A Tree-Based Packet Routing Table for Berkeley Unix

Keith Sklower

Computer System Research Group
EECS Department, Computer Science Division
University of California
Berkeley, California 94720

ABSTRACT

Packet forwarding for OSI poses strong challenges for routing lookups: the algorithm must be able to efficiently accommodate variable length, and potentially very long addresses. The 4.3 Reno release of Berkeley UNIX† uses a reduced radix tree to make decisions about forwarding packets.

This data structure is general enough to encompass protocol to link layer address translation such as the Address Resolution Protocol (ARP), and the End System to Intermediate System Protocol (ES–IS), and should apply to any hierarchical routing scheme, such as source and quality-of-service routing, or choosing between multiple Datakits on a single system.

The system uses a message oriented mechanism to communicate between the kernel and user processes to maintain the routing database, inform user processes of spontaneous events such as redirects, routing lookup failures, and suspected timeouts through gateways.

1. Introduction

An important focus of the 4.3 Reno release of Berkeley UNIX was to make support for the OSI protocols publicly available. OSI addresses are typically very long (20 bytes) and with the explosive growth of the Internet, a router may have to contend with thousands of them.

The traditional hash-based scheme of routing lookups would perform poorly in this environment. The older algorithm assumed that it would be cheap to compute hashes, that one could easily identify the network portion of an address, and easily compare them.

It is likely to be expensive to compute the hash of a 20 byte address. Moreover, where there are multiple hierarchies, it would be complicated and context dependent to identify which portion of the address should be considered as ‘‘the network portion’’ for comparison at changing levels. In general, it is not apparent how to accommodate hierarchies while using hashing, other than rehashing for each level of hierarchy possible.

Van Jacobsen, of the Lawrence Berkeley Laboratory, suggested using the PATRICIA algorithm (described below), but with an additional invariant to maintain a routing tree. This meshes extremely well with notions of multiple hierarchical defaults, and the cost of an entire lookup is approximately the same as the cost of computing a single hash.

Since there is now a means to store variable length addresses, and reason to use addresses of differing sizes within a given route (using a protocol destination address with a link-layer gateway to accomplish ARP-like translation, for example), it was decided that using fixed length ioctl’s to communicate between the kernel and routing process would be too restrictive. Instead, a message

† UNIX is a registered trademark of AT&T Bell Laboratories in the USA and other countries.

based mechanism is used for passing routing information to and from the kernel. This mechanism provides additional potential for remote management when future releases supply the ability to splice communications channels.

2. Kernel Issues

2.1. Routing Lookups

2.1.1. Restatement of the Problem.

Let's describe the problem once again in a little more detail: A packet arrives with a very long protocol address. If the destination address is not that of the local system, one wants to decide quickly how to forward it. This decision entails choosing a network interface and a next-hop agent. (Point-to-point links only have one agent on the other end of the link, so sometimes it's enough just to figure out which link!).

Some of the routing protocols currently in use give us criteria for making this choice, in what may seem a bizarre way: the space of addresses is partitioned into a set of equivalence classes by specifying a pair consisting of a prototype address and a bitmask; a test address is deemed to belong to the class if any bit in which it differs from the prototype address corresponds to a zero bit of the provided mask.

Let's give an example, using the protocol addresses for the Internet Family [Post], which are 32-bit numbers:

Example 1: Some Address Classes

Prototype	Mask	ClassName
0x80030000	0xffff0000	LBL
0x80200000	0xffff0000	Berkeley
0x80208200	0xffffff00	CsDivSubnet
0x80209600	0xffffff00	SpurSubnet
0	0	TheOutside

The author's machine (okeeffe.Berkeley.EDU) has the address 0x80208203. Consequently, it belongs to the classes Berkeley, CsDivSubnet, TheOutside, but not LBL nor SpurSubnet. With each class is associated a networking interface, in most cases a next-hop agent, and a collection of other useful information, and that collection is referred to as a ‘‘route’’. Continuing the example, okeeffe.Berkeley.EDU can talk directly to any system in the class CsDivSubnet (all such systems are on a single ethernet), but requires an intermediary to talk to anybody else.

