solely for convenience in tracing a protocol’s operation (e.g. PRU_SLOWTIMO).

PRU_FASTTIMO

A ‘‘fast timeout’’ has occurred. This request is made when a timeout occurs in the protocol’s *pr_fastimo* routine. The *addr* parameter indicates which timer expired.

PRU_SLOWTIMO

A ‘‘slow timeout’’ has occurred. This request is made when a timeout occurs in the protocol’s *pr_slowtimo* routine. The *addr* parameter indicates which timer expired.

PRU_PROTORCV

This request is used in the protocol-protocol interface, not by the routines. It requests reception of data destined for the protocol and not the user. No protocols currently use this facility.

PRU_PROTOSEND

This request allows a protocol to send data destined for another protocol module, not a user. The details of how data is marked ‘‘addressed to protocol’’ instead of ‘‘addressed to user’’ are left to the protocol modules. No protocols currently use this facility.

8. Protocol/protocol interface

The interface between protocol modules is through the *pr_usrreq*, *pr_input*, *pr_output*, *pr_ctlinput*, and *pr_ctloutput* routines. The calling conventions for all but the *pr_usrreq* routine are expected to be specific to the protocol modules and are not guaranteed to be consistent across protocol families. We will examine the conventions used for some of the Internet protocols in this section as an example.

8.1. pr_output

The Internet protocol UDP uses the convention,

```
error = udp_output(inp, m);
int error; struct inpcb *inp; struct mbuf *m;
```

where the *inp*, ‘‘*in*ternet *p*rotocol *c*ontrol *b*lock’’, passed between modules conveys per connection state information, and the mbuf chain contains the data to be sent. UDP performs consistency checks, appends its header, calculates a checksum, etc. before passing the packet on. UDP is based on the Internet Protocol, IP [Postel81a], as its transport. UDP passes a packet to the IP module for output as follows:

```
error = ip_output(m, opt, ro, flags);
int error; struct mbuf *m, *opt; struct route *ro; int flags;
```

The call to IP's output routine is more complicated than that for UDP, as befits the additional work the IP module must do. The *m* parameter is the data to be sent, and the *opt* parameter is an optional list of IP options which should be placed in the IP packet header. The *ro* parameter is is used in making routing decisions (and passing them back to the caller for use in subsequent calls). The final parameter, *flags* contains flags indicating whether the user is allowed to transmit a broadcast packet and if routing is to be performed. The broadcast flag may be inconsequential if the underlying hardware does not support the notion of broadcasting.

All output routines return 0 on success and a UNIX error number if a failure occurred which could be detected immediately (no buffer space available, no route to destination, etc.).

8.2. pr_input

Both UDP and TCP use the following calling convention,

```
(void) (*protosw[].pr_input)(m, ifp);
struct mbuf *m; struct ifnet *ifp;
```

Each mbuf list passed is a single packet to be processed by the protocol module. The interface from which the packet was received is passed as the second parameter.

The IP input routine is a VAX software interrupt level routine, and so is not called with any parameters. It instead communicates with network interfaces through a queue, *ipintrq*, which is identical in structure to the queues used by the network interfaces for storing packets awaiting transmission. The software interrupt is enabled by the network interfaces when they place input data on the input queue.

8.3. pr_ctlinput

This routine is used to convey ‘‘control’’ information to a protocol module (i.e. information which might be passed to the user, but is not data).

The common calling convention for this routine is,

```
(void) (*protosw[].pr_ctlinput)(req, addr);
int req; struct sockaddr *addr;
```

The *req* parameter is one of the following,

```
#define PRC_IFDOWN              0     /* interface transition */
#define PRC_ROUTEDEAD           1     /* select new route if possible */
#define PRC_QUENCH              4     /* some said to slow down */
#define PRC_MSGSIZE             5     /* message size forced drop */
#define PRC_HOSTDEAD            6     /* normally from IMP */
#define PRC_HOSTUNREACH         7     /* ditto */
#define PRC_UNREACH_NET         8     /* no route to network */
#define PRC_UNREACH_HOST        9     /* no route to host */
#define PRC_UNREACH_PROTOCOL    10    /* dst says bad protocol */
#define PRC_UNREACH_PORT        11    /* bad port # */
#define PRC_UNREACH_NEEDFRAG    12    /* IP_DF caused drop */
#define PRC_UNREACH_SRCFAIL     13    /* source route failed */
#define PRC_REDIRECT_NET        14    /* net routing redirect */
#define PRC_REDIRECT_HOST       15    /* host routing redirect */
#define PRC_REDIRECT_TOSNET     14    /* redirect for type of service & net */
#define PRC_REDIRECT_TOSHOST    15    /* redirect for tos & host */
#define PRC_TIMXCEED_INTRANS    18    /* packet lifetime expired in transit */
#define PRC_TIMXCEED_REASS      19    /* lifetime expired on reass q */
#define PRC_PARAMPROB           20    /* header incorrect */
```

