Chapter 6

Routing

Communicating over a network means sending data from one computer to another. This sounds simple, and it is…when only two computers are involved…when those two computers are alike…when those two computers are physically near one another…and especially when the equipment doesn't fail.

Alas, few if any of these conditions are present on the Internet. Indeed, when an individual user of the Internet communicates with a popular server, the connection can involve dozens of intermediate computers, each with its own name, IP address, and designated duration. If you're curious about the number, you can use the **traceroute(1)** utility on your Linux computer (or the **tracert** program in a DOS window on a Microsoft system) to find out exactly how many other systems lie between you and a given destination site.

The **traceroute(1)** utility uses the Internet Control Message Protocol (ICMP) facility of the Internet Protocol (IP) to determine how packets are transferred through the network, and to measure the transmission time required for each step of the journey.

For example, we often communicate with **www.linux.org**, starting from our connection to Nevada Bell's Internet site (**nvbell.com**). The following report (produced by the Windows **tracert** program) shows the path followed by our data packets, and the stops our packets make along that path, during their journey from our office to the server that hosts the Linux Home Page:

```
Tracing route to www.linux.org [198.182.196.56]
over 14 hops:
  1    33 ms    18 ms    19 ms
     adsl-216-101-13-254.dsl.renocs.pacbell.net
     [216.101.13.254]
  2    18 ms    19 ms    19 ms
     ign1-e45.renocs.nvbell.net
     [206.171.130.249]
  3    21 ms    19 ms    19 ms
     ign0-f00.renocs.nvbell.net
     [206.13.6.194]
  4    37 ms    41 ms    32 ms
     sfra1sr2-3-4.ca.us.ibm.net
     [165.87.225.30]
  5    29 ms    30 ms    32 ms
     165.87.160.193
  6    31 ms    29 ms    28 ms
     114.ATM3-0.XR1.SFO1.ALTER.NET
     [146.188.148.210]
  7    32 ms    34 ms    27 ms
     187.ATM2-0.TR1.SCL1.ALTER.NET
     [146.188.147.146]
  8    95 ms    98 ms   105 ms
     107.ATM6-0.TR1.DCA1.ALTER.NET
     [146.188.136.221]
  9    94 ms    94 ms   101 ms
     299.ATM6-0.XR1.TCO1.ALTER.NET
     [146.188.161.169]
 10   101 ms    91 ms    92 ms
     193.ATM9-0-0.GW2.TCO1.ALTER.NET
     [146.188.160.57]
 11   102 ms    96 ms    96 ms
     uu-peer.pos-4-oc12-core.ai.net
     [205.134.160.2]
 12   111 ms   112 ms   113 ms
     border-ai.invlogic.com
     [205.134.175.254]
 13   123 ms   109 ms   113 ms
     router.invlogic.com
     [198.182.196.1]
 14   158 ms   115 ms   165 ms
     www.linux.org [198.182.196.56]

Trace complete.
```

The report tells us that our data packets made 14 hops, starting at the end of our DSL link located at Nevada Bell's central office. The report then gives us information about each of the 14 hops: first, the round-trip duration (in milliseconds) of each of the three probes that the **tracert** program sends to each node in the series, and then the name (if available) and the IP address of each of those nodes.

Where did this arrangement come from, and why is it so complex? To answer these questions, we need to take a brief look at the history of distributed network computing.

The Legacy Of The ARPAnet

The granddaddy of today's Internet was an experimental communications system—the Advanced Research Projects Agency Network, or ARPAnet—conceived in the 1960s at the RAND Corporation on behalf of the U.S. Department of Defense (DOD), and launched in 1971 with four sites in the western United States. By 1976, the projected number of nodes in the ARPAnet had grown to an unmanageable figure (255, to be precise), and a new way to connect them had to be found. Enter the *router*, which enabled the ARPAnet to double in size over the next few years, acquiring 200 host machines by 1981. In 1985, the ARPAnet broke the thousand-host barrier. By 1989, the experiment was over, and the ARPAnet as a system ceased to exist…leaving more than 100,000 machines interconnected via something called the "Internet."

Because of the circumstances envisioned for its use, the ARPAnet was required to heal itself whenever any single part of the network failed. (One type of failure contemplated by the DOD was the physical destruction of the ARPAnet's computers and communications links by "nuclear events.") This requirement set the ARPAnet apart from the commercial computer networks of the day, which were centrally controlled. To prevent a single point of failure from bringing down the entire communications system, the ARPAnet's controlling functions had to be distributed, as evenly as possible, among computers physically located many miles apart.

Obviously, the number of possible failure points was huge. Moreover, because any given computer could, in theory, fail at any given time, the ARPAnet's designers also faced the daunting task of dealing with random changes in the topology of the network. These changes, the designers theorized, would be caused by nodes and links that not only disappeared, but, in the fashion of some subatomic particles, also *appeared* utterly unexpectedly and unpredictably—which, as it turned out, is exactly what happened.

The ARPAnet's protocol design was continually revised, based on an analysis of the hardware, telecommunications, and software failures that occurred. The changes that were made allowed the ARPAnet to cope with most failures automatically, finding alternate routes to keep data flowing even if several nodes or links went down. These workarounds are still part of the Internet today.

Here's how the ARPAnet's workarounds worked. Each connection node on the network, known as an *interface message processor (IMP)*, kept a living record of neighboring nodes and traded information with those nodes at regular intervals. The sum of each IMP's knowledge—including the list of neighboring nodes and a history of the information exchanged with those nodes—was stored in a dynamic structure known as a *routing table*. (Routing tables are still present, playing a more important role than ever, in today's Internet hosts. Entire books have been written about their design and use, and an in-depth discussion is beyond the scope of this book.)

From a cold standing start, all the nodes on the ARPAnet needed only a few minutes to learn enough about the network's topology to be able to send data to the proper destination. As the nodes built a history and exchanged data with each other, they "learned" the best data-transmission routes. In less than an hour, the nodes learned so much about ARPAnet routing that, when a glitch occurred (and glitches could range from a momentary interruption of transmission to the complete shutdown of a node), users rarely noticed it.

This learn-as-you-go technique is what the Internet uses today. As an example, consider how this technique helps your data get from your machine to a server located 12,000 miles away.

Before any data goes anywhere, it has to be properly "gift-wrapped." In the Internet, the bow-adorned object is an IP packet, which consists of a box with data inside, wrapping paper to ensure that the data stays together, and a tag that states who is sending the packet and who is supposed to receive it. Unlike the typical recipient of a holiday gift, the intended recipient of an IP packet can be miles or continents away...and the IP tag tells you absolutely nothing about how to get the package to the right tree.

In a well-known experiment done in the 1960s, researchers found that a letter, addressed (by name only) to a randomly selected individual and entrusted to another randomly selected individual, reached the addressee after a remarkably small number of person-to-person transfers. (In fact, the researchers concluded that any living person could theoretically reach any other living person in six or fewer "hops.") The purpose of the experiment was to show how many people each human being knows directly and, by concatenation and extension, how interconnected all of humanity is.

However, computers aren't people, so the designers of the ARPAnet had to devise a less complex way to accomplish the same task; that is, determining a path between two random points without the help of a master directory. When the ARPAnet was small (with fewer than 64 nodes), each IMP kept a record of how far away and in what (logical) direction each host was located. As the ARPAnet evolved toward the Internet, network designers started to group hosts together, into *subnets*, to limit the amount of record-keeping data to a manageable size. Thanks to subnets, the amount of data that a pair of routers had to transfer to one another was also kept to a reasonable size. Today's Internet routers still operate this way, working in layers, so that each router's task is relatively small—even if the number of hosts exceeds a billion. "Divide and conquer" could well be the Internet's motto.

Routing: What Makes The Internet The Internet

At its most elementary level, routing function means switching data through a series of ports located in a single physical unit, so that the data gets from its source to its destination. For example, the routing box shown in Figure 6.1 has a total of nine ports: eight ports connected to remote computers or to other routing boxes, and one local connection within itself.

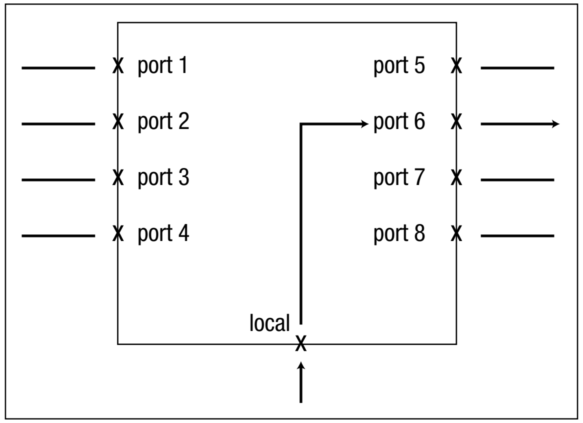

Figure 6.1 Block diagram view of a router.

