Network Communication

The network subsystem provides a general-purpose framework for network communication including:

- A structured interface to the socket level.
- A consistent interface to the hardware devices that transmit and receive data.
- Network independent support for message routing.

Network Communication

The network subsystem is situated below the socket layer and is logically divided into three layers:

Transport layer The transport layer provides interprocess data transports and implements the transport level protocols.

Network layer The network layer is responsible for the delivery of data to the correct host. It is responsible for host addressing, routing and if needed packet fragmentation and reassembly.

Network-interface layer The bottom layer, also called *link layer*, is responsible for transmitting data between hosts connected to a common transmission media.

- The transport, network and network-interface layers of the network subsystem corresponds to the *transport*, *network* and *link* layers in the ISO OSI model.
- The internal structure of the network subsystem is not externally visible, instead all networking facilities are accessed through the socket interface.
- Each communication protocol that permits external access to its facilities exports a set of user request routines to the socket layer.

Data Flow

There are only four real paths through a network node:

Inbound Data received at a network interface flows upward through communication protocols until they are placed in a receive queue.

Outbound Data delivered by system calls flows down to the network-interface for transmission.

Forward Whether bridged or routed the packets is not for this node but to be sent on to another host or network.

Error A packet has arrived that requires the network subsystem itself to send a response.

- Data flowing upward are received asynchronously and are passed from the network interface to the appropriate communication protocol through per-protocol input message queues (Fig.12.1).
- The system handles inbound network traffic by splitting the processing of packets between the network driver's upper and lower halves:
  - → The lower half runs at interrupt level.
  - → The upper half of the driver runs as an interrupt thread that queues the packets for the network thread *swi_net*.

Network thread

The network thread, *swi_net*, is a kernel thread whose sole job is to handle the network layer protocol input queues.

Once the packet is queued by the device's interrupt thread, the *swi_net* thread is responsible for handling the packet:

- If the packet is for a higher level protocol, this protocol is invoked directly.
- If the packet is for another host and the system is configured as a router, the packet may be returned to the network interface for retransmission

Communication Protocols

- Protocol modules are described by a *protocol-switch* struct (Fig. 12.2) that contains the set of externally visible entry points and certain attributes.
- The socket layer interacts with a communication protocol solely through the protocols *protocol-switch* struct.
- The protocol-protocol interface in the kernel also use the *protocol-switch* struct.
- Before a protocol is used the protocols initialization routine, *pr_init()* is called.
- Thereafter, the protocol will be invoked for timer-based actions every 200 ms if the *pr_fasttimo()* entry is present and every 500 ms if the *pr_slowtimo()* entry is present.
- Protocols mostly use the slower timer, the major use of the fast timeout is for delayed-acknowledgment processing for reliable protocols.

Communication Protocols

Protocols may pass data between their layers by calling routines in the *protocol-switch* table.

The following routines are defined for protocol-protocol communication:

pr_input() Passes data up toward the user.

pr_output() Passes data down toward the network.

pr_ctlinput() Passes control information up toward the user.

pr_ctloutput() Passes control information down toward the network.

- The interface between the protocols and the socket layer is through the *pr_usrreqs()* table.
- On output the lowest level reached must free space passed as arguments.
- At input the highest level is responsible for freeing space passed up to it.
- The *pr_flags field* in the *protocol-switch* struct defines some aspects of the protocol (Table 12.1).

Network Interfaces

- Each *network interface* configured in the system defines a link-layer path through which messages can be sent or received.
- The *network interface* abstraction provides the protocols with a consistent interface to all network hardware devices.
- In most cases a *network interface* is associated with a hardware device but it may also use a software loopback device.
- A *network interface* and its addresses are described by the *ifnet* and *ifaddr* data structures (Fig. 12.3).
- As network devices are found at system configuration time, an *ifnet struct* is allocated, initialized and placed at a linked list.
- Each network interface is identified in two ways: a character string (e.g. eth0) and a systemwide index number.
- The actual addresses are stored in a link level version of the *sockaddr struct* (sockaddr_dl) with family code AF_LINK (Fig. 12.4).

Network Interfaces

The *ifnet* struct includes an *if_data* struct that contains the externally visible description of the interface.

Part of *ifnet* struct is also the *if_flags* field that contains flags that define the state of the interface (Table 12.2).

Some examples of such flags are:

IFF_BROADCAST The interface has broadcast capabilities.

IFF_POINTOPOINT The interface is associated with a point-to-point hardware link.

IFF_UP Set when the interface is configured and ready to transmit messages.