while the *addr* parameter is the address to which the condition applies. Many of the requests have obviously been derived from ICMP (the Internet Control Message Protocol [Postel81c]), and from error messages defined in the 1822 host/IMP convention [BBN78]. Mapping tables exist to convert control requests to UNIX error codes which are delivered to a user.

8.4. pr_ctloutput

This is the routine that implements per-socket options at the protocol level for *getsockopt* and *setsockopt*. The calling convention is,

```
error = (*protosw[].pr_ctloutput)(op, so, level, optname, mp);
int op; struct socket *so; int level, optname; struct mbuf **mp;
```

where *op* is one of PRCO_SETOPT or PRCO_GETOPT, *so* is the socket from whence the call originated, and *level* and *optname* are the protocol level and option name supplied by the user. The results of a PRCO_GETOPT call are returned in an mbuf whose address is placed in *mp* before return. On a PRCO_SETOPT call, *mp* contains the address of an mbuf containing the option data; the mbuf should be freed before return.

9. Protocol/network-interface interface

The lowest layer in the set of protocols which comprise a protocol family must interface itself to one or more network interfaces in order to transmit and receive packets. It is assumed that any routing decisions have been made before handing a packet to a network interface, in fact this is absolutely necessary in order to locate any interface at all (unless, of course, one uses a single ‘‘hardwired’’ interface). There are two cases with which to be concerned, transmission of a packet and receipt of a packet; each will be considered separately.

9.1. Packet transmission

Assuming a protocol has a handle on an interface, *ifp*, a (struct ifnet *), it transmits a fully formatted packet with the following call,

```
error = (*ifp->if_output)(ifp, m, dst)
int error; struct ifnet *ifp; struct mbuf *m; struct sockaddr *dst;
```

The output routine for the network interface transmits the packet *m* to the *dst* address, or returns an error indication (a UNIX error number). In reality transmission may not be immediate or successful; normally the output routine simply queues the packet on its send queue and primes an interrupt driven routine to actually transmit the packet. For unreliable media, such as the Ethernet, ‘‘successful’’ transmission simply means that the packet has been placed on the cable without a collision. On the other hand, an 1822 interface guarantees proper delivery or an error indication for each message transmitted. The model employed in the networking system attaches no promises of delivery to the packets handed to a network interface, and thus corresponds more closely to the Ethernet. Errors returned by the output routine are only those that can be detected immediately, and are normally trivial in nature (no buffer space, address format not handled, etc.). No indication is received if errors are detected after the call has returned.

9.2. Packet reception

Each protocol family must have one or more ‘‘lowest level’’ protocols. These protocols deal with internetwork addressing and are responsible for the delivery of incoming packets to the proper protocol processing modules. In the PUP model [Boggs78] these protocols are termed Level 1 protocols, in the ISO model, network layer protocols. In this system each such protocol module has an input packet queue assigned to it. Incoming packets received by a network interface are queued for the protocol module, and a VAX software interrupt is posted to initiate processing.

Three macros are available for queuing and dequeuing packets:

IF_ENQUEUE(ifq, m)
: This places the packet *m* at the tail of the queue *ifq*.

IF_DEQUEUE(ifq, m)
: This places a pointer to the packet at the head of queue *ifq* in *m* and removes the packet from the queue. A zero value will be returned in *m* if the queue is empty.

IF_DEQUEUEIF(ifq, m, ifp)
: Like IF_DEQUEUE, this removes the next packet from the head of a queue and returns it in *m*. A pointer to the interface on which the packet was received is placed in *ifp*, a (struct ifnet *).