How does this router work? Assume that it receives a data packet at its local port. The routing software acquires the packet's destination address and, using the information stored in the router's internal routing tables, determines the best port from which the data packet should be sent.

In Figure 6.2, the routing software has determined that for the data packet to reach its destination efficiently, the packet should leave the router via Port 6.

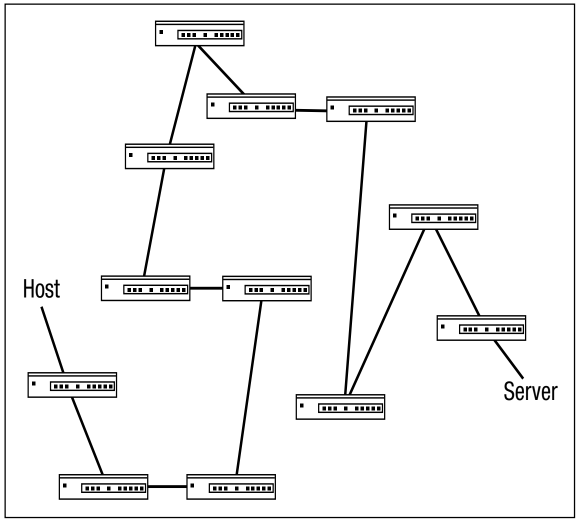

Figure 6.2 Schematic of router with data path for a specific packet.

Routers do not have an actual physical address to which they send data packets that are supposed to be discarded (for example, data packets for which no route can be found, or whose Time To Live (TTL) value has reached 0). However, a Linux kernel routine allows such doomed data packets to be removed from a router's memory, in an operation known familiarly as "sending the data to the bit-bucket."

A router follows the same steps each time it performs the analysis and port-assignment process. So, why doesn't the router always send out data packets via the same port? Because the port that the router ultimately selects depends on the information *stored in the routing table*, not on any change in the way the router performs its analysis of that information.

In short, then, the act of routing embraces the entire process of getting a message from, for example, the small (and very real) town of Truth or Consequences, New Mexico to its ultimate destination, the capital city of Ouagadougou, in the African nation of Burkina Faso.

The Data Packet Delivery Service

The router code that is the subject of this chapter uses information about the network (as collected by the IP and ICMP protocol handlers and by other node-resident processes that exchange information with neighboring nodes) to decide how to handle a data packet that is being "sent." However, in the router world, a data packet's destination is not necessarily a physically different machine. An internal process within a router is just as valid as an external destination, and routers often "send" packets to such internal processes.

The routing table is kept up to date by a set of pick-a-little talk-a-little router-to-router protocols. These protocols, which are implemented as *daemons* (programs that live in the computer, but that are not part of the kernel), gossip continually with each other. By exchanging information this way, and especially by discarding obsolete information about traffic conditions and external nodes, they make the kernel code's task much easier. The kernel code, which uses the routing table several hundred times a second when data is being transferred, is appropriately grateful.

Back to your gift-wrapped data package. The IP packet starts out from its point of origin in Truth or Consequences, New Mexico. The first router that your packet reaches on the Internet contains a collection of hints in its routing table, gathered from its neighboring routers, that suggests the best way to forward a packet toward Ouagadougou. (In the data-transmission world, "best way" may mean any of several things. It might be either the fastest way to send information, the method that's cheapest in terms of resources or money, the route that's the least sensitive politically, or the pathway that's the least likely to damage the data.)

The first router sends your package to a second router, which, with luck, is located well toward your package's destination. This second router then makes its best guess about how to pass the data, and speeds the package on its way toward the next router. The process continues until the package reaches its destination in Ouagadougou and the recipient opens it. In the very worst case, the package never gets to Ouagadougou. Instead, it hits a digital dead-end and is dumped unceremoniously into a bit-bucket.

In an ideal world, your package would travel as directly as possible to its destination. However, the data-transmission world is far from ideal. Just as on city streets or at airports, traffic congestion can affect a router's forwarding decision, such that packages are sent on detours around the slow or stacked-up areas. Quite conceivably, your Africa-bound package could arrive in New York City or Casablanca, only to be turned around and routed back westward.

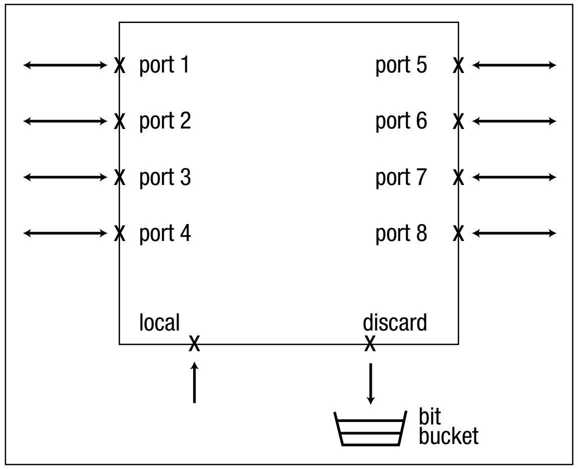

Figure 6.3 An example of the geographical progression of a packet route. This route depicts the physical path taken by a packet on a college campus circa 1987.

Or not. Sometimes, a router doesn't know that a path is congested, and forwards an IP packet into a gridlock anyway. Remember, in selecting the outgoing path for a packet, a router bases its decision on the information it obtains from neighboring routers. When traffic conditions change, routers have no way of learning about the changes instantaneously. It takes time for the word to propagate outward from a congested node or link, and, in the meantime, a packet could arrive at a router and be sent onward, in all innocence, into trouble.

At its most frustrating, this slow spread of information among routers can cause an IP packet to travel in circles—in other words, in an *endless loop*. An endlessly looping packet eventually self-destructs, dying of old age. Unlike the U.S. Postal Service, the Internet has no Dead Letter Office for defunct datagrams. Instead, looping IP packets evaporate without a trace, considerately relieving the Internet of the need to dispose of them. (This wasn't always the case. In the early 1970s, the ARPAnet was sometimes very, very busy, but no data was getting through to users. The activity level was caused by packets that had invalid host addresses and therefore stayed alive—and undelivered—indefinitely in the network, like a subway passenger caught short by a fare increase, doomed to ride forever beneath the streets of Boston—the datagram that never returned. That's when the network designers gave packets a finite lifespan.)

The routing tables are implemented in Linux as linked lists, and the elements of the list are defined in the structure described at line 41260. Although the tables are updated once or twice a minute, they are referenced many, many times—on a busy system, hundreds of times per second—by routing protocols, by certain free-running processes, and, most frequently of all, by the IP module in the kernel code.

The IP module's job is to handle packet routing requests. Each packet is associated with its own routing request, which means that the IP module may have to process as many as 150,000 requests per second. Unfortunately, one routing request does not necessarily imply only one pass through the routing table. To handle a single routing request, for a single packet, the IP module may need to make several trips through the routing table. First, it looks for a specific host entry. If the host entry appears in the routing table, the IP module uses the designated port associated with that host. If the desired host doesn't appear in the list, then the IP module searches for the name of a subnet that contains the host. If the IP module does find such a subnet, it sends the data out through the designated port for that subnet. If it doesn't find the right subnet, then it looks for a default port to which to send the data. If it finds the name of a default port, it sends the data to that port. Otherwise, it returns an error message and takes the appropriate action, depending on the source of the routing request.

