Chapter 4. Networking

Table of Contents

This chapter describes code modules related to core networking. It begins with a discussion on core networking components: the routing code, sockets and mbufs. After this attention is given to the TCP/IP suite. Information on services provided by the networking stack, including various pseudo devices, can be found from the chapter on networking services .

This chapter describes only the involved interfaces and is meant to give an overview for people wishing to work with the code. A line-by-line discussion on most parts of the networking subsystem can be found from TCP/Illustrated, Vol2. It is still mostly accurate these days, and this chapter does not attempt to rewrite what was already written.

4.1. Routing

The routing code controls the flow of all packets within the networking subsystem. It makes the decision of where a packet with a given destination address should be sent next. The current routing table can be dumped using the command **netstat -r**; going over the normal table contents may be beneficial for the following discussion.

For example, in TCP/IP networking the routing module forwards the packets to the correct addresses and is involved in choosing the network interface through which the packets should go. The decision whether to send to an intermediate gateway or directly to the target is made here. And, finally, link layer lookups are done, which in the case of Ethernet and IPv4 would be ARP.

One important concept in the routing subsystem is cloning routes. These routes are "cloned" to create a more specific route each time they are resolved; copy-on-read if it makes any sense. A natural example is an interface route to the local Ethernet. Any packet sent through this link will get a cloned route entry for ARP resolution. This enables the caching of ARP responses without adding any explicit support for it on the routing layer.

The routing code can be thought to consist of three separate entities. An overview will be given now and a more dedicated discussion can be found later.

- The routing database backend is implemented in `net/radix.c`. As hinted by the name, the internal structure is a radix tree.
- The route query and control interface is located in `net/route.c`. This module is accessed from within the kernel and parameters are passed mainly using the struct route and the struct rtentry structures.
- The routing socket interface is located in `net/rtsock.c`. It is used to control the routing table entries from userspace and is accessed by opening a socket in the protocol family PF_ROUTE.

It is good to keep in mind that routing control messages can come either from a process in the operating system (user running **/sbin/route**, **gated**, etc.) or from the network (e.g. ICMP). This explains why internally some portions are controlled and parameters passed using a fashion similar to what the routing socket uses.

4.1.1. Network Routing DB Backend

This section describes the interface to the radix tree routing database. The actual internal operation of the radix code is beyond the scope of this document and can be found, for example, in TCP/IP Illustrated Vol2. Much of the complexity of this module is due to its aspiration to be as generic as possible and adaptable to any networking domain.

A radix tree is accessed mostly through the member functions in struct radix_node_head (but there are some exceptions). These function pointers are initialized after a radix tree is created with `rn_inithead` called from each networking domain's initialization routine. The head is then stored in the global array `rt_tables` using the domain type as the indexing value.

As arguments the radix tree functions take void *'s, which point to a struct sockaddr type of data structure. But since the radix tree layer treats the arguments as opaque data and a struct sockaddr contains header data (such as the sockaddr family *`sa_family`* or, specific to IP, the port *`sin_port`*) before the actual network address, each networking domain specifies a bit offset where the network address is located. This offset is given in struct domain by `dom_rtoffset` and used when testing against the supplied void *'s. Additionally, the radix tree expects to find the length of the structure underlying the void * at the first byte of the provided argument. This also matches the (BSD) struct sockaddr layout.

Most of the interface methods are located in the struct radix_node_head accessible through `rt_tables`. Theoretically the jump through the function pointer hoop is unnecessary, since the pointers are initialized to the same values for all trees in `net/radix.c`. Not all of the members of the structure are ever used and only the ones used are described here. Included in the description are also the ones not provided through function pointers, but accessed directly. They can be differentiated by looking at the function prefix: ones inside radix_node_head start with rnh, while directly accessed ones start with rn.

Table 4.1. struct radix_node_head interfaces

name	description
rnh_addaddr	Adds an entry with the given address and netmask to the database. The storage space for the database structures is provided by the caller as an argument (usually these are part of struct rtentry, as we will see later). The route target is already prefilled into the storage space by the caller.
rnh_deladdr	Removes an address with a matching netmask from the given radix database.
rnh_matchaddr	Locate the entry in the database which provides the best match for the given address. "Best match" is defined as the match having the longest prefix of 1-bits in the netmask.
rnh_lookup	Locate the most exact entry in the database for the given address with the netmask of the entry matching the argument given.
rnh_walktree	Walks the database calling the provided function for each entry. This is useful e.g. for dumping the entire routing table or flushing routes cloned from a certain parent entry.
rn_search	Returns the leaf node found at the end of the bit comparisons. This is either a match or the leaf in the the tree that should be backtracked to find a match.
rn_refines	Compares two netmasks and checks which one has more bits set, i.e. which one is "more exact".
rn_addmask	Enters the supplied netmask into the netmask tree.