IFF_SIMPLEX Set if the network hardware is unable to receive packets that they send.

IFF_PROMISC Set by network-monitoring programs to receive all packets incoming on the interface and not just packets addressed to the local system.

IFF_OACTIVE Indicates that the interface is busy doing output.

Network Interfaces

- Interface addresses and flags are set with *ioctl* requests.
- An address is assigned to an interface by the SIOCSIFADDR ioctl request.
- One or more extra *alias addresses* can be assigned to an interface by the SIOCAIFADDR ioctl request.
- In either case the protocol allocates an ifaddr struct and adds it to the *if_addrhead* list in the *ifnet* struct (Fig. 12.5).
- An address can be deleted with the SIOCDIFADDR request.
- The SIOCSIFFLAGS request can be used to change the state of an interface.

Network Interfaces

The *ifnet* struct also contains the head of a link level output queue, *if_snd,* and interface routines for calling the driver:

if_output() Accepts a packet from the protocol level, adds link level headers and queues the packet at the *if_snd* queue. If the IFF_OACTIVE flag is not set, it calls *if_start()*.

if_start() Calls the driver specific start routine to initiate transmission.

Socket-to-Protocol Interface

- The interface from the socket layer to the protocol layer is through the *pr_usrreqs* table.
- The entry points in the *pr_usrreqs* table is listed in Table 12.3.
  - → The first argument to the routines is always a pointer to a socket struct.
  - → An mbuf data chain is provided for output operations.
  - → A pointer to a *sockaddr* struct is provided for address-oriented requests.

Examples of routines are:

pru_listen() Protocol level routine for the listen system call. The protocol routine should make any state changes needed to meet this request.

pru_send() Each user request to send data is translated into one or more calls to this routine.

pru_rcvoob() This routine requests that any out-of-band data now available be returned. It is no routine at this level to receive normal data; They are queued at the socket receive queue by protocol routines activated from interrupt level.

Socket-to-protocol Control Routines

- The *pr_ctloutput()* is called from the *getsockopt* and *setsockopt* system calls to set or get protocol options.
  - → The direction is determined by a parameter that can be SOPT_GET or SOPT_SET.
- The *pr_ctlinput()* routine passes *control* information upward from one protocol module to another.
  - → It takes two parameters: *cmd* and a *sockaddr* struct.
  - → The *cmd* parameters takes the values shown in Fig. 12.4. Most of these values are from the Internet Control Message Protocol (ICMP).

Interface Between Protocol and Network Interface

- The lowest level in a protocol stack must interact with an interface to send or receive packets.
- It is assumed that any routing decision is taken before a packet is sent to the interface; In fact a routing decision is needed to locate the interface.
- This leaves only two cases to be concerned with at the interface level: Transmission of a packet and receipt of a packet.

Network Interface Level

Packet transmission

- When an interface has been chosen, it is identified by *ifp*, a pointer to an *ifnet* struct.
- The output routine is called by *ifp->if_output()*.
- The output routine takes four parameters: a pointer to *ifnet* struct, a pointer to an *mbuf* chain, a pointer to a *sockaddr* and pointer to a *rtentry* struct.
- The network destination address is passed to the output routine in the *sockaddr* struct.
  - → This destination address must be mapped to a link layer address by the *if_output()* routine.
  - → The output routine must understand the address formates for all protocols it supports to perform the mapping.
  - → The mapping may be a table lookup or (if a broadcast network) it may require more involved techniques such as the Address Resolution Protocol (ARP).

Network Interface Level

Packet Reception

- Network interfaces receive packets and enqueue the packet at the appropriate network level input queue based on information in the link layer protocol header (Fig.12.6).
- The packet queues at the interface level use mbufs in the same way as the higher levels, but they use a special queue header defined by the *ifqueue* struct.
- Data received from the network are converted to a chain of mbufs, typically by the hardware specific interrupt handler, before being queued at the protocol input queue.

Some macros are available for manipulating the mbuf queues:

IF_ENQUEUE(ifq, m) Place the packet *m* at the tail of queue *ifq*.

IF_PREPEND(ifq, m) Place the packet *m* at the head of queue *ifq*.

IF_DEQUEUE(ifq, m) Dequeue a packet from the queue *ifq* to mbuf *m*.

Every queue is protected by a mutex to prevent that it is manipulated by different processors at the same time.

Packet Reception Example

Every protocol that communicates with the interface level must register it's *pr_input()* and input queue with the network thread via the *netisr_register()* routine.