IF_PREPEND(ifq, m)
: This places the packet *m* at the head of the queue *ifq*.

Each queue has a maximum length associated with it as a simple form of congestion control. The macro IF_QFULL(ifq) returns 1 if the queue is filled, in which case the macro IF_DROP(ifq) should be used to increment the count of the number of packets dropped, and the offending packet is dropped. For example, the following code fragment is commonly found in a network interface's input routine,

```
if (IF_QFULL(inq)) {
        IF_DROP(inq);
        m_freem(m);
} else
        IF_ENQUEUE(inq, m);
```

10. Gateways and routing issues

The system has been designed with the expectation that it will be used in an internetwork environment. The "canonical" environment was envisioned to be a collection of local area networks connected at one or more points through hosts with multiple network interfaces (one on each local area network), and possibly a connection to a long haul network (for example, the ARPANET). In such an environment, issues of gatewaying and packet routing become very important. Certain of these issues, such as congestion control, have been handled in a simplistic manner or specifically not addressed. Instead, where possible, the network system attempts to provide simple mechanisms upon which more involved policies may be implemented. As some of these problems become better understood, the solutions developed will be incorporated into the system.

This section will describe the facilities provided for packet routing. The simplistic mechanisms provided for congestion control are described in chapter 12.

10.1. Routing tables

The network system maintains a set of routing tables for selecting a network interface to use in delivering a packet to its destination. These tables are of the form:

```
struct rtentry {
        u_long  rt_hash;                 /* hash key for lookups */
        struct  sockaddr rt_dst;         /* destination net or host */
        struct  sockaddr rt_gateway;     /* forwarding agent */
        short   rt_flags;                /* see below */
        short   rt_refcnt;               /* no. of references to structure */
        u_long  rt_use;                  /* packets sent using route */
        struct  ifnet *rt_ifp;           /* interface to give packet to */
};
```

The routing information is organized in two separate tables, one for routes to a host and one for routes to a network. The distinction between hosts and networks is necessary so that a single mechanism may be used for both broadcast and multi-drop type networks, and also for networks built from point-to-point links (e.g DECnet [DEC80]).

Each table is organized as a hashed set of linked lists. Two 32-bit hash values are calculated by routines defined for each address family; one based on the destination being a host, and one assuming the target is the network portion of the address. Each hash value is used to locate a hash chain to search (by taking the value modulo the hash table size) and the entire 32-bit value is then used as a key in scanning the list of routes. Lookups are applied first to the routing table for hosts, then to the routing table for networks. If both lookups fail, a final lookup is made for a ‘‘wildcard’’ route (by convention, network 0). The first appropriate route discovered is used. By doing this, routes to a specific host on a network may be present as well as routes to the network. This also allows a ‘‘fall back’’ network route to be defined to a ‘‘smart’’ gateway which may then perform more intelligent routing.

Each routing table entry contains a destination (the desired final destination), a gateway to which to send the packet, and various flags which indicate the route’s status and type (host or network). A count of the number of packets sent using the route is kept, along with a count of ‘‘held references’’ to the dynamically allocated structure to insure that memory reclamation occurs only when the route is not in use. Finally, a pointer to the a network interface is kept; packets sent using the route should be handed to this interface.

Routes are typed in two ways: either as host or network, and as ‘‘direct’’ or ‘‘indirect’’. The host/network distinction determines how to compare the *rt_dst* field during lookup. If the route is to a network, only a packet’s destination network is compared to the *rt_dst* entry stored in the table. If the route is to a host, the addresses must match bit for bit.

The distinction between ‘‘direct’’ and ‘‘indirect’’ routes indicates whether the destination is directly connected to the source. This is needed when performing local network encapsulation. If a packet is destined for a peer at a host or network which is not directly connected to the source, the internetwork packet header will contain the address of the eventual destination, while the local network header will address the intervening gateway. Should the destination be directly connected, these addresses are likely to be identical, or a mapping between the two exists. The RTF_GATEWAY flag indicates that the route is to an ‘‘indirect’’ gateway agent, and that the local network header should be filled in from the *rt_gateway* field instead of from the final internetwork destination address.

It is assumed that multiple routes to the same destination will not be present; only one of multiple routes, that most recently installed, will be used.