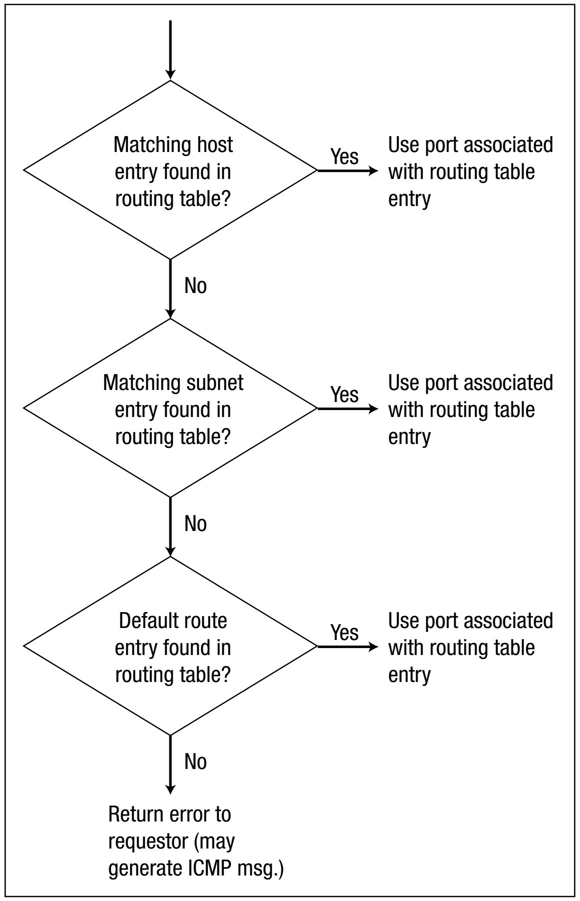
Matching host entry found in routing table?
Yes
Use port associated with routing table entry
No
Matching subnet entry found in routing table?
Yes
Use port associated with routing table entry
No
Default route entry found in routing table?
Yes
Use port associated with routing table entry
No
Return error to requestor (may generate ICMP msg.)

Figure 6.4 The routing decision tree.

Based on the information stored in the routing table, a data packet is directed to a specific interface on the computer. In the trivial case (when a router box has only one outgoing port), the packet is sent to a port (such as a modem or an Ethernet port) that leads to the outside world. When a router has two or more ports to the outside world, the decision process becomes more complex, but the result is the same: the IP module decides which port gets the packet.

For an idea of the amount of work a router does, consider a digital DS3 trunk operating at a speed of 44megabits/second (Mbps). A router connected to such a trunk handles from 60,000 to 150,000 packets per second, virtually nonstop. In other words, this router has six whole microseconds (6/1,000ths of a second) to redirect each packet that comes in through any single port. If the redirection takes longer, the whole stream of packets may slow down, or even stop.

A T1 or ADSL (asymmetric digital subscriber line) channel is much slower, pouring in only 2,000 to 5,300 packets/second, whereas a 56-kilobyte/s (Kbps) modem or frame relay connection drizzles just 75 to 180 packets/second into the router. But no matter how fast or slowly the packets come in, the time required to make a routing decision directly affects a router's performance, even if the router lives in a single system. Indeed, packets can be lost when input buffers overflow, which can happen when a router's decisions are too slow. Fortunately, as you'll see in the commentary section of this chapter, the Linux code contains some time-saving tricks that keep the routing process from becoming a data bottleneck.

Of Packets And Checks

Messages travel through the Internet much the same way that checks flow through the commercial banking system. From the bottom up:

- Individual Internet surfers are analogous to businesses depositing checks that they've received from customers.

- Companies whose networks are completely internal (the *intranets*, used only by company employees and authorized guests) are analogous to small banks.
- A typical regional ISP is analogous to a standard bank.
- The *upstream provider*s are analogous to bigger banks.
- The Internet has several nationwide ISPs, each with its own network, which are analogous to the multistate megabanks.
- At the very top of the heap, the similarity diverges slightly. Instead of a single analog of the U.S. Federal Reserve System, the Internet has several *backbone providers*, in the form of the companies who run the large networks that link the various regions of the United States. However, like the Fed, these companies are the lords of all they survey.

Checks fly in and out of banks in all directions. A few checks are deposited at a bank against accounts at that same bank—that's the easy case and is handled completely in-house. The vast majority of checks, though, are drawn on one bank and deposited at another. They come in through teller windows and ATM machines; from correspondent banks; and (for member banks) from the Fed itself. The incoming checks drawn on accounts at the bank itself are separated and processed, and then the checks drawn on other banks are sorted and sent on their way.

Analogously, data packets fly in and out of IP routers through a number of ports. When you log on to the Internet, your system receives packets confirming or denying your access. Those packets are sent to, and stay in, processes that live in your computer system. However, the great majority of packets that are sent out from your computer, via your modem or broadband link to the Internet, are processed at another system—usually a server of some kind—living at your ISP. A router box located at your ISP, at the upstream provider, or in the backbone can, in theory, have as few as three ports; however, most of these routers have several hundred ports through which incoming packets arrive and reshuffled packets depart.

What makes the check-clearing analogy to the Internet so remarkable is that the decisions that govern how checks are routed are exactly the same decisions that are used to determine how data packets are steered. In the banking system, a check comes in and, based on its ABA/FRD (American Banking Association/Federal Reserve District) number, also known as the "routing number," the bank decides where the check goes next (usually to another bank). On the Internet, a data packet comes in and, based on the Internet address in the packet header, the router decides where the packet goes next (usually to another router). In the banking system, the final decision is which truck a bag of checks should be tossed into. In an Internet router, the "ultimate decision" is the one that determines the I/O port to which the packet should be directed.

The major difference between the Internet and the U.S. check-clearing system lies in the acquisition of the information on which routing decisions are based. Relationships between banks are essentially static. A "Grand Opening" here or an unexpected bank failure there is a relatively small change, and is managed pretty much by exception. And in the banking world, mergers and takeovers don't "just happen." They're planned long in advance and often are heralded widely. In the online world, though, the network is constantly changing, with no fanfare at all. The beauty of the Internet is that it makes adjustments—automatically and continuously—that the banking system just couldn't handle.

Another difference between the banking system and the Internet is that every bundle of checks has to be accompanied by a "cash letter" that indicates the dollar value of each check in the bundle. In the Internet, as implemented in the United States, there is no accounting of any kind. That's not true in some other countries, where information transfers are billed by the kilopacket. In such cases, "least-cost routing" has a real monetary value, and mistakes can cost someone a bundle.

The Internet's paramount feature—flexibility—also makes it fast. For instance, if a courier company's truck is full, banks and the Fed won't ship a bag of checks on a different truck on a different route. Instead, the bag waits for the next truck on the usual route. In contrast, many Internet routers don't "wait for the next truck." Instead, they divert data packets to paths that are open (albeit sometimes basing that decision on the cost of the alternate paths, in terms of time or reliability).

This *dynamic routing*—with parts of a message being forwarded over different and sometimes wildly divergent routes—can cause packets in a TCP stream, or fragments of a UDP packet, to arrive at their destination in wildly scrambled order. Putting packets (and sometimes fragments of packets) back into their original, comprehensible order takes time and computational effort. But, this effort is a small price to pay for coherent messages, even if some packets have to make detours, sometimes in rattly decrepit trucks over dusty back roads, to avoid fatal obstacles.

This matter of packet reassembly comes up again, with renewed significance, in connection with the IPv4 and TCP protocols, as you'll see in the following chapters. For now, though, the next section examines the Linux router code in detail.

Routing In Linux: The Routing-Table Handling Routines

The routines in the source module route.c appear in arbitrary order and are not organized with any particular logic. To help you locate the modules in the code listings, Table 6.1 and Table 6.2 list the functions in the source module route.c.

Table 6.1 Index of functions, sorted by line number.

Number	Function
19680	**rt_logmask**
19691	**rt_mask**
19698	**fz_hash_code**

(continued)

Table 6.1 Index of functions, sorted by line number *(continued)*.

Number	Function
19708	**fib_free_node**
19732	**fib_lookup_gateway**
19778	**fib_lookup_local**
19828	**fib_lookup**
19855	**get_gw_dev**
19868	**bad_mask**
19879	**fib_del_list**
19916	**fib_del_1**
19966	**fib_create_info**
20040	**fib_add_1**
20246	**rt_flush_list**
20274	**fib_flush_1**
20322	**rt_get_info**
20421	**rt_cache_get_info**
20490	**rt_free**
20521	**rt_kick_free_queue**
20555	**ip_rt_run_bh**
20582	**ip_rt_check_expire**
20655	**rt_redirect_1**
20688	**rt_cache_flush**
20730	**rt_garbage_collect_1**
20764	**rt_req_enqueue**
20789	**rt_req_dequeue**
20810	**rt_kick_backlog**
20838	**rt_del**
20854	**rt_add**
20868	**ip_rt_flush**
20882	**ip_rt_redirect**
20923	**rt_garbage_collect**
20933	**rt_cache_add**
21036	**ip_rt_slow_route**
21161	**ip_rt_put**
21181	**ip_rt_dev**
21190	**ip_rt_route**
21222	**ip_rt_new**
21353	**ip_rt_kill**
21393	**ip_rt_ioctl**
21417	**ip_rt_advice**
21423	**ip_rt_update**

Table 6.2 Index of functions, listed alphabetically by name.