The routing lookup problem is to find the most specific class containing a given protocol address. Paradoxically, that will be the one with largest number of one bits in the mask. The NSF net may provide a regional router with about 2000 routes of this type. The lookup algorithm must look up the appropriate class quickly (among both numerous and lengthy addresses), and yet have nice properties with respect to masks.

2.1.2. The algorithm

The collection of prototype addresses are assembled into a variant of a PATRICIA tree, which is technically a binary radix tree with one-way branching removed. (In fact some writers call any tree with explicit external and internal nodes a *trie*). Although this algorithm is given a lengthy exposition in [Sedg] and also is discussed in [Knut] and [Morr], we will review it here.

We build a tree with internal nodes and leaves. The leaves will represent address classes, and will contain information common to all possible destinations in each class. As such, there will be at least a mask and prototype address. Each internal node represents a bit position to test. Given the tree and a candidate address thought of as a sequence of bits, the lookup algorithm is as follows:

1. Set node to the top of the tree.
2. If at a leaf node, stop.
3. Extract a bit position to test.
4. If that bit of the candidate address is on, set the node to the right child of the current node, otherwise set node to the left child.
5. Repeat steps 2 – 4.

Once we arrive at a leaf node, we need to check whether we have selected the appropriate class. The class may consist of a single host. This is a special case where the mask consists of all one bits, but it is a common enough occurence that we check for it (by use of a null pointer for the mask), and do an outright string compare. Otherwise, in which case we perform the masking operation.

It is possible to have the same prototype address with differing masks; this is handled by a linked list of leaf nodes. This arises due to boundary conditions for the smallest representation of the default route (which collides with the boundary marker for the empty tree). It also arises if you want to route to subnet 0 of a subnetted class A or B internet address.

If the leaf node isn't correct, then we backtrack up the tree looking for indications that a more general mask may apply (i.e. one having fewer one bits). This may happen if we are asked to look up an address other than the prototype addresses used to construct the tree. Rather than keep a separate stack of nodes traversed while searching the tree, backtracking is facilitated by having explicit parent pointers in each node. This also facilitates deletion, and allows non-recursive walks of the tree.

2.1.3. A Lookup Example

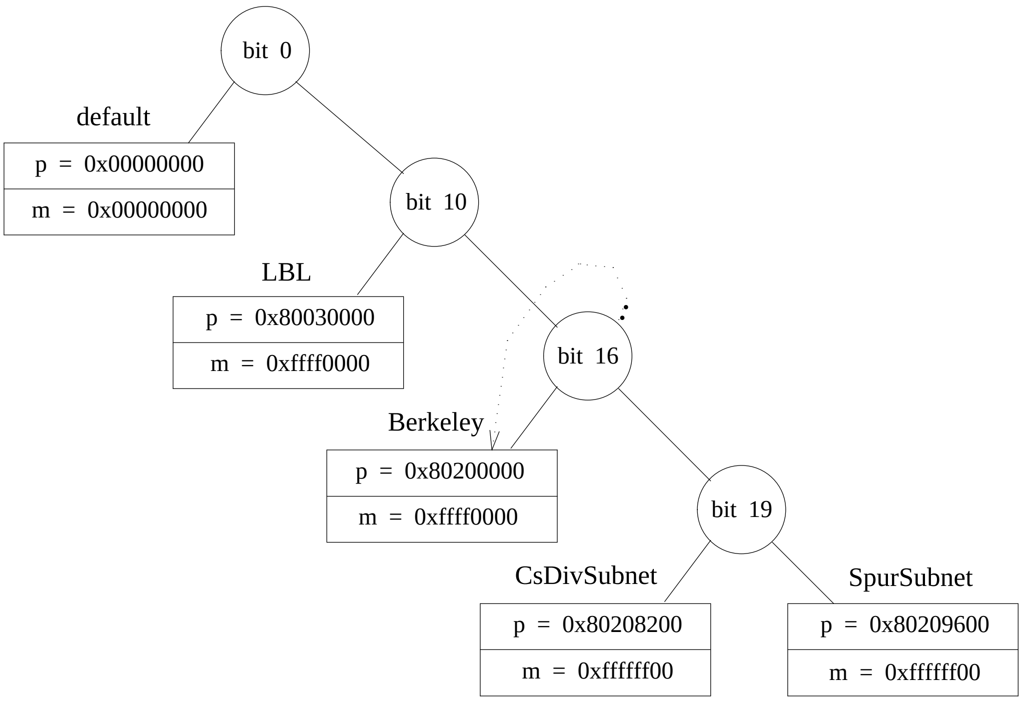

Figure 1.