Routing redirect control messages are used to dynamically modify existing routing table entries as well as dynamically create new routing table entries. On hosts where exhaustive routing information is too expensive to maintain (e.g. work stations), the combination of wildcard routing entries and routing redirect messages can be used to provide a simple routing management scheme without the use of a higher level policy process. Current connections may be rerouted after notification of the protocols by means of their *pr_ctlinput* entries. Statistics are kept by the routing table routines on the use of routing redirect messages and their affect on the routing tables. These statistics may be viewed using

Status information other than routing redirect control messages may be used in the future, but at present they are ignored. Likewise, more intelligent ‘‘metrics’’ may be used to describe routes in the future, possibly based on bandwidth and monetary costs.

10.2. Routing table interface

A protocol accesses the routing tables through three routines, one to allocate a route, one to free a route, and one to process a routing redirect control message. The routine *rtalloc* performs route allocation; it is called with a pointer to the following structure containing the desired destination:

```
struct route {
        struct    rtentry *ro_rt;
        struct    sockaddr ro_dst;
};
```

The route returned is assumed ‘‘held’’ by the caller until released with an *rtfree* call. Protocols which implement virtual circuits, such as TCP, hold onto routes for the duration of the circuit’s lifetime, while connection-less protocols, such as UDP, allocate and free routes whenever their destination address changes.

The routine *rtredirect* is called to process a routing redirect control message. It is called with a destination address, the new gateway to that destination, and the source of the redirect. Redirects are accepted only from the current router for the destination. If a non-wildcard route exists to the destination, the gateway entry in the route is modified to point at the new gateway supplied. Otherwise, a new routing table entry is inserted reflecting the information supplied. Routes to interfaces and routes to gateways which are not directly accessible from the host are ignored.

10.3. User level routing policies

Routing policies implemented in user processes manipulate the kernel routing tables through two *ioctl* calls. The commands SIOCADDRT and SIOCDELRT add and delete routing entries, respectively; the tables are read through the /dev/kmem device. The decision to place policy decisions in a user process implies that routing table updates may lag a bit behind the identification of new routes, or the failure of existing routes, but this period of instability is normally very small with proper implementation of the routing process. Advisory information, such as ICMP error messages and IMP diagnostic messages, may be read from raw sockets (described in the next section).

Several routing policy processes have already been implemented. The system standard ‘‘routing daemon’’ uses a variant of the Xerox NS Routing Information Protocol [Xerox82] to maintain up-to-date routing tables in our local environment. Interaction with other existing routing protocols, such as the Internet EGP (Exterior Gateway Protocol), has been accomplished using a similar process.

11. Raw sockets

A raw socket is an object which allows users direct access to a lower-level protocol. Raw sockets are intended for knowledgeable processes which wish to take advantage of some protocol feature not directly accessible through the normal interface, or for the development of new protocols built atop existing lower level protocols. For example, a new version of TCP might be developed at the user level by utilizing a raw IP socket for delivery of packets. The raw IP socket interface attempts to provide an identical interface to the one a protocol would have if it were resident in the kernel.

The raw socket support is built around a generic raw socket interface, (possibly) augmented by protocol-specific processing routines. This section will describe the core of the raw socket interface.

11.1. Control blocks

Every raw socket has a protocol control block of the following form:

```
struct rawcb {
        struct   rawcb *rcb_next;        /* doubly linked list */
        struct   rawcb *rcb_prev;
        struct   socket *rcb_socket;     /* back pointer to socket */
        struct   sockaddr rcb_faddr;     /* destination address */
        struct   sockaddr rcb_laddr;     /* socket's address */
        struct   sockproto rcb_proto;    /* protocol family, protocol */
        caddr_t  rcb_pcb;                /* protocol specific stuff */
        struct   mbuf *rcb_options;      /* protocol specific options */
        struct   route rcb_route;        /* routing information */
        short    rcb_flags;
};
```

All the control blocks are kept on a doubly linked list for performing lookups during packet dispatch. Associations may be recorded in the control block and used by the output routine in preparing packets for transmission. The *rcb_proto* structure contains the protocol family and protocol number with which the raw socket is associated. The protocol, family and addresses are used to filter packets on input; this will be described in more detail shortly. If any protocol-specific information is required, it may be attached to the control block using the *rcb_pcb* field. Protocol-specific options for transmission in outgoing packets may be stored in *rcb_options*.