Number	Function
19868	**bad_mask**
20040	**fib_add_1**
19966	**fib_create_info**
19916	**fib_del_1**
19879	**fib_del_list**
20274	**fib_flush_1**
19708	**fib_free_node**
19828	**fib_lookup**
19732	**fib_lookup_gateway**
19778	**fib_lookup_local**
19698	**fz_hash_code**
19855	**get_gw_dev**
21417	**ip_rt_advice**
20582	**ip_rt_check_expire**
21181	**ip_rt_dev**
20868	**ip_rt_flush**
21393	**ip_rt_ioctl**
21353	**ip_rt_kill**
21222	**ip_rt_new**
21161	**ip_rt_put**
20882	**ip_rt_redirect**
21190	**ip_rt_route**
20555	**ip_rt_run_bh**
21036	**ip_rt_slow_route**
21423	**ip_rt_update**
20854	**rt_add**
20933	**rt_cache_add**
20688	**rt_cache_flush**
20421	**rt_cache_get_info**
20838	**rt_del**
20246	**rt_flush_list**
20490	**rt_free**
20923	**rt_garbage_collect**
20730	**rt_garbage_collect_1**
20322	**rt_get_info**
20810	**rt_kick_backlog**
20521	**rt_kick_free_queue**
19680	**rt_logmask**
19691	**rt_mask**

(continued)

Table 6.2 Index of functions, listed alphabetically by name *(continued)*.

Number	Function
20655	**rt_redirect_1**
20789	**rt_req_dequeue**
20764	**rt_req_enqueue**

Routing Table Structure

The routing table consists of a series of elements in a singly-linked list. The definition of the structure starts on line 41260. The following list describes the most important fields. (Other fields exist that are used internally by the router code, and yet other fields exist that are commonly used in routers, but not in the Linux implementation.)

41262: **rt_next**, the link to the next routing-table entry.

41263: **rt_dst**, the host or network address described by this entry.

41265: **rt_gateway**, the gateway address (the recommended immediate path to the destination address contained in **rt_dst**).

41266: **rt_refcnt**, the reference counter. When decremented to 0, this routing-table entry becomes a candidate for removal during garbage collection, so that the memory space used by the routing-table entry element can be recovered.

41268: **rt_window**, the size of the TCP window that should be used for the destination described by this routing-table entry.

41271: **rt_dev**, the pointer to the name (a null-terminated ASCII string) of the device to be used to reach the destination described by this routing-table entry.

41272: **rt_flags**, the flags for this routing-table entry. The flags themselves are described separately, in the next section.

41273: **rt_mtu**, the maximum transmission unit (MTU) size, in bytes, from this host/router to the destination described by this routing-table entry.

41274: **rt_irtt**, the initial round-trip delay time that TCP shall assume exists between this host and the destination described by this routing-table entry.

The following bit flags may appear in the value contained in **rt_flags**:

38704: **RTF_UP**, which, when set, indicates that the route is usable.

38706: **RTF_GATEWAY**, indicating that the destination contained in **rt_dst** is a gateway (rather than a network or host).

38708: **RTF_HOST**, indicating that the destination contained in **rt_dst** is a host (not a network) with a possible gateway.

38712: **RTF_DYNAMIC**, indicating that this routing-table entry was created by an ICMP redirect packet received by the router/host.

38714: **RTF_MODIFIED**, indicating that this routing-table entry was modified dynamically by an ICMP redirect packet received by the router/host.

38716: **RTF_MSS**, which instructs the code to use the maximum segment size contained in **rt_mtu** for this route.

38718: **RTF_WINDOW**, which instructs the code to use the per-route window size maximum contained in **rt_window** for this route.

38720: **RTF_IRTT**, which instructs the code to use the initial round-trip time contained in **rt_irtt** for this route.

38722: **RTF_REJECT**, indicating that the route is a “reject” route. If the search finds a match on this route, the assumption should be made that no route exists. This flag is used to block transfers involving port-to-private-network addresses (and netmasks), such as 10.0.0.0 (255.0.0.0), 172.16.0.0 (255.240.0.0), and 192.168.0.0 (255.255.0.0).

38724: **RTF_NOTCACHED**, indicating that this route is not cached.

These same flags are used in the **rt_entry** structure, defined at line 38677, in the structure member **rt_flags** (line 38687).

The address of an instance of this structure is specified in the **argp** parameter. This parameter is copied from the corresponding parameter **ioctl** calls that specify the function requests **SIOCADDRT** (add route, at line 39518) and **SIOCDELRT** (delete route, at line 39520).

Although no explicit limit exists on the number of entries that a routing table can contain, a long list naturally takes longer to traverse than a short one.

/proc File System Support

In the Linux environment, users and many utility programs get information about the internal state of the system through the proc file system. This is a pseudo-file system, usually accessible through the directory /proc, that lets a user (via the shell and the **cat** utility) or an application (via standard file I/O) read the current condition of various internal processes.

The file system “files” are actually the output of executable routines, such as the ones in route.c that we will be reviewing, that report on the internal state of the system. In other portions of Linux, the “files” also accept data; as we’ll see in other portions of the TCP/IP stack.

The routine **rt_get_info** is accessed via a pointer reference at line 6317, which in turn is referenced by the source file /usr/src/linux/fs/proc/net.c, which is a portion of the file system source (not included in this book). It outputs the contents of the main routing table. The routine outputs a header line followed by a line for each routing-table entry. Here is a sample of the output (with each physical line of output shown in the form of three lines):

```
Iface  Destination Gateway     Flags  RefCnt
Use  Metric   Mask         MTU    Window IRTT

eth0   0001010A    00000000    01     0
10   0        00FFFFFF     1500   0      0

lo     0000007F    00000000    01     0
5    0        000000FF     3584   0      0
```

rt_cache_get_info is accessed via a pointer reference at line 6323. The routine outputs a header line followed by a line for each routing-table cache entry, as in this example (in which each physical line of output is shown in the form of three lines):

```
Iface  Destination  Gateway      Flags  RefCnt
Use    Metric Source       MTU   Window IRTT
HH     ARP

eth0   FF01010A     FF01010A     05     0
1      0      1501010A     1500  0      0
1      1

eth0   1401010A     1401010A     05     0
1      0      1501010A     1500  0      0
2      1

lo     1501010A     1501010A     05     1
2      0      1501010A     3584  0      0
-1     0

eth0   1601010A     1601010A     05     0
1      0      1501010A     1500  0      0
2      1

eth0   1701010A     1701010A     05     0
1      0      1501010A     1500  0      0
2      0

eth0   8A01010A     8A01010A     05     0
1      0      1501010A     1500  0      0
2      1
```

Later in this chapter , you'll see where each of these fields comes from.

To generate the routing-table cache shown here, we polled every computer on a small network and then immediately requested the dump just shown. After a few minutes, we asked for another cache dump. The result of the second request is shown here:

```
Iface  Destination  Gateway      Flags  RefCnt
Use    Metric Source        MTU   Window IRTT
HH     ARP

10     0100007F     0100007F     05     1
9      0      0100007F     3584  0      0
-1     0
```

All the entries we had generated earlier were cleared out of the cache very quickly. This speedy disposal makes sense, because the purpose of the cache is to speed up routing for active connections.

The formats for the output of these two pseudofiles are identical. The information in the lines forms a columnar database, with tab (\t) characters separating the fields. A database record is complete when the program sees a newline (\n) character or an end-of-file condition. Because this data is intended to be read and manipulated by programs, the formatting of entries that are longer than the default tab width of the terminal (or other device, or printing protocol) becomes an issue. For example, in the caching table, you see "Metric Source"—that's two separate columns, not a two-word description of a single column.

rt_get_info

This function is called multiple times. Based on the parameter **offset**, the function returns information in sequence about the routing table. The first time the function is called, the value of **offset** should be 0. Each subsequent time, the value of **offset** is incremented by the prior value of the offset plus the length of the information returned during the prior call. This way, the file system routines can return a "buffer" of information for each call, without having to save significant state information for each process. (This technique also plugs a potential source of memory leaks.)

Although this technique increases the amount of processing to be performed, and also increases the risk of garbled data due to changes to the table that can occur between calls, it reduces the risk of other problems. For example, if you use the **cat** utility with a pipe to the **more** utility, early termination of the **more** utility will cause a "broken pipe" condition, which may prevent the rest of the output from ever being read.