4.1.2. Routing Interface

The route interface implemented in `net/route.c` is used by the kernel proper when it requires routing services to discover where a packet should be sent to.

The argument passing to this modules revolves around, in addition to struct sockaddr and the radix structures mentioned in the previous chapter, two structures: struct route and struct rtentry.

The structure struct route contains a struct rtentry in addition to a struct sockaddr signalling the destination address for the route. The destination is decoupled from the routing information so that multiple destinations could share the same route structure; think outgoing connections routed to the gateway, for example.

The routing information itself is contained in struct rtentry defined in `net/route.h`. It consists of the following fields:

Table 4.2. struct rtentry members

type	name	description
struct radix_node [2]	*rt_nodes*	radix tree glue
struct sockaddr *	*rt_gateway*	gateway address
int	*rt_flags*	flags
u_long	*rt_use*	number of times the route was used
struct ifnet *	*rt_ifp*	pointer to the interface structure of the route target
struct ifaddr *	*rt_ifa*	Pointer to the interface address of the route target. The sockaddr behind this pointer can be for example of type struct sockaddr_in or struct sockaddr_dl.
const struct sockaddr *	*rt_genmask*	Cloning mask for struct sockaddr.
caddr_t	*rt_llinfo*	Link level structure pointer. For example, this will point to struct llinfo_arp for Ethernet.
struct rt_metrics	*rt_rmx*	The route metrics used by routing protocols. This includes information such as the path MTU and estimated RTT and RTT variance.
struct rtentry *	*rt_gwroute*	In the case of a gateway route, this contains (after it is resolved) the route to the gateway. For example, on IP/Ethernet, the route containing the gateway IP address will have a pointer to the gateway MAC address route here.
LIST_HEAD(, rttimer)	`rt_timer`	Misc. timers associated with a route. This queue is accessed through the `rt_timer*` family of functions declared in `net/route.h`.
struct rtentry *	`rt_parent`	The parent of a cloned route. This is used for cleanup when the parent route is removed.

Routes can be queried through two different interfaces:

```
        void rtalloc(struct route *ro);
        struct rtentry *rtalloc1(const struct sockaddr *dst, int report);
```

The difference is that the former is a convenience function which skips the routing lookup in case the rtentry contained within struct route already contains a route that is up. As an example, struct route is included in struct inpcb to act as a route cache. It helps especially for TCP connections, where the endpoint does not change and therefore the route remains the same.

The latter is used to lookup information from the routing database. The `report` parameter is overloaded to control two operations: whether to create a routing socket message in case there is no route available and whether to clone a route in case the resolved route is a cloning route (quite clearly both of these conditions cannot be true during the same lookup, so the only possibility for ambiguity is for the programmers).

After routing information (struct rtentry) is no longer needed, the routing entry is to be released using `rtfree`. This takes care of appropriate reference counting and releasing the underlying data structures. Notice that this does not delete a routing table entry, it merely releases a route entry created from a routing table entry.

4.1.3. Routing Sockets

Routing sockets are used for controlling the routing table. The routing socket kernel portion is implemented in the file `net/rtsock.c`.

While in BSD systems the routing code is within the kernel, the decisions on the routing table entries are controlled from outside, usually from a userspace routing daemon. The routing socket (simply a socket of type PF_ROUTE) enables the communication between the user and kernel portions. For simpler systems, such as normal desktop machines with essentially the default gateway address being the only routing information, the routing socket is used to set the gateway address. Smart routers will use this interface for programs such as **routed**.

The routing socket acts like any other socket: it is possible to write to it or read from it. Writing (from the userland perspective) is handled by `route_output`, while data is transferred to userspace using the `raw_input` call giving raw data and the routing socket addressing identifiers as the parameters.

The data passed through the routing socket is described by a structure called struct rt_msghdr, which is declared in `net/route.h`. The message type field in the header identifies the type of activity taking place, e.g. RTM_ADD means adding a new route and RTM_REDIRECT means redirecting an existing route. In the latter case, the input comes from the network e.g. in the form of an ICMP packet.

The addresses involved with the routing messages are handled in a somewhat non-obvious way within the file `net/rtsock.c`. They are passed as binary data in the message, read into a structure called struct rt_addrinfo, and used like local variables throughout the file because of preprocessor magic such as the following:

```
#define dst     info.rti_info[RTAX_DST]
```

To be compatible with the routing socket and its propagation of information, networking subsystems should support the `rtrequest` interface. It is called through the *`if_rtrequest`* member in struct ifaddr. Examples include `arp_rtrequest` for Ethernet interfaces and `llc_rtrequest` in the OSI ISO stack. Rtrequest methods handle the same requests as what are communicated via the routing socket (RTM_ADD, RTM_DELETE, ...), but the actual routing socket message layout is handled within the routing socket code.

4.2. Sockets

The Berkeley abstraction for a communication endpoint is called a socket. From the userspace perspective, the handle to a socket is an integer, no different from any other descriptor. There are differences, such as support for interface calls reserved only for sockets (`getsockopt`, `setsockopt`, etc.), but on the basic level, the interface is the same.

However, under the hood things diverge quickly. The integer descriptor value is used to lookup the kernel internal file descriptor structure, struct file, as usual. After this the socket data structure, struct socket is found from the pointer *f_data* in struct file.

When discussing sockets it is important to remember that they were designed as an abstraction to the underlying protocol layers.

4.2.1. Socket Data Structure

Inside the kernel, a socket's information is contained within struct socket. This structure is not defined in `sys/socket.h`, which mostly deals with the user-kernel interface, but rather in `sys/socketvar.h`.

Table 4.3. struct socket members

type	name	description
short	*so_type*	The generic socket type. Well-known examples are SOCK_STREAM and SOCK_DGRAM
short	*so_options*	socket options. Most of these, such as SO_REUSEADDR, can be set using `setsockopt`. Others, such as SO_ACCEPTCONN as set using other methods (in this case calling `listen`)
short	*so_linger*	Time the socket lingers on after being closed. Used if SO_LINGER is set. An example user is TCP.
short	*so_state*	internal socket state flags controlled by the kernel. Some, however, are indirectly settable by userspace, for example SS_ASYNC to deliver async I/O notifications.
const struct protosw *	*so_proto*	socket protocol handle. This is used to attach the socket to a certain protocol, for example IP/UDP. The socket protocol requests, such as PRU_USRREQ used for sending packets, are accessed through this member. See also section on socket protocol support.
short	*so_timeo*	connection timeout, not used except as a wakeup() address(?)
u_short	*so_error*	an error value. This field is used to store error values that should be returned to socket routine callers once they are executed/scheduled.
pid_t	*so_pgid*	the pgid use to identify the target for socket-related signal delivery.
u_long	*so_oobmark*	counter to the oob mark
struct sockbuf	*so_snd*, *so_rcv*	Socket send and receive buffers, see section on socket buffers for further information.
void (*so_upcall)	*so_upcall*	In-kernel upcall to make when a socket wakeup occurs. The canonical example is nfs, which uses sockets from inside the kernel for network request servicing.
caddr_t	*so_upcallarg*	argument to be passed `so_upcall`.
int (*so_send)	*so_send*	socket receive method. This is always currently `sosend` (which eventually leads to *so_proto*'s PR_USRREQ), but might change in the future.
int (*so_receive)	*so_receive*	socket receive method. Same holds as for *so_send*.
struct mowner *	*so_mowner*	owner of the mbuf's for the socket, used to track mbufs other than socket buffer mbufs. This can be used to debug mbuf leaks. Available only when the

		kernel is compiled with options MBUFTRACE.
struct uidinfo *	*so_uidinfo*	socket owner information. This is currently used to limit socket buffer size.

Additionally, *so_head*, *so_onq*, *so_q0*, *so_q*, *so_qe*, *so_qlen* and *so_qlimit* are use to queue and control incoming partial connections and handle aborts.

4.2.2. Socket Buffers

The socket buffer plays a critical role in the operation of the networking subsystem. It is used to buffer incoming data before it is read by the application and outgoing data before it can be sent to the network. As noted above, a struct socket contains two socket buffers, one for each direction. A socket buffer is described by struct sockbuf in `sys/socketvar.h`.

Table 4.4. struct sockbuf members

type	name	description
struct selinfo	*sb_sel*	Contains the information on which process (if any) wants to know about changes in the socket, for example `poll` called with POLLOUT on the socket.
struct mowner *	*sb_mowner*	Used to track owners of the socket buffer mbufs, tracking enabled by options MBUFTRACE.
u_long	*sb_cc*	counter for octets in the buffer.
u_long	*sb_hiwat*	high water mark for the socket buffer
u_long	*sb_mbcnt*	bytes allocated as mbuf memory in the socket buffer. This is the sum of regular mbufs and mbuf externals.
u_long	*sb_mbmax*	maximum amount of mbuf memory that is allowed to be allocated for the socket buffer.
long	*sb_lowat*	low watermark for socket buffer. Writing is disallowed unless there is more than the low watermark space in the socket and conversely reading is disallowed, if there is less data than the low watermark.
struct mbuf *	*sb_mb*, *sb_mbtail*, *sb_lastrecord*	mbuf chains associated with the socket buffer
int	*sb_flags*	flags for the socket buffer, such as locking and async I/O information
int	*sb_timeo*	time to wait for send space or receivable data.
u_long	*sb_overflowed*	statistics on times we had to drop data due to the socket buffer being full.

Socket buffers are manipulated by the `sb*` family of functions. Examples include `sbappend`, which appends data to the socket buffer (it assumes that relevant space checks have been made prior to calling it) and `sbdrop`, which is used to remove packets from the front of a socket buffer queue. The latter is used also by e.g. TCP for removing ACKed data from the send buffer (recall that "original" TCP requires to ACK data in-order).

4.2.3. Socket Creation

A socket is created by making the system call `socket`, which is handled inside the kernel by `sys__socket30` in `kern/uipc_syscalls.c` (`sys_socket` is reserved for compat30 ABI). First, a file descriptor structure is allocated for the socket using `fdalloc`. Then, the socket structure itself is created and initialized in `socreate` in `kern/uipc_socket.c`.

`socreate` reserves memory for the socket data structure from a pool and initializes the members that were discussed in the section Socket Data Structure. It also calls the socket's protocol's `pr_usrreq` method with the PRU_ATTACH argument. This allows to do protocol-specific initialization, such as reserve memory for protocol control blocks.

4.2.4. Socket Operation

Sockets are handled through the `so*` family if functions. Some of them map directly to system calls, such as `sobind` and `soconnect`, while others, such as `sofree` are meant for kernel internal consumption.

Socket control routines take data arguments in the form of the memory buffers (mbufs) used in the networking stack. The callers of these functions must be prepared to handle mbufs, although usually this can arranged for with a calls to `sockargs`. It should be noted, that the comment above the function takes an attitude towards this behaviour:

```
/*
 * XXX In a perfect world, we wouldn't pass around socket control
 * XXX arguments in mbufs, and this could go away.
 */
```

Note: In case a perfect world is some day being planned, the author should also be contacted, since he can contribute a whole lot of ideas for that goal.

The critical socket routines are `sosend` and `soreceive`. These are used for transmitting and receiving network data. They make sure that the socket is in the correct state for data transfer, handle buffering issues and call the socket protocol methods for doing data access. They are, as mentioned above in the socket member discussion, not called directly but rather through the *`so_send`* and *`so_receive`* members.

4.2.5. Socket Destruction

Sockets are destroyed once they are no longer useful. This is done when their reference count in the file descriptor table drops to zero. The socket is first disconnected, if it was connected at all. Then, if the socket option SO_LINGER was set, the socket lingers around until either the timer expires or the connection is closed. After this the socket is detached from its associated protocol and finally freed.

4.2.6. Protocol Support

Each protocol (e.g. TCP/IP or UDP/IP) has operations which depend on its functionality. These are controlled through the *`so_proto`* member in struct socket. While the member provides many different interfaces, the socket is interested in two: `pr_ctloutput`, which is used for control output and `pr_usrreq`, which is used for user requests. Additionally, the socket code is interested in the flags set for the protocol pointed to by *`so_proto`*. These flags are defined in `sys/protosw.h`, but examples include PR_CONNREQUIRED and PR_LISTEN, both of which the TCP protocol sets but UDP sets neither.

Control output is used to set or get parameters specific to the protocol. These are called from the kernel implementations of `setsockopt` and `getsockopt`. If the level parameter for the calls is set appropriately, the calls will trickle to the correct layer (e.g. TCP or IP) before taking action. For instance, `tcp_ctloutput` checks if the request is for itself and proceeds to query the IP layer if it discovers that the call should be passed down.

The user request method handles multiple types of different requests coming from the user. A complete list is defined in `sys/protosw.h`, but examples include PRU_BIND for binding the protocol to an address (e.g. making data received at an address:port UDP pair accepted), PRU_CONNECT for initiating a protocol level connect (e.g.

TCP handshake) and PRU_SEND for sending data.

4.3. mbufs

The data structure used to pass data around in the networking code is known as an mbuf. An mbuf is described by struct mbuf, which is defined in `sys/mbuf.h`. However, it is not defined in the regular fashion, but rather through the macro MBUF_DEFINE. To understand the need for this trickery, we need to first look at the structure of an mbuf.

To accommodate for the needs of networking subsystem, an mbuf needs to provide cheap operations for prepending headers and stripping them off. Therefore an mbuf is structured as a list of constant-size struct mbufs, of which each consist of a structure header and optional secondary headers or data.

this is mostly TODO, still

4.4. IP layer

TODO

4.5. UDP

TODO

4.6. TCP

TODO

Prev Next
Chapter 3. Processes and threads Home Chapter 5. Networking Services