A raw socket interface is datagram oriented. That is, each send or receive on the socket requires a destination address. This address may be supplied by the user or stored in the control block and automatically installed in the outgoing packet by the output routine. Since it is not possible to determine whether an address is present or not in the control block, two flags, RAW_LADDR and RAW_FADDR, indicate if a local and foreign address are present. Routing is expected to be performed by the underlying protocol if necessary.

11.2. Input processing

Input packets are ''assigned'' to raw sockets based on a simple pattern matching scheme. Each network interface or protocol gives unassigned packets to the raw input routine with the call:

```
raw_input(m, proto, src, dst)
struct mbuf *m; struct sockproto *proto, struct sockaddr *src, *dst;
```

The data packet then has a generic header prepended to it of the form

```
struct raw_header {
        struct      sockproto raw_proto;
        struct      sockaddr raw_dst;
        struct      sockaddr raw_src;
};
```

and it is placed in a packet queue for the ''raw input protocol'' module. Packets taken from this queue are copied into any raw sockets that match the header according to the following rules,

1) The protocol family of the socket and header agree.
2) If the protocol number in the socket is non-zero, then it agrees with that found in the packet header.
3) If a local address is defined for the socket, the address format of the local address is the same as the destination address's and the two addresses agree bit for bit.
4) The rules of 3) are applied to the socket's foreign address and the packet's source address.

A basic assumption is that addresses present in the control block and packet header (as constructed by the network interface and any raw input protocol module) are in a canonical form which may be ''block compared''.

11.3. Output processing

On output the raw *pr_usrreq* routine passes the packet and a pointer to the raw control block to the raw protocol output routine for any processing required before it is delivered to the appropriate network interface. The output routine is normally the only code required to implement a raw socket interface.

12. Buffering and congestion control

One of the major factors in the performance of a protocol is the buffering policy used. Lack of a proper buffering policy can force packets to be dropped, cause falsified windowing information to be emitted by protocols, fragment host memory, degrade the overall host performance, etc. Due to problems such as these, most systems allocate a fixed pool of memory to the networking system and impose a policy optimized for ''normal'' network operation.

The networking system developed for UNIX is little different in this respect. At boot time a fixed amount of memory is allocated by the networking system. At later times more system memory may be requested as the need arises, but at no time is memory ever returned to the system. It is possible to garbage collect memory from the network, but difficult. In order to perform this garbage collection properly, some portion of the network will have to be ''turned off'' as data structures are updated. The interval over which this occurs must kept small compared to the average inter-packet arrival time, or too much traffic may be lost, impacting other hosts on the network, as well as increasing load on the interconnecting mediums. In our environment we have not experienced a need for such compaction, and thus have left the problem unresolved.

The mbuf structure was introduced in chapter 5. In this section a brief description will be given of the allocation mechanisms, and policies used by the protocols in performing connection level buffering.

12.1. Memory management

The basic memory allocation routines manage a private page map, the size of which determines the maximum amount of memory that may be allocated by the network. A small amount of memory is allocated at boot time to initialize the mbuf and mbuf page cluster free lists. When the free lists are exhausted, more memory is requested from the system memory allocator if space remains in the map. If memory cannot be allocated, callers may block awaiting free memory, or the failure may be reflected to the caller immediately. The allocator will not block awaiting free map entries, however, as exhaustion of the page map usually indicates that buffers have been lost due to a ‘‘leak.’’ The private page table is used by the network buffer management routines in remapping pages to be logically contiguous as the need arises. In addition, an array of reference counts parallels the page table and is used when multiple references to a page are present.

Mbufs are 128 byte structures, 8 fitting in a 1Kbyte page of memory. When data is placed in mbufs, it is copied or remapped into logically contiguous pages of memory from the network page pool if possible. Data smaller than half of the size of a page is copied into one or more 112 byte mbuf data areas.

12.2. Protocol buffering policies

Protocols reserve fixed amounts of buffering for send and receive queues at socket creation time. These amounts define the high and low water marks used by the socket routines in deciding when to block and unblock a process. The reservation of space does not currently result in any action by the memory management routines.