Figure 1. shows how one would construct a reduced radix tree to decide among the prototype addresses given in the example of address classes. In examining the address for okeeffe.Berkeley.EDU (0x80208203), we find that bit 0 is on (0x80000000: go right), bit 10 is on (0x00200000: go right), bit 16 is on (0x00008000: go right), but that bit 19 is off (0x00001000:

go left). And, in fact okeeffe does match the CsDivSubnet class.

If we were to look up another machine at Berkeley, say miro.Berkeley.EDU (0x80209514), we are driven down to the SpurSubnet class, which does not match. So we backtrack up to the second internal node above it, which has an indication that there is a mask which may apply. This is represented in the diagram by the dotted line, which actually points to data associated with mask contained in the leaf, rather than the leaf itself.

Backtracking only occurs when given packets are covered by a default route, or when non-prefix masks are employed. The current implementation deals with non-contiguous masks in a way requiring an explicit masking and re-lookup operation for each possibly applicable mask encountered while backtracking. This has the advantage that routes can be entered one by one without requiring searches or reorganization of subtrees.

Researchers in the field ([Tsuc], [Butl]) have suggested this might be avoided by constructing a tree in which the nodes test bits in non-increasing order, governed by the masks found in leaves underneath. This is the object of current study.

2.1.4. Comparison with the previous method.

Releases of Berkeley UNIX prior to 4.3 Reno employed an explicit three level hierarchy for routes, routing first to hosts, then to networks, then to defaults. The collections of host routes and network routes were entirely seperate hash arrays.

Given a candidate address, an address family specific method would be invoked to compute a hash value for the hosts array, and a bucket chosen. Each element of the bucket would be compared against the candidate address, via a second address family specific method for each comparison (i.e. requiring a subroutine call per comparision).

If the candidate address was not found, the process would be repeated with the network hash array. If that failed, a list of defaults would be searched to see if there were any for the address family of the candidate address.

By contrast, the initial search of the tree also is written in a protocol independent way. Furthermore, the new algorithm performs its comparisons in a protocol independent way, permitting the back-tracking loop to occur without separate subroutine calls.

It is interesting to note that in the average case, PATRICIA trees are approximately balanced. The expected length of a search is only 1.44 log (number of entries); and of course the maximum possible search is the number of bits in the address. By contrast, the worst case for a degenerate hash is the number of entries to be searched. So, if we had 2000 IP entries, in the PATRICIA case one would expect 15 bit tests, whereas in the typical hashing situation of sqrt(N), on would expect about 44 hash entries in 44 hash chains, with an average of 22 comparisons after hashing. The (pathological) worst case is 32 bit tests versus 2000 compares.

2.2. The Routing Entry

The routing entry is a collection of information for use by protocol implementations. It has a base part which contains all the aspects common to a class of hosts with which we might want to communicate that we can express in a protocol-independent (or protocol-uniform) way. It certainly includes the connectors used in constructing the tree, the prototype address (which we think of as the destination address), and the mask.

There are binary flags present which may greatly alter the interpretation of the route, or even cause new ones to spring into existence. (see **Cloning Operation**, below). There are pointers to a protocol independent control structure describing the network interface (the *ifnet* structure), and

---

The internal node based on bit 10 does not have an indication that there is a mask which may apply to it. This because any search backtracking through there would have had to had a 1 for bit 10 (since it otherwise would have been trapped by the leaf for LBL itself), as the LBL class has that bit off. There is a measure of how high in the tree a mask can apply (in our current scheme) which we call the index of the mask. The interested reader can peruse the source code in 4.3 Reno for further elaboration.

the protocol level address which should be used in identifying the local system (the *ifaddr* structure).