20334: Output the header line to the buffer, if appropriate. Note that the **sprintf** call pads the output to 128 characters (including the new line), regardless of the actual length of the header.

20343: If necessary, wait on the **ip_rt_lock** semaphore.

20347: Go through the internal list of Forward Information Block (FIB) elements, one by one, from beginning to end.

20352: If no entries are associated with the current FIB element, then go to the next FIB element and try again.

20355: If the current FIB element is associated with a line that has already been printed, advance the virtual line pointer, go to the next FIB element, and try again.

20362: If a hash table is present for the current FIB element, then set the number of slots to the size of the hash table and point to the first list pointer. Otherwise, set the number of slots to 1 and position the pointer at the start of the list.

20373: Walk through the hash table (either one element or **RTZ_HASH_DIVISOR** elements).

20376: Walk through the current routing-table list.

20385: If this FIB has been output already, skip the entry.

20391: Prepare the line of information describing the route. (The correspondence between the column label printed and the data field references is shown in Table 6.3.) The variable **f** is a pointer to the current block, while the variable **fz** points to the current zone; that is, to the current set of routes having a specific number of one-bits in the mask. (In the code, the variable **fi** is set to the pointer expression **f->fib_info**. The expression is expanded in the table, to clarify the path to the data.)

Table 6.3 Column heads and corresponding data source.

Label	Data Source	Data Type
Iface	**f->fib_info->fib_dev->name**	char[]
Destination	**f->fib_dst**	unsigned long
Gateway	**f->fib_info->fib_gateway**	unsigned long
Flags	**f->fib_info->fib_flags**	unsigned short
RefCnt	(zero)	—
Use	**f->fib_use**	unsigned long
Metric	**f->fib_metric**	short
Mask	**fz->fz_mask**	unsigned long
MTU	**f->fib_info->fib_mtu**	unsigned short
Window	**f->fib_info->fib_window**	unsigned long
IRTT	**f->fib_info->fib_irtt**	unsigned short

20401: Take the prepared information (written to the temporary buffer) and output it in the form of a 128-byte line.

20403: Increment the position by 128 bytes. If the buffer passed by the caller is full, then break out of the loop prematurely (by branching to the program label **done**). The **goto done** instruction is the cleanest way to break out of all three loops at once, even if it makes structured-programming purists wince.

20406: These right braces close the loops started in line 20376, line 20373, and line 20347.

20410: The definition for the program label **done**.

20411: Free the lock and restart any processes that were waiting on the lock.

20414: Calculate and send back to the caller the starting offset for the next call, and return the length of the new data in the buffer.

rt_cache_get_info

This function is considerably simpler than the **rt_get_info** function, because the information is in a simple linked list. Each call returns a specified amount of information, with each call getting the next set of lines.

20432: If this is the first call, then write the column headers to the buffer.

20442: Wait for the semaphore **ip_rt_lock** to become available, and then lock it.

20446: Cycle through each of the cache table lists (the number of which is defined by the manifest constant **RT_HASH_DIVISOR**), taking the elements one at a time.

20455: If the element in question has been printed, then continue searching for the point at which to resume printing.

20461: Print the information for the cached route. (The correspondence between the column label, as printed, and the data-field references is shown in Table 6.4.) The variable **r** is a pointer to the current block

20471: Take the prepared information (written to the temporary buffer) and output it in the form of a 128-byte line.

20473: If the buffer is full, abort the loop (by branching to the program label **done**); otherwise, continue scanning the lists.

20475: The right braces close the list-traversing loop (line 20448) and hash-table scan loop (line 20446).

20479: Unlock the semaphore and restart any process that is waiting on this lock.

Table 6.4 Column heads and corresponding data source.

Label	Data Source	Data Type
Iface	**r->rt_dev->name**	string
Destination	**r->rt_dst**	unsigned long
Gateway	**r->rt_gateway**	unsigned long
Flags	**r->rt_flags**	unsigned short
RefCnt	**r->rt_refcnt**	atomic_t (int)
Use	**r->rt_use**	atomic_t (int)
Metric	(zero)	—
Source	**r->src**	unsigned long
MTU	**r->rt_mtu**	unsigned short
Window	**r->rt_window**	unsigned long
IRTT	**r->rt_irtt**	unsigned short
HH	**r->rt_hh->hh_refcnt**	int
ARP	**r->rt_hh->hh_uptodate**	char (Boolean)

20482: Calculate and return the offset for the next call; then, return the length of information for the buffer in this call.

The ioctl Handling Routine

External processes and some of the kernel routines talk to the routing system via the standard **ioctl** system call. This routine handles all requests for I/O control directed toward routes.

ip_rt_ioctl

This steering function takes a system call from a process and determines the proper routine for executing the request. This function also performs all necessary validation operations, so that memory isn't corrupted and no unexpected machine exceptions occur.

21393: This is the entry point called from the file /usr/src/linux/net/ipv4/af_inet.c (line 6079).

21398: This code selects one of the two permitted function requests for the **ioctl** call.

21402: The **ioctl** calls that affect routing-table entries must be made by a superuser process. This code checks for the superuser condition.

21404: To prevent panic (which can occur when a kernel routine causes a memory fault), the memory area used by the **ioctl** routine has to be checked for sanity.

21408: The routing-table information is copied from the process space to local memory (allocated on the stack at line 21396). This operation speeds processing immensely, and simply is a sensible thing to do.

21410: Depending on the function, a route is either killed or created/updated. For route creation, the function **ip_rt_new** is called. For route deletion, the function **ip_rt_kill** is called. In either case, when the applicable function is called, the local routing-table data is passed via a pointer, and the function's return value is propagated to the calling routine, from where it eventually returns to the process that started the entire operation.

Adding Routes To The Table

Before the routing system can be used, entries need to be placed in the routing table. This action is performed at system boot time, when the **route(8)** utility is used to install some basic routing information. The startup calls are located in various places, depending on the distribution of Linux being used. In Slackware, these calls are in the file /etc/rc.d/rc.inet1. In Red Hat, they're in /etc/sysconfig/network-scripts/ifup, with the parameters in /etc/sysconfig/network. Other distributions place the initial routes in other scripts.

Linux users on LANs will find routes for **localhost** and for the LAN itself. In many cases, the LAN will have a gateway to other networks, so the desired routes will be included in the static startup information. The system administrator enters these numbers manually when the Linux system is set up.

For Linux users who rely on dial-up modems and the Point-to-Point Protocol (PPP) for Internet access, the initial route at boot consists only of the **localhost** entry. The code that establishes PPP (or SLIP) connections then adds a route when the connection is made.

Boot time isn't the only time at which routes are added to the routing table. As you shall see (both later in this chapter and in Chapter 7), control messages can provide information that is placed in the routing table. In particular, *ICMP redirects* are used when an upstream router finds a route to a particular host that is better than the route provided in the default configuration. However, because ICMP redirects are a function of network complexity, you won't run into them unless you're running on a large intranet.

ip_rt_new

The **ip_rt_new** function creates a new route entry in the routing table. The new entry can replace (and, in practical terms, update) an existing routing entry.

21235: If the caller specified a device name in the structure **rt** (passed as **argp** in the **ioctl** call), this code block tries to find the device as named. The **getname** function is defined in the /usr/src/linux/fs/namei.c file.

21250: Check for the correct family value, which should be **AF_INET**.

21259: Copy the flags, target address, netmask, gateway address, and metric into local variables. This operation eliminates pointer dereferencing when these values are manipulated. (Note that the metric is decremented by 1 from the metric provided by the caller.)

21277: This code supports the case in which a gateway is being added to the table, but the gateway device is specified by IP address instead of by name. The code searches the device-block linked list for a working interface that has a matching IP address. The entries for the device-block linked list are defined by the **device** structure at line 38347. When a device is configured, the structure member **pa_addr** is filled with the IP address. If the address matches and the **IFF_UP** flag is also asserted, then a match has been found and the device name is used thereafter.

21293: If the **RTF_HOST** flag is set, this code overwrites the netmask for the request to all 1's.

21295: If a non-0 mask has been defined and if the socket isn't an **AF_INET** socket, then the balloon goes up and a "not-supported" return code is returned to the caller.

21300: Gateways are specific to Internet routes; so if this route isn't an Internet route, a "not-supported" return code is returned to the caller.

21308: Gateways are useless if you can't reach them. This code makes sure that the specified gateway is available.