Protocols which provide connection level flow control do this based on the amount of space in the associated socket queues. That is, send windows are calculated based on the amount of free space in the socket’s receive queue, while receive windows are adjusted based on the amount of data awaiting transmission in the send queue. Care has been taken to avoid the ‘‘silly window syndrome’’ described in [Clark82] at both the sending and receiving ends.

12.3. Queue limiting

Incoming packets from the network are always received unless memory allocation fails. However, each Level 1 protocol input queue has an upper bound on the queue’s length, and any packets exceeding that bound are discarded. It is possible for a host to be overwhelmed by excessive network traffic (for instance a host acting as a gateway from a high bandwidth network to a low bandwidth network). As a ‘‘defensive’’ mechanism the queue limits may be adjusted to throttle network traffic load on a host. Consider a host willing to devote some percentage of its machine to handling network traffic. If the cost of handling an incoming packet can be calculated so that an acceptable ‘‘packet handling rate’’ can be determined, then input queue lengths may be dynamically adjusted based on a host’s network load and the number of packets awaiting processing. Obviously, discarding packets is not a satisfactory solution to a problem such as this (simply dropping packets is likely to increase the load on a network); the queue lengths were incorporated mainly as a safeguard mechanism.

12.4. Packet forwarding

When packets can not be forwarded because of memory limitations, the system attempts to generate a ‘‘source quench’’ message. In addition, any other problems encountered during packet forwarding are also reflected back to the sender in the form of ICMP packets. This helps hosts avoid unneeded retransmissions.

Broadcast packets are never forwarded due to possible dire consequences. In an early stage of network development, broadcast packets were forwarded and a ‘‘routing loop’’ resulted in network saturation and every host on the network crashing.

13. Out of band data

Out of band data is a facility peculiar to the stream socket abstraction defined. Little agreement appears to exist as to what its semantics should be. TCP defines the notion of ‘‘urgent data’’ as in-line, while the NBS protocols [Burruss81] and numerous others provide a fully independent logical transmission channel along which out of band data is to be sent. In addition, the amount of the data which may be sent as an out of band message varies from protocol to protocol; everything from 1 bit to 16 bytes or more.

A stream socket’s notion of out of band data has been defined as the lowest reasonable common denominator (at least reasonable in our minds); clearly this is subject to debate. Out of band data is expected to be transmitted out of the normal sequencing and flow control constraints of the data stream. A minimum of 1 byte of out of band data and one outstanding out of band message are expected to be supported by the protocol supporting a stream socket. It is a protocol’s prerogative to support larger-sized messages, or more than one outstanding out of band message at a time.

Out of band data is maintained by the protocol and is usually not stored in the socket’s receive queue. A socket-level option, SO_OOBINLINE, is provided to force out-of-band data to be placed in the normal receive queue when urgent data is received; this sometimes amelioriates problems due to loss of data when multiple out-of-band segments are received before the first has been passed to the user. The PRU_SENDOOB and PRU_RCVOOB requests to the *pr_usrreq* routine are used in sending and receiving data.

14. Trailer protocols

Core to core copies can be expensive. Consequently, a great deal of effort was spent in minimizing such operations. The VAX architecture provides virtual memory hardware organized in page units. To cut down on copy operations, data is kept in page-sized units on page-aligned boundaries whenever possible. This allows data to be moved in memory simply by remapping the page instead of copying. The mbuf and network interface routines perform page table manipulations where needed, hiding the complexities of the VAX virtual memory hardware from higher level code.

Data enters the system in two ways: from the user, or from the network (hardware interface). When data is copied from the user’s address space into the system it is deposited in pages (if sufficient data is present). This encourages the user to transmit information in messages which are a multiple of the system page size.

Unfortunately, performing a similar operation when taking data from the network is very difficult. Consider the format of an incoming packet. A packet usually contains a local network header followed by one or more headers used by the high level protocols. Finally, the data, if any, follows these headers. Since the header information may be variable length, DMA’ing the eventual data for the user into a page aligned area of memory is impossible without *a priori* knowledge of the format (e.g., by supporting only a single protocol header format).