There is a possible gateway address, that is used in situations in which protocol packets must be sent to the intended recipients via an intermediary. This mode of operation is identified by having a flag **RTF_GATEWAY**. In the other case (where no gateway is required), there is a pointer to device– and protocol– specific information, such as link-layer to protocol address translations, or even other protocol control blocks for situations such as running connectionless protocols over X.25.

The route includes a collection of statistics that are commonly maintained by reliable and flow controlled protocols, such as round-trip time, round-trip time variance, maximum packet size for this path, maximum number of gatewaying or forwarding operations expected, in-bound and out-bound throughput measures, and a bitmask to indicate if any of these values should be left unaltered by protocol operation. There are some other statistics that are purely housekeeping matters, such as the number of protocol control blocks keeping a reference to this route.

2.2.1. Cloning Operation

As mentioned above, it is sometimes convenient for skeletal routing entries to be created and partially filled in upon first reference (or lookup), with the missing information to be supplied later. Allocating a dedicated routing entry at initial connect time saves the expensive of checking validity on each use.

An example of this would be a user opening a TCP connection to another machine on the same ethernet, for which the link-layer address was not yet known. The creation of such entries would triggered by the flag **RTF_CLONING** in a route being looked up.

For other sorts of link layer translations, such as IP address to X.121 addresses for use over a public data network, it may be desirable to have a message sent to a user level daemon when the route is created, requesting an external resolution of protocol addresses. This mode is enabled by the flag **RTF_XRESOLVE**.

Another example would be a configuration in which there are many different subnets on the other side of a serial link, where each subnet may have different performance characteristics (which could be learned operationally), but that each use would be infrequent and random enough that it would be wasteful permanently to allocate space for routing entries to each possible subnet in advance.

Here, a way to specify the netmask for the newly cloned route is necessary, which needs to be more specific than the netmask for the cloning route which creates it. Thus, the route structure includes a pointer for this secondary mask, which is only used in such a situation. The primary netmask is used for ‘‘trapping’’ the lookup; the secondary mask would be used as the primary mask in the newly created route which would restrict additional lookups to that newly identified class of hosts.

2.2.2. Black Holes (or Border Patrol)

A handy use for hierarchical defaults would be at the gateway of a campus to catch packets for non-existent subnets or hosts within the campus that would otherwise be sent to the default route advertised by the regional connection to a backbone network. This is easily implemented by the flag **RTF_REJECT**.

In 4.3 Reno, the network output routines added an additional parameter, a pointer to the route. This parameter enabled cached link layer information to be retrieved, but also allows the loopback driver to recognize the **RTF_REJECT** flag. When it does so, it consumes the offending packet and returns EHOSTUNREACH or ENETUNREACH, prompting the protocols to do the appropriate magic with no other changes.

2.3. Ancillary addressing structures:

2.3.1. The protocol independent network interface structure (ifnet)

For each network interface device, there is a structure describing a number of protocol independent elements. Some of these serve to identify the device: a printable name and a unit identifier. There are also general statistics kept about each device, some inherent about the device itself (such as maximum packet size, a generic type for the device, and binary flags indicating whether the device is point-to-point or broadcast or deaf to its own broadcasts). There are other statistics reflecting use, such as number of packets in and outbound, (and how many of those encountered errors), time of last use, total number of bytes transmitted and received. There are a collection of methods associated with the device. These include a general output routine to process packets and place them on an output queue, an internal routine to initiate transmission, an ioctl routine, initialization, reset, and two routines used at startup time.

This structure has escaped relatively unchagned from previous versions of BSD; a good description of it can be found in [Leff]. The new additions include the device start method which has makes it possible for all ethernet drivers to use a common output routine, more statistics required by SNMP [Case], and throughput statistics used for protocol operation.

2.3.2. The protocol address structure (sockaddr)

All protocol addresses have a common two byte header detailing the length and type of the address.

The 4.3 Reno release adds a device independent link-layer address format, which may be used in sending link-layer packets or disambiguating interfaces when more than one have the same protocol addresses.

2.3.3. The protocol dependent interface addressing structure (ifaddr)

There may be multiple protocol addresses associated with each network interface. The *ifaddr* structure provides a place to store them and other device– and protocol–specific information. In fact, some protocols allow either multiple names for the same interface, or the same name for multiple interfaces or both.

Even though the values will differ from protocol to protocol, there are some other common elements that can be identified, so that this structure has a protocol independent header, with a protocol specific tail expected to follow immediately.

The protocol independent elements include the address, associated subnet mask, destination or broadcast address, linkage to the next address and the associated *ifnet*, a method to be invoked when routes associated with this address are created or deleted, a routing entry associated with this address for this interface, and generic flags for this level.

This structure is also discussed in [Leff]. The method, routing entry, and flags fields are new (since 4.3 BSD), and the protocol addresses have been changed to pointers rather than allocating fixed size spaces for them.

3. Messages and Formats

As mentioned above, BSD has adopted a message passing approach for management of the routing table, for a variety of reasons. First, network address are of variable length, and we may have varying numbers of them in differing operations. Second, it provides a clean and uniform way of informing a routing process of spontaneous events, such a redirects, routing misses, requests to resolve link layer address translations, or internal evidence that a gateway may have crashed, (due to lack of acknowledgments across a class of connections). Third, it provides a way for making additions or changes to the management interface while maintaining backwards compatibility. A version number is embedded in the message header, and each message is self delimiting, so that any unknown message to the user program can be skipped. Finally, for the future when it will be possible to splice message streams together, it provides an easy path towards remote system management.

3.1. The basic format

All messages have a common header, and some varying number of protocol addresses appended to them. The header includes the total length of the message, a version number and a type, which allows non-understood messages to be skipped. There is space for a user-supplied sequence number. The returned message includes the pid of the originating process.

The header also includes a number of metrics, a bit mask to identify which are being specified, and a second bit mask to specify which metrics must remained unchanged by the protocols.

The interpretation and number of the trailing protocol addresses is specified by a bitmask. The potential addresses are:

Symbolic Name	Description
RTA_DST	Prototype address
RTA_NETMASK	Bitmask for describing class
RTA_GATEWAY	Gateway
RTA_GENMASK	Bitmask for routes created by cloning
RTA_IFA	The protocol address to be used as a source address when sending to hosts covered by this route
RTA_IFP	An address unambiguously specifying which interface (struct *ifnet*) is associated with this route, such as a link-level address.
RTA_AUTHOR	Address identifying sender of redirect, etc.

3.2. Message Types.

In this section, we'll discuss each of the message types, describing features unique to each, and contrasting the intent of otherwise similar looking messages.

3.2.1. RTM_ADD − Enter a new route into the table.

This is the basic operation for creating the routing table. The destination and gateway addresses must be present. If there is no netmask present, the route is assumed to be a route to a host. In the case where the host or class specified by the route is directly reachable, the gateway address may be used to specify a link layer address (for hosts), or the protocol address of the outgoing interface, which may implicity identify the *ifaddr* and *ifnet* structure pointers. Even the case of a class reached via a gateway, one may be able to deduce the interface from the address of the gateway. If there is ambiguity about this, as may be the case in OSI protocol operation, they must be explicitly supplied.

The flags may specify cloning operation, as described in section 2.2.1. If the the new routes are to specify a subclass instead of a host route, a generating bit mask needs to be supplied.

3.2.2. RTM_DELETE − Remove an entry from the table

If there is only one entry in the routing table with a given prototype address, that is sufficient to identify the route to be deleted. Otherwise, the netmask associated with the route must additionally be specified.

3.2.3. RTM_CHANGE − Alter characteristics of a route

Due to gateways going up or down, it may be desireable to change the designated forwarding agent for a class of hosts. It is also desirable to do so atomically (locking out forwarding requests), so that there isn't a period in which incorrect host or network unreachable protocol messages are generated in response to packets to be forwarded. Changing the gateway implicitly or explicitly requires changing the associated *ifaddr* and *ifnet* structures.

In this message, one can also alter the metrics associated with a route or some of the flags (cloning, resolving, link-layer-ness).

Altering the netmask associated with a route is not permitted, since this would affect the geometry of the tree; instead one deletes and re-inserts.

3.2.4. RTM_GET − Look up route and report characteristics.

This message is diagnostic in nature. The user supplies a destination and the best match route indication is returned, along with all of the metrics filled in. Where there are multiple routes with the same prototype address (but multiple netmasks), specifying the netmask will allow the user to select the appropriate route.

3.2.5. RTM_REDIRECT − Request to change gateway.

This message is an example of a spontaneous event. Both the TCP/IP and OSI family of protocols have the potential for receiving an advisory report from a gateway that the initiating system would be better off sending a packet to another gateway on the same network for forwarding to a remote location.

When the routing table is maintained by a user-level process, it is important that the routing process be notified of any changes to the routing table.

For OSI protocols, the initiating system may get a message specifying the original destination, a bitmask specifying a class of hosts for which this redirect also pertains, the replacement gateway to be used, and the author of the message.

3.2.6. RTM_LOSING − Trouble reports.

‘‘Reliable’’ byte and message stream protocols such as TCP or OSI−TP keep retransmission timers. If a connection suddenly stops working, it may signal the loss of a gateway. User-level routing processes may be interested in keeping track of such events, at the very least to determine if it appears the local or a remote gateway as failed. This message identifies the route which covers the remote hosts involved in such lossage.

3.2.7. RTM_MISS − A routing table lookup failed.

The local system was asked to forward a packet or initiate a connection to a destination for which it could not find a suitable route. One could imagine a system attached to a wide area network which would only allow a limited number of active reachable destinations, such as an X.25 network. The system might only enter those active peers in the network table, and open new ones (or close old ones) based on the number of misses.

This may be useful for purely diagnostic purposes as well.

3.2.8. RTM_RESOLVE − Request to complete route info via CHANGE.

This is very similar to the RTM_MISS message. It is intended for cloning operation (which would not otherwise cause an RTM_MISS type message) where some information needs to be obtained externally from some process that is not convenient to be coded directly into the kernel.

4. Measurements

We performed a synthetic test of constructing a routing table of about 1600 entries using both the new and old methods (in a user-level process). We then searched each table randomly 100000 times for entries in the table. The routing table was constructed from data obtained on a gateway system at Cornell University, which stands between the Cornell campus and the NSF net.

In fact, the time required to construct a table of 1600 routes was on the order of half a second for either method; our test actually measured constructing the table 10 times and emptying it 9 times. The test results show the new (radix tree based) method to be about 50% faster in constructing the tree and 200% faster in searching it. The overhead column represents the time required to loop through all routes calling a routine that does nothing instead of adding, deleting or lookup a route. The units in the table below are user time in seconds, as measured on a CCI tahoe processor running 4.3 Reno BSD.

operation	old	new	overhead
create	10.28	6.75	.10
search	29.72	7.38	.86

5. References:

[Butl]
Butler, Duane M. Private communication, August 1990.

[Case]
Case, J.D., *et al.* Simple Network Management Protocol RFC 1157, SRI Network Information Center, May 1990.

[Knut]
Knuth, Donald E. *The Art of Computer Programming*, Vol 3. pp. 490-493, Addison-Wesley, Reading MA 1973

[Morr]
Morrison, Donald R. *J ACM* **15** (1968), pp. 514-534

[Leff]
Leffler, Samuel J. *et al.* *The Design and implementation of the 4.3BSD UNIX© Operating System.* Addison-Wesley, Reading, MA 1988

[Post]
Postel, J. ‘‘Internet Protocol – DARPA Internet Program Protocol Specification,’’ RTC 791, USC Information Sciences Institute, September 1981.

[Sedg]
Sedgwick, Robert. *Algorithms in C.* pp. 253-257, Addison-Wesley, Reading MA 1990

[Tsuc]
Tsuchiya, Paul. Efficient assignement of addresses in the Internet. (IETF Proceedings, July 1990),

Appendix A. Radix Tree Declarations and Search Algorithm

We include a somewhat simplified version of the header file for the radix tree; but the algorithm for searching the tree is taken verbatim.

```
/*
 * Copyright (c) 1988, 1990 Regents of the University of California.
 * All rights reserved.
 *
 *          @(#)radix.h          7.4a (Berkeley) 11/28/90
 */

/*
 * Common Indices
 */
struct radix_info {
          short     ri_b;                         /* bit offset; -1-index(netmask) */
          char      ri_bmask;                     /* node: mask for bit test*/
          u_char    ri_flags;                     /* enumerated next */
}
#define RNF_ROOT              1                   /* leaf is root leaf for tree */
#define RNF_ACTIVE            2                   /* This node is alive (for rtfree) */

/*
 * Radix search tree node layout.
 */

struct Radix_node {
          struct    radix_mask *rn_mklist;        /* indication a mask may apply */
          struct    radix_node *rn_p;             /* parent */
          struct    radix_info rn_ri;             /* bit number and mask, flags */
          int       rn_off;                       /* precomputed offset for byte test */
          struct    radix_node *rn_l;             /* progeny */
          struct    radix_node *rn_r;             /* progeny */
};

struct Radix_leaf {
          struct    radix_mask *rn_mklist;        /* our handle to the annotation */
          struct    radix_node *rn_p;             /* parent */
          struct    radix_info rn_ri;             /* bit number and mask, flags */
          caddr_t   rn_key;                       /* object of search */
          caddr_t   rn_mask;                      /* netmask, if present */
          struct    radix_node *rn_dupedkey;
};
/*
 * The actual radix node struct is defined
 * in terms of a structure containing a union with copious defines such as:
 */
#define rn_key rn_u.rn_leaf.rn_Key
#define rn_b      rn_ri.ri_b
```

```
/*
 * Annotations to tree concerning potential routes applying to subtrees.
 */

extern struct radix_mask {
        struct   radix_info rm_ri;           /* bit number and mask, flags */
        struct   radix_mask *rm_mklist;      /* more masks to try */
        caddr_t  rm_mask;                    /* the mask */
        int      rm_refs;                    /* # of references to this struct */
} *rn_mkfreelist;

struct radix_node *
rn_search(v, head)
        struct radix_node *head;
        register caddr_t v;
{
        register struct radix_node *x;

        for (x = head; x->rn_b >= 0;) {
                if (x->rn_bmask & v[x->rn_off])
                        x = x->rn_r;
                else
                        x = x->rn_l;
        }
        return x;
};
```

Appendix B: Header Files for routing mesages, structures.

This also is a slightly simplified version of the actual header file:

```
/*
 *  Copyright (c) 1980, 1990 Regents of the University of California.
 *  All rights reserved.
 *
 *         @(#)route.h            7.12a (Berkeley) 11/28/90
 */

/*
 * These numbers are used by reliable protocols for determining
 * retransmission behavior and are included in the routing structure.
 */
struct rt_metrics {
        u_long   rmx_locks;          /* Kernel must leave these values alone */
        u_long   rmx_mtu;            /* MTU for this path */
        u_long   rmx_hopcount;       /* max hops expected */
        u_long   rmx_expire;         /* lifetime for route, e.g. redirect */
        u_long   rmx_recvpipe;       /* inbound delay–bandwith product */
        u_long   rmx_sendpipe;       /* outbound delay–bandwith product */
        u_long   rmx_ssthresh;       /* outbound gateway buffer limit */
        u_long   rmx_rtt;            /* estimated round trip time */
        u_long   rmx_rttvar;         /* estimated rtt variance */
};
/*
 * Bits for locking and initializing metrics
 */
#define RTV_MTU          0x1       /* init or lock _mtu */
#define RTV_HOPCOUNT     0x2       /* init or lock _hopcount */
#define RTV_EXPIRE       0x4       /* init or lock _hopcount */
#define RTV_RPIPE        0x8       /* init or lock _recvpipe */
#define RTV_SPIPE        0x10      /* init or lock _sendpipe */
#define RTV_SSTHRESH     0x20      /* init or lock _ssthresh */
#define RTV_RTT          0x40      /* init or lock _rtt */
#define RTV_RTTVAR       0x80      /* init or lock _rttvar */

struct rtentry {
        struct   radix_node rt_nodes[2];        /* tree glue, and other values */
#define rt_key(r)        ((struct sockaddr *)((r)–>rt_nodes–>rn_key))
#define rt_mask(r)       ((struct sockaddr *)((r)–>rt_nodes–>rn_mask))
        struct   sockaddr *rt_gateway;          /* value */
        short    rt_flags;                      /* up/down?, host/net */
        short    rt_refcnt;                     /* # held references */
        u_long   rt_use;                        /* raw # packets forwarded */
        struct   ifnet *rt_ifp;                 /* the answer: interface to use */
        struct   ifaddr *rt_ifa;                /* the answer: interface to use */
        struct   sockaddr *rt_genmask;          /* for generation of cloned routes */
        caddr_t  rt_llinfo;                     /* pointer to link level info cache */
        struct   rt_metrics rt_rmx;             /* metrics used by rx´ing protocols */
        short    rt_idle;                       /* easy to tell llayer still live */
};
```

```
/*
 * Flags
 */
#define RTF_UP          0x1     /* route useable */
#define RTF_GATEWAY     0x2     /* destination is a gateway */
#define RTF_HOST        0x4     /* host entry (net otherwise) */
#define RTF_REJECT      0x8     /* host or net unreachable */
#define RTF_DYNAMIC     0x10    /* created dynamically (by redirect) */
#define RTF_MODIFIED    0x20    /* modified dynamically (by redirect) */
#define RTF_DONE        0x40    /* message confirmed */
#define RTF_MASK        0x80    /* subnet mask present */
#define RTF_CLONING     0x100   /* generate new routes on use */
#define RTF_XRESOLVE    0x200   /* external daemon resolves name */
#define RTF_LLINFO      0x400   /* generated by ARP or ESIS */
/*
 * Structures for routing messages.
 */
struct rt_msghdr {
        u_short rtm_msglen;     /* to skip over non–understood messages */
        u_char  rtm_version;    /* future binary compatability */
        u_char  rtm_type;       /* message type */
        u_short rtm_index;      /* index for associated ifp */
        pid_t   rtm_pid;        /* identify sender */
        int     rtm_addrs;      /* bitmask identifying sockaddrs in msg */
        int     rtm_seq;        /* for sender to identify action */
        int     rtm_errno;      /* why failed */
        int     rtm_flags;      /* flags, incl. kern & message, e.g. DONE */
        int     rtm_use;        /* from rtentry */
        u_long  rtm_inits;      /* which metrics we are initializing */
        struct  rt_metrics rtm_rmx; /* metrics themselves */
};
/*
 * Message Types
 */
#define RTM_ADD         0x1     /* Add Route */
#define RTM_DELETE      0x2     /* Delete Route */
#define RTM_CHANGE      0x3     /* Change Metrics or flags */
#define RTM_GET         0x4     /* Report Metrics */
#define RTM_LOSING      0x5     /* Kernel Suspects Partitioning */
#define RTM_REDIRECT    0x6     /* Told to use different route */
#define RTM_MISS        0x7     /* Lookup failed on this address */
#define RTM_LOCK        0x8     /* fix specified metrics */
#define RTM_OLDADD      0x9     /* caused by SIOCADDRT */
#define RTM_OLDDEL      0xa     /* caused by SIOCDELRT */
#define RTM_RESOLVE     0xb     /* req to resolve dst to LL addr */
/*
 * Bits for identifying trailing or optional sockaddrs.
 */
#define RTA_DST         0x1     /* destination sockaddr present */
#define RTA_GATEWAY     0x2     /* gateway sockaddr present */
#define RTA_NETMASK     0x4     /* netmask sockaddr present */
#define RTA_GENMASK     0x8     /* cloning mask sockaddr present */
```

```
#define RTA_IFP         0x10    /* interface name sockaddr present */
#define RTA_IFA         0x20    /* interface addr sockaddr present */
#define RTA_AUTHOR      0x40    /* sockaddr for author of redirect */
```