21318: If this address is not a gateway (network or host), then zero out the gateway address. If no device was specified, then use the **ip_dev_bynet** function (line 8803) to find the device to use for this route. If no device can be found, then tell the caller that the designated network was unreachable. (We perform a routing operation whenever the caller doesn't tell us what he or she wants—even when we're creating routes.)

21323: If the caller didn't specify a mask, then copy the device's mask.

21332: The **ip_get_mask** function at line 8643 is used to set the mask to the correct mask for the address. This operation is performed if the working netmask is still 0 and if **CONFIG_IP_CLASSLESS** was not defined.

21336: After all of this work, the mask may still be ill-formed. If so, it must be rejected, and an invalid-request return code must be sent back to the caller. The inline function **bad_mask** confirms that the mask is acceptable.

21343: The information in the request to add a route entry has passed muster, so now we call the routine that actually does the job. The function **rt_add** doesn't return a status, so we return 0 to the caller to indicate that that the job's been properly done.

ip_rt_redirect

This routine is called by ICMP handlers that have to deal with redirect requests. They do so by creating a new route and then deleting the old one.

20888: If no route exists for the destination, then kick the call back (that is, return the call, accompanied by an error code).

20892: If the information is not identical, create a new route record and return. (This operation is not a true redirection, but rather the publication of a route.)

20900: Create the modified route.

20903: If the semaphore can be captured, then perform the redirection now. Otherwise, queue up a request for service by the backend handlers (described later in this chapter).

ip_rt_update

This function isn't implemented in Linux kernel release 2.0.34. However, our crystal ball says that this function will automatically add a route to the device when it comes up, and will remove all routes for the device when it goes down. These events currently take place through configuration scripts, rather than happening all by themselves.

ip_rt_advice

This function is called by the **tcp_write_timeout** function in the file tcp_timer.c (described in Chapter 9).

21419: This function does nothing. The comment may amuse readers who appreciate programmer humor.

ip_rt_put

This routine handles the details of eliminating a reference to a routing element.

21163: Decrement the reference count.

21170: If the element has not been cached and if the reference count goes to 0, release the element (which has already been unlinked from any lists).

bad_mask

The **bad_mask** function takes a network address and a netmask and determines whether the address and mask are acceptable. It returns 0 if the mask is acceptable (not a bad mask), and non-0 if the mask is unacceptable (bad mask).

19868: The function **bad_mask** is expanded inline when it is encountered, so there is no call and no parameter passing. However, because it is used so often, it should be coded only once. This way, code can be executed faster without any undue expansion in size. The function returns FALSE (zero) if the mask is acceptable and returns NOT FALSE (non-zero) if the mask is ill-formed for the address.

19870: The address in question and the subnet address (obtained by taking the one's complement of the netmask) must yield a non-0 address.

19872: This function appears often when code performs arithmetic on addresses and masks. The information about addresses is stored in "network order," which has the most significant byte in the leftmost position in byte-addressed storage. The Intel 8086 family of processors puts the most significant byte in the rightmost position in byte-addressed storage, which in Internet terminology is called *host-byte order*. The **ntohl** library function does the actual "byte swapping" when this operation is required.

19873: This is a tricky piece of code. For a mask (as passed originally into the inline function) to be a valid, it must consist of a string of one-bits followed by a string of zero-bits. The one's complement reverses the state of the bits. When a constant (that is, the value 1) is added to the latter value, a valid mask is changed from a mass of one-bits to a single one-bit. The AND function checks for the occurrence of the change, to speed up the testing for this condition. If any bits survive the operation, then the mask is invalid. Note that a mask consisting of all 1's is valid (that is, it signifies a host address), but a mask of all 0's is not. The latter result is the one we're looking for.

fib_add_1

20040: Note that this function extends to line 20224.

20057: Memory for the new node is allocated, and basic information is filled in.

20065: Note that the Type of Service field is set to 0, signifying the absence of any special considerations for sending packets. (No way exists to change the value in the routing table, which makes us wonder why the structure contains this member at all.)

20067: Now find (or create) a FIB (line 19966). If the FIB comes back as **NULL**, then release the new node allocated in line 20057.

20073: A pointer to the FIB node is placed in the routing-table block.

20075: The length of the host portion of the mask is calculated by the inline routine **rt_logmask**, based on the mask calculated thus far.

20076: Get the pointer to the FIB zone, based on the length of the mask. If no zone block is present, create one. (Can't create one? Then release the memory allocated for the FIB and get out.)

20089: Zero out the block and then fill it with the information available thus far.

20092: Find the insert point, such that the FIB zone list is in mask-length order.

20095: Turn off interrupts and then insert the FIB zone block into the proper place in the list.

20106: Put the pointer to the FIB zone block in the array and then turn the interrupts back on. It's remotely possible that two processes could cause a FIB block to be allocated more than once (no semaphore), but both blocks would appear in the linked list (even though only one would have the pointer in the array **fib_zones**).

When the number of routine-table cache elements in a particular FIB zone exceeds a given limit (**RTZ_HASHING_LIMIT**, which, in production systems, is 16), a hashing table (consisting of 256 entries) is created for that zone. The idea is that a search for a particular record can be handled in fewer cycles if most of the list can be bypassed.

20115: The purpose of this code block is to convert a single zone list into 256 zone lists. Accordingly, if the number of entries in the FIB zone exceeds the hashing threshold, no hash table exists. If the mask indicates that the routing entry is not for a host, then create a hash table, initialize it, and stroll through the FIB node list to build sublists based on the calculated hash value. To preclude the possibility of race conditions, interrupts are turned off during the update process.

20146: Save the fact that a hashing table now exists.

20150: What happens if no memory is available for the creation of a hashing table? Why, nothing. Because the old FIB zone list is still there, many attempts will be made to create the hash table for that zone. These attempts will slow the proceedings, but won't bring them to a screeching halt.

20151: Find the proper head of the list to scan. If a hash list is present, point to the head of the sublist for the zone. Otherwise, point to the head of the zone itself.

20161: Scan the zone list for the desired destination address. Note that the routine will stop when it reaches a **NULL** pointer, while the rest of the list-scanning routines can handle a **NULL** flag.

20172: Continue scanning the list as long as the destination address still matches.

20174: Lower metric values are more desirable. Therefore, when you find a node with a metric whose value is equal to or higher than the metric value of the candidate node, the search is over.

20182: If the gateway addresses match and a matching gateway address or device has been specified, then save the pointer to the block pointer that meets these criteria.

20192: If the code already has a route with the same metric value, then dump the new block and return from the function.

20203 Insert the new FIB into the list. This operation lays the groundwork for an "insert sort" based on the destination and the metric value.

20210: Increment the zone entry count.

20211: Send to the peer a new route message. The **ip_netlink_msg** routine is at line 17171 in module **ip_output.c** (which is discussed in Chapter 7).

20219: If a route was marked at line 20184, start looking for duplicate routes at that point. Otherwise, start with the route past the newly inserted route.

20224: Search the list until either the destinations don't match or the list is exhausted.

20226: Look for identical gateway addresses and device addresses. When one is found, remove the route from the list, free the node, and decrement the zone entry count. Then, break out of the loop. (At most, only one duplicate route should be in the routing table.)

20242: Because the structure of the routing table has changed, the general hash table is cleared out (via the **rt_cache_flush** routine at line 20688).

fib_create_info

The **fib_create_info** function allocates and fills in a FIB structure element (defined at line 19598). (The comment in the code is that the FIB is "shared by many of the routes.")

19975: If the **RTF_MSS** flag is not set and if the kernel has been compiled not to discover the MTU for the path, then use the default MTU for the device. If the call defines a gateway (**RTF_GATEWAY**), then cap the maximum segment size at 576 bytes.

19994: If the **RTF_WINDOW** flag was not set, clear the window-size value.

19996: If the **RTF_IRTT** flag was not set, clear the initial round-trip time value.

19999: Search the FIB list, starting with the head point **fib_info_list** and continuing until a match is found or the end of the list (indicated by a **NULL**) is reached.

20008: If the gateway address, device, flags, maximum segment size, window size, and initial round-trip time all match, increment the reference count **fib_refcnt**, log the fact, and return the address of the reference FIB.

20015: If the FIB wasn't found, create a FIB by allocating memory from kernel memory. If no memory can be allocated, return **NULL**.

20020: Initialize the structure to all 0's and then fill in all the information that has been gathered so far. Lines 20026 and 20029 through 20031 add this new element to the beginning of the FIB list, so that the search order is last-in/first-found. Log this fact and return the address of the new FIB.

fib_lookup_gateway

This function searches the routing table for a gateway.

19737: For each element in the FIB zone list, find the list in which the destination address is most likely to be found. If the zone-list entry has a hash table, select the correct list by calculating the hash code for the destination and searching the appropriate sublist. Otherwise, use the zone's master list.

19745: Whenever the destination network doesn't match, or if the entry is marked as the default gateway, reject the current FIB entry and try the next one.

19750: A hit! Return the pointer to the Forward Information Block.

get_gw_dev

This inline function performs a route lookup based on the destination address. If the desired address is found, this function returns a pointer to the device name.

19858: Perform the search based on the destination address.

19859: If the desired address is found, return the pointer to the device name. Otherwise, return the **NULL** pointer.

rt_add

This function wraps the handling of the routing-table semaphore around the function **fib_add_1**, so that user processes won't step on each other.

20859: Only one process at a time is allowed to make changes in the routing table. If another process "holds the token" **ip_rt_lock**, then that process must be put to sleep until the other process (or processes) is finished with it. The **sleep_on** function lives in the /usr/src/linux/kernel/sched.c source file (not included with this book).

20861: The **ip_rt_fast_lock** function (defined at line 41304) is a call to the function **atomic_inc**, which lives in the /usr/src/linux/include/asm-i386/atomic.h file (not included with this book). The **atomic_inc** function is written in assembler, so that it works properly in shared-memory multiprocessing systems. This way, nothing—not even other hardware operations—can interfere with the incrementing of the semaphore **ip_rt_lock**. (In older, 16-bit systems, such a function was needed to handle 32-bit numbers in environments with interrupts, because a 32-bit increment was implemented in two instructions, namely, a 16-bit increment followed by a 16-bit increment with carry. Intel 386, 486, Pentium, and clone chips have 32-bit pathways, so that such an incrementation scheme is no longer necessary. Multiprocessing systems keep them around.)

20862: This function incorporates the inline function **fib_add_1** (located at line 20040), which allocates the memory for the routing-table entry and fills the node. Note that this function does not provide an error return.

20864: The next two lines of code clear the semaphore flag and wake up any processes that may be waiting. Then, the function returns with no status.

rt_logmask

19680: This inline function returns the length of the low-order zero-bits in a mask. When the value is 0, the value 32 is returned. Otherwise, the value is changed from network-byte order to host-byte order, and the inline routine **ffz** ("find first zero bit"), located in file /usr/src/linux/include/asm-i386/bitops.h, is invoked. This routine executes a single 80386 bit-search operator, using a pattern such that 0xFFFFFFFF yields 0, 0xFFFFFFFE yields 1, and so on, until 0x80000000 yields 31.

rt_mask

19691: This inline function is the inverse of the **rt_logmask** function. It takes a number that represents the number of zero-bits to be generated at the low-order end of the network address. Any value greater than or equal to 32 yields a result of 0, while a value of 0 results in an all-ones mask. The mask is placed in network-byte order before it is returned.

Removing Routes From The Table

Other ways exist to remove a route from a routing table, but this series of functions is the only one that has the effect of removing a route explicitly. Such a deletion can be triggered by a network administrator who wants to make a change that reflects a modification of the physical network…or who needs to remove an improperly added route that, because of the improper addition, makes a given host unreachable.

ip_rt_kill

This relatively small routine deletes routes from the routing table.

21362: This block of code copies information from the user's passed structure into local storage.

21365: If the caller told us the name of the interface (such as "eth0"), this routine gets the device information block for the device. Otherwise, it leaves the **NULL** pointer that was set in line 21360. If the caller made an error, the appropriate error code is returned.

21381: Call the routine **rt_del** (line 20838) that actually deletes the route, and propagate the return value from the deletion attempt.

21386: This code completes the function.

ip_rt_flush

This function is called when all routes for a device need to be cleaned out. Such a drastic action may be necessary, for example, when a device is taken to the Down state by the system operator, or when a PPP or SLIP connection is closed.

The **ip_rt_flush** routine is a user wrapper for the **fib_flush_1** routine. The code starts at line 20868.

ip_rt_check_expire

This function, which is called by the /usr/src/linux/net/core/arp.c module, checks the routing cache table for outdated entries. This action also has the effect of emptying outdated hardware handles, because the hh blocks are eliminated along with the cache entries.

20593: Run through the 256 lists for the cache, using the hash table as the starting point. Process each element in the list.

20606: If the element has reached its age limit, then remove the element completely and continue searching the list.

20628: The following code is a very complicated way to implement a fuzzy bubble sort. The idea is that if two adjacent elements are present for which the last-used timestamps are within **RT_CACHE_BUBBLE_THRESHOLD** time of each other (defined as five seconds), don't bother reordering the elements. Otherwise, if the physically earlier element is "newer" than the next element on the list, force these two elements to swap places. This swap puts the list in least recently used order, so that the oldest element is at the head of the list. This way, the relatively oldest element is the first to be pruned when the cache gets too big.

ip_rt_hash_code

This inline function calculates a hash value (returning a value between 0 and 255) whose purpose is to distribute IP address references evenly.

41323: Add the top 16 bits of the IP address to the bottom 16 bits of the IP address.

41324: Add bits 7 through 0 (counting the least significant bit as 0) to bits 8 through 15 of the previously calculated sum, and mask off all but the bottom 8 bits of the result. Return this result to the caller.

fib_del_1

This routine searches the FIB subsystem and deletes the references to this route contained in that subsystem.

19924: If the **mask** parameter is 0, then the reference is to a host, not to a network.

19926: This loop sweeps the FIB zone list from top to bottom.

19929: Set the local variable **fp** to the address of the FIB, as pointed to by the FIB zone block. (Note that this operation uses the hash-table entry, if any, that is attached to the FIB block.)

19935: Call the routine **fib_del_list** (line 19879) that extracts the FIB block from the database.

19937: If the deletion was acceptable, decrement the FIB zone block "contains" counter and increment the "number of blocks found" counter.

19941: Otherwise, if the mask is non-0, perform the following operations:

19943: If a route has the same length mask, then pick up the pointer to the list and traverse the list. (The list may be obtained from the zone table itself, or from a hash table.)

19951: Release the appropriate entries from the list. The routine **fib_del_list** returns the number of elements freed.

19957: If any routing-table elements were indeed found and deleted, then flush the cache and return the success indicator (0).

19962: If no elements were removed from the routing table, return the appropriate error code.

fib_del_list

This routine takes the parameters for removing a route and traverses the list passed to it, performing deletions as it goes. It returns the number of elements removed.

19879: The beginning of the function. Because the list is a pointer to a pointer (usually referred to as a *handle*), the pointer needs to be doubly dereferenced before access can be gained to any internal element. Moreover, if necessary, the list head can be altered without causing major problems.

19886: Walk through the list, performing the steps on each element of it.

19895: If the following items do not match—destination, mask, metric (if specified), gateway (if specified), and device (if specified)— then go on to the next element.

19904: Remove the route block from the chain.

19905: If the route being removed is the loopback route, then set the shortcut to the route to the **NULL** pointer.

19908: Tell the neighboring systems that the route is being removed.

19910: Free the node.

19911: Increment the released-node counter.

19913: Return to the caller, indicating the number of routes that have been removed.

fib_flush_1

This routine kills off all routes for a given device. When it's done, it calls the cache flush function.

20279: Walk through the FIB zone list. If a hash table is present for the zone, then walk through all the lists pointed to by the hash table. Otherwise, just walk through the single list. For each list, call the function **rt_flush_list**.

20300: If any routes were removed, call **rt_cache_flush** to clean up the cache.

rt_flush_list

This routine kills off all the routes for a specific device.

20252: Search each node of the list passed by the caller.

20257: If the FIB element is for a different device, and if the block isn't pointing to the loopback device or if the FIB element isn't for the device's IP address, continue the search.

20264: Point to the next element.

20265: If the loopback device entry is being killed off, then reset the pointer.

20268: Free the node and increment the counter for the number of elements removed.

20271: When the list has been exhausted, return the number of elements that were freed.

fib_free_node

This function takes a routing-table element, removes it from the linked list, and frees the kernel memory associated with it. The function also checks the FIB associated with the route, to see whether the routing-table element contains any more references. If not, the FIB is also removed from the list and freed.

19711: If the reference counter in the FIB has been decremented by 1 and is not 0, then don't purge the FIB.

19717: Remove the FIB forward link.

19719: Remove the FIB backward link.

19721: If this block is the first one on the list, update the header pointer.

19723: Free the memory for the FIB.

19725: Free the memory for the router-table entry.

fz_hash_code

This inline function calculates a hash value (returning a value between 0 and 255) based on the network portion of the destination address. This function uses the inline function **ip_rt_hash_code** (line 41321).

19701: Return the 8-bit hash value of the network portion of the address. The parameter **logmask** is the number of zero-bits in the mask value. When **logmask** is shifted right by the number of zero-bits, the network portion of the address is right-aligned before the hash is calculated.

rt_del

This routine removes a specified route (or routes) from the routing table.

20844: This code implements a semaphore for a routing-table lock, to ensure that only a single process at a time manipulates the routing table.

20862: Call the routine **fib_del_1** (line 19916) that actually deletes the route.

20849: Remove the lock.

20850: Release any processes that may be waiting because of this lock.

20851: Propagate the return code to the caller.

rt_free

This routine handles the mechanics of freeing up a routing-table cache entry.

20494: Turn off interrupts. To do so, use the **save_flags** function to save the current CPU state, and then tell the system to disable device interrupts.

20496: Before doing anything else, confirm that the reference count for this entry is 0.

20498: Save the pointer to any hardware handle(s), and clear it from memory. (Strictly speaking, this step isn't necessary, because the memory will be going away anyway.)

20501: Decrement the reference count in the hardware handle. If the count is 0, then get rid of it.

20503: Delete the table entry.

20506: If the reference count isn't 0, add this route to the front of the free-element queue and reset the **RTF_UP** flag.

20509: Set a flag to indicate that the free-element queue contains an element.

Cache Management Routines

The designers of the Linux TCP/IP stack were well aware that network access is "session oriented"—in other words, that once two computers establish a conversation, the conversation lasts for a time and then ends. By recognizing and using the session-oriented nature of network access, the designers were able to implement techniques that led to significant improvements in the performance of network operations—which, on the Internet today, is overwhelmingly synonymous with Web browsing.

Web surfers look at sites one at a time. When surfers land on a site and find the first taste interesting, they tend to stay at that site—often to the exclusion of others—until they're satisfied or take a tangent to another, related site.

Yesterday's dominant application—file transfer—was even more narrowly focused, because there were no links to tempt users away from the FTP server they were using. Unlike Web surfing, which only requires that users specify the correct URL to access a page, FTP requires that users explicitly log onto the server. Consequently, users tend to milk the server they're currently using before moving on to the next one.

In the routing world, this "fixity of access" means that a route that has been used once is very, very likely to be used again in the near future. Conversely, a route that hasn't been used for a given length of time (say, 10 minutes) most likely will not be used again in the near future. These paired assumptions work equally well for routers and host systems, with the only difference lying in the size of the cache in which the recently accessed routes should be saved.

rt_cache_add

This function places in the routing cache a destination used by a routing request, and also performs a housekeeping task, by deleting the least recently used elements after a given number of elements have been placed in the cache.

20942: During development, this code checks to ensure that the value of the semaphore is exactly equal to 1. A semaphore value other than 1 means that the caller hasn't captured the semaphore, and that problems may therefore occur.

20952: If the device associated with the passed router-table entry has a bind routine, and if the gateway address is not the destination address, then search for the correct route. Note that the function **ip_rt_route** expects to be able to capture the semaphore. Therefore, before the call is made, this routine has to let the semaphore go and then snatch it back on return.

20963: If a route was found, and if the route is identical to the route that was passed by the caller, then call the device-bind routine.

20971: If the route is different, then update the hardware handle (if one is present) and copy the pointer from the original route to the new route. Then, call **ip_rt_put** to place the new route information in the cache.

20979: Caches work best when they're small. This code triggers a cleanup when the cache grows too large.

20983: Link the route into the hash table.

20994: Point the hash table to the new route (so that the most recently used route is at the head of the list).

21003: This loop walks through the list, checking for entries that have timed out (manifest constant **RT_CACHE_TIMEOUT**, currently 300 seconds) or that are duplicates of the previously checked route. If an entry is found that meets either of these conditions, then that element is removed and the cache-size counter is decremented. In any case, the search continues until all the elements have been checked.

21024: The CPU flags are restored to their previous state (the flag of interest is the interrupt-enable flag) and the function returns.

rt_cache_flush

This small routine cleans up the cache buffer.

20693: This loop cycles through all **RT_HASH_DIVISOR** possible cache lists.

20698: If the hash table points to nothing, then continue the loop.

20704: Wipe out the pointer to the hash-table entry. (The value of the pointer was saved earlier, in the variable **rth**.)

20707: Cycle through the link list and free up the elements.

20728: Complete the flushing of the routing cache table.

rt_garbage_collect_1

This routine swings through the FIB list and purges any element(s) that shouldn't be there.

20737: Continue through the process until the number of cache elements is below the limit defined by the manifest constant **RT_CACHE_SIZE_MAX** (256 elements).

20739: Run through the hash table and process each list entry.

20746: If the entry hasn't expired (based on the time of last use, adjusted by the number of routes using this entry), then continue searching.

20749: Decrement the cache-content count, adjust the list, and free the element. The **break** statement causes the code to start the search from the start of the hash table.

20758: When no candidates for removal remain, determine whether the garbage-collection operation was thorough enough. If not, decrease the expiration interval and go again. This way, enough elements will be deleted to reduce the cache to a reasonable size.

rt_garbage_collect

The purpose of this function (which is a wrapper for the function **rt_garbage_collect_1**) is to ensure that the routing-table semaphore is captured before the cleanup operation is performed. The function starts at line 20923.

Backend Handlers

Some jobs in route.c just can't be done when the original request comes in. To handle such instances, the Linux programmers set up queues and a flag word **ip_rt_bh_mask** to deal with the issues as they arise.

ip_rt_run_bh

This function is the entry point for the backend handlers. It is called when the lock is released (using **ip_rt_unlock**) and a backend job needs to be done. In one respect, this function is a wrapper for a series of functions that run with interrupts inhibited. Consequently, no way exists that the semaphore can be "stolen." When required (as indicated by bits in **ip_rt_bh_mask**), the following three routines are called: **rt_kick_backlog**, **rt_garbage_collect_1**, and **rt_kick_free_queue**.

When work needs to be deferred, the indicator flags are set by various routines in route.c.

rt_req_enqueue

This routine is used by **ip_rt_redirect** when a redirect request has to be queued. This simple interrupt-safe routine just adds a request to a request list.

rt_req_dequeue

This routine is used by **rt_kick_backlog** when a redirect request has to be removed from a queue. This simple interrupt-safe routine simply removes a request from a request list and returns it.

rt_kick_free_queue

This routine is a cleanup routine borrowed from **rt_free**, previously described. Any routing element that has a non-0 reference count and that was to be freed is handled here.

20525: Turn off (in other words, reset) the "gotta-do-it" flag.

20529: Walk through the free-element list. If the element now has a 0 reference count, remove the hardware-handle cache element and then remove the routing-table entry, too.

rt_kick_backlog

This routine goes through the list of requests for route redirection. Each previously queued request is processed (by calling **rt_redirect_1**, using the information about the request), and the request element is removed from memory.

rt_redirect_1

This routine accepts a call from ICMP. The call causes the routine to record a new route to a host, as determined by the ICMP message-processing code.

20661: If the gateway address is the desired interface, or if the device isn't accessible via the gateway address, forget it.

20665: Build a new routing-table element. Mark it as "dynamic", "modified", "host" (as opposed to "net"), "gateway address valid", and "up".

20684: Add the routing-table element to the queue.

Routing Request Handlers

This section is where the real action takes place. Although routing tables are read-mostly structures, because routes are supposed to change at a relatively slow rate, a busy host or router routinely has to handle thousands of routing requests.

ip_rt_route

Find the route for a particular destination. This routine examines the cache, because, as noted earlier, once a route is used, it tends to be used repeatedly for the life of a session.

21198: Select the correct cache list to examine for the route, and parse the list.

21203: Got a hit? That is, did the destination and any device qualifications match? If so, update the use time, increment the in-use and reference counts, and return to the caller. (Fast, huh?)

21213: Missed. Go look for the route the slow way.

ip_rt_dev

This routine is a wrapper for **fib_lookup**. The result is similar to that of **ip_rt_route**, except that the slow search is always used and no hardware handles are generated. This routine is used only by the ip_alias.c module when aliasing is compiled into the module.

ip_rt_slow