To allow variable length header information to be present and still ensure page alignment of data, a special local network encapsulation may be used. This encapsulation, termed a *trailer protocol* [Leffler84], places the variable length header information after the data. A fixed size local network header is then prepended to the resultant packet. The local network header contains the size of the data portion (in units of 512 bytes), and a new *trailer protocol header*, inserted before the variable length information, contains the size of the variable length header information. The following trailer protocol header is used to store information regarding the variable length protocol header:

```
struct {
	short	protocol;		/* original protocol no. */
	short	length;			/* length of trailer */
};
```

The processing of the trailer protocol is very simple. On output, the local network header indicates that a trailer encapsulation is being used. The header also includes an indication of the number of data pages present before the trailer protocol header. The trailer protocol header is initialized to contain the actual protocol identifier and the variable length header size, and is appended to the data along with the variable length header information.

On input, the interface routines identify the trailer encapsulation by the protocol type stored in the local network header, then calculate the number of pages of data to find the beginning of the trailer. The trailing information is copied into a separate mbuf and linked to the front of the resultant packet.

Clearly, trailer protocols require cooperation between source and destination. In addition, they are normally cost effective only when sizable packets are used. The current scheme works because the local network encapsulation header is a fixed size, allowing DMA operations to be performed at a known offset from the first data page being received. Should the local network header be variable length this scheme fails.

Statistics collected indicate that as much as 200Kb/s can be gained by using a trailer protocol with 1Kbyte packets. The average size of the variable length header was 40 bytes (the size of a minimal TCP/IP packet header). If hardware supports larger sized packets, even greater gains may be realized.

Acknowledgements

The internal structure of the system is patterned after the Xerox PUP architecture [Boggs79], while in certain places the Internet protocol family has had a great deal of influence in the design. The use of software interrupts for process invocation is based on similar facilities found in the VMS operating system. Many of the ideas related to protocol modularity, memory management, and network interfaces are based on Rob Gurwitz's TCP/IP implementation for the 4.1BSD version of UNIX on the VAX [Gurwitz81]. Greg Chesson explained his use of trailer encapsulations in Datakit, instigating their use in our system.

References

[Boggs79] Boggs, D. R., J. F. Shoch, E. A. Taft, and R. M. Metcalfe; *PUP: An Internetwork Architecture*. Report CSL-79-10. XEROX Palo Alto Research Center, July 1979.

[BBN78] Bolt Beranek and Newman; Specification for the Interconnection of Host and IMP. BBN Technical Report 1822. May 1978.

[Cerf78] Cerf, V. G.; The Catenet Model for Internetworking. Internet Working Group, IEN 48. July 1978.

[Clark82] Clark, D. D.; Window and Acknowledgement Strategy in TCP, RFC-813. Network Information Center, SRI International. July 1982.

[DEC80] Digital Equipment Corporation; *DECnet DIGITAL Network Architecture – General Description*. Order No. AA-K179A-TK. October 1980.

[Gurwitz81] Gurwitz, R. F.; VAX-UNIX Networking Support Project – Implementation Description. Internetwork Working Group, IEN 168. January 1981.

[ISO81] International Organization for Standardization. *ISO Open Systems Interconnection – Basic Reference Model*. ISO/TC 97/SC 16 N 719. August 1981.

[Joy86] Joy, W.; Fabry, R.; Leffler, S.; McKusick, M.; and Karels, M.; Berkeley Software Architecture Manual, 4.3BSD Edition. *UNIX Programmer's Supplementary Documents*, Vol. 1 (PS1:6). Computer Systems Research Group, University of California, Berkeley. May, 1986.

[Leffler84] Leffler, S.J. and Karels, M.J.; Trailer Encapsulations, RFC-893. Network Information Center, SRI International. April 1984.

[Postel80] Postel, J. User Datagram Protocol, RFC-768. Network Information Center, SRI International. May 1980.

[Postel81a] Postel, J., ed. Internet Protocol, RFC-791. Network Information Center, SRI International. September 1981.

[Postel81b] Postel, J., ed. Transmission Control Protocol, RFC-793. Network Information Center, SRI International. September 1981.

[Postel81c] Postel, J. Internet Control Message Protocol, RFC-792. Network Information Center, SRI International. September 1981.

[Xerox81] Xerox Corporation. *Internet Transport Protocols*. Xerox System Integration Standard 028112. December 1981.

[Zimmermann80] Zimmermann, H. OSI Reference Model – The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*. Com-28(4); 425-432. April 1980.