For an Ethernet the following events will typically take place when a packet is received:

- The hardware specific interrupt handler will allocate a mbuf chain and copy data from the controller memory to this mbuf chain.
- The hardware specific handler calls *if_input()* (which is *ether_input*) with the mbuf chain as parameter.
- *ether_input* calls *ether_demux()* that determines which protocol this packet is destined to, by decoding the type code in the Ethernet packet header.
- *ether_demux()* then strips off the Ethernet header and calls *netisr_dispatch()* to enqueue the packet at the correct protocol.
  - → When the packet is placed on the queue *schednetisr()* is called to wake up the *swi_net* network thread that takes care of calling the protocol input routine.
- In most cases *netisr_dispatch()* will actually call the protocol input routine itself instead of queuing the packet.
  - → The direct call method requires that the protocol input routine is multithread safe (several instances of it can run in parallel).

Routing

- The Networking system was designed for a heterogeneous network environment with local-area networks interconnected through routers (also called gateways) (Fig. 12.7).
- All BSD systems have the potential to work as a router if running on a machine with multiple network interfaces and configured as a router.

A routing system have two major components (Fig. 12.8):

- Gathering and maintenance of route information.
  - → Performed by a user mode routing daemon.
- Forwarding of packets
  - → Selection of a network interface on which a packet will be sent - performed by the kernel.
  - → Uses a simple lookup that provides a first-hop route for each outbound packet.
- Communication between the routing daemon and the kernel is through a routing socket.

Kernel Routing

- The kernel routing mechanism implements a routing table for looking up a first-hop route.
- The main components of the routing mechanism is the *rtentry* data structure and a lookup algorithm.
- A destination is described by a *sockaddr* struct.

Routes can be classified in two categories:

- Host routes
  - → The destination address must exactly match the desired destination.
- Network routes
  - → The destination address is paired with a mask.
  - → The route matches any address that contains the same bits as the destination in the positions indicated by the mask.
- A special case is the *wildcard route* that has an empty mask
  - → Matches every destination and is used as default route for destinations not otherwise known.

Kernel Routing Cont.

Another classification of routes is direct or indirect:

- A *direct route* leads directly to the destination.
  - → The first hop of the path is the entire path.
- An *indirect route* specifies a router on a local network that is the first-hop destination for the packet.
  - → For an *indirect route* the network protocol header specifies the address of the eventual destination, but the link level header specifies the address of the first-hop destination.

In FreeBSD the local-remote decision is made as part of the routing lookup.

- If the best route is direct, then the destination is local otherwise the route is indirect.

Kernel Routing Tables

- The kernel maintains one routing table for every address family it supports.
- The kernel routing tables consist of routing entries specified by the *rtentry* struct (Table 12.5).
- The *rt_flags* field in the *rtentry* struct defines the type of the route and some other attributes (Table 12.6).

Some flags are:

RTF_HOST The route is for a specific host (if not set its a network route)

RTF_GATEWAY The route is to a gateway and the destination address in the link layer header should be set from the *rt_gateway* field.

RTF_REJECT Marks the destination of the route as being unreachable causing an error when trying to send.

RTF_BLACKHOLE Similar to RTF_REJECT but packets are silently dropped.

RTF_CLONING Indicates that the route is a generic route that must be *cloned* and made more specific before use.

When a route is added, created by cloning or deleted, the link layer is called via the *ifa_rtrequest()* entry in the *ifaddr* struct for the interface.

Routing Lookup

- The routing table is organized as a *radix search trie*.
- The radix search algorithm uses a binary tree of nodes beginning with a root node for each address family.
- The search keys are the 32 bit IP addresses (for the Internet domain).
- The most significant bit position in the address is numbered as bit 0.
- The internal nodes in the tree stores the bit position to be tested in that node. If the lookup address have 1 in the tested bit position the search continues in the right subtree otherwise in the left subtree (Fig. 12.9)
- This lookup technique tests the minimum number of bits required to distinguish between the keys that are stored in the tree.
- Once a leaf node is reached it either contains the lookup key or the key is missing in the tree.
- If the lookup key is missing, a modified algorithm is used to backtrack up the tree checking each parent with a mask until a match is found.

Routing-Table Interface

- A protocol calls the *rtalloc()* routine to look up a route in the routing table.
- *rtalloc()* takes a pointer to a *route* struct as parameter (Fig.12.10).
- A *route* struct has two entries:
  - → A *sockaddr* struct that contains the destination address.
  - → A pointer to an *rtentry* struct that will be set to reference the routing entry that is the best match for the destination.
- The returned route is assumed to be held by the caller until released by the RTFREE macro.

User-Level Routing Interface

- User mode programs use a raw socket in the routing protocol family, AF_ROUTE, to communicate with the kernel routing layer.
- This socket operates like a normal datagram socket, except that communication takes place between a user process and the kernel.
- Messages to the kernel are requests to add, modify or delete a route.
- Messages include a header with a message type identifying the action (table 12.7).
- Requests to add or modify a route includes all the information needed for the route including the route flags (Table 12.6).
- The kernel sends a message in reply to a request.
- The kernel also sends messages to all open routing sockets to inform about asynchronous state changes in local interfaces.

User-Level Routing Polices

- *Routing policies* are determined by user mode *routing daemons* such as **routed**.
- When routers talk to each other, they use routing protocols such as RIP (Routing Information Protocol).
- Work stations configured as routers usually do not run routing daemons, instead *static routes* are set up with commands such as *route*.

Buffering and Congestion Control

- A fixed amount of memory is allocated at boot time to the zone allocator to use for mbufs and mbuf clusters.
- More system memory can be requested for mbuf clusters at later time if need arises.
- When a socket is created, the protocol reserves some amount of buffer space for input and output queues.
  - → These amounts defines the high watermarks.
- Incoming packets are always received unless memory allocation fails.
- However, each network level protocol input queue has a maximum queue length and if this length is exceeded packets will be dropped.
- Limiting output queue lengths can be used on hosts that routes from a fast network to a slow network.
  - → To large queues in this case will cause unacceptable delays.

Raw Sockets

- A *raw socket* allows privileged processes direct access to protocols other than those normally used for transport of user data.
  - → The **ping** program is implemented using an ICMP (Internet Control Message Protocol) socket.
  - → The raw IP socket provide an external interface to IP that can be used by usermode implementations of transport protocols.
- Every raw socket has a protocol control block defined by a *rawcb* struct (Fig. 12.11).
- All raw control blocks are kept on a linked list headed by a global variable in the kernel.
- This list of *rawcb* blocks is searched by the *raw_input*() routine to locate a raw socket for a received packet.
- The *raw_input()* routine match on protocol, family and addresses to determine if a packet matches a raw socket.
- Input packets are placed into the socket input queue for all raw sockets that match the packet header.
- Each protocol calls the *raw_input*() routine to handle unassigned packets.

ARP

- The *Address Resolution Protocol* (ARP) is a link level protocol that provides dynamic address translations for networks that support broadcast or multicast addressing.
- ARP maps a 32-bit IPv4 address to a 48-bit Ethernet address.
- The interface to ARP is the *arpresolve()* routine that is called by interface output routines.
- If the translation is in the ARP cache, the Ethernet address is immediately returned otherwise an ARP message is created that specifies the requested Internet address.
- The ARP message is broadcast by the Ethernet interface with the expectation that some host at the network will know the translation - usually because it is the intended recipient for the packet.

ARP Implementation

- The ARP cache is implemented as part of the routing tree.
  - → It is stored in an *llinfo_arp* struct, pointed to by the *rt_llinfo* pointer in the *rtentry* struct.

The *arpresolve()* routine is called with the following parameters:

struct ifnet *ifp The interface to use
struct rtentry *rt The route to the final destination
struct mbuf *msg The packet to send
struct sockaddr *dst The next hop IP address
u_char *desten Return parameter - The looked up Ethernet address

- If the route passed in the *rt* parameter already contains a *complete* translation that has not timed out, the Ethernet address is returned in the *desten* return parameter.
- If the link level address is not known or has timed out, the *msg* mbuf is queued in *llinfo_arp* until the address has been resolved.
  - → An ARP broadcast message is sent requesting the Ethernet address for the *dst* address.
  - → *Arpresolve()* returns the EWOULDBLOCK error code.

ARP Implementation Cont.

- After some time a response is received to the ARP broadcast.
- The received packet is processed in the normal way by the *ether_demux()* routine.
- Because the Ethernet type code in the packet is ETHERTYPE_ARP, it is queued for the *arpintr*() routine.

arpintr()

- If the packet was an answer to the own broadcast, the ARP cache is updated with the received Ethernet address.
  - → The queued *msg* is retransmitted with a call to the interface output routine.
  - → This time the resultant call to *arpresolve()* will return without delay.
- If the packet was an ARP request from another host, a response packet is sent.
- ARP normally times out entries in the cache after 20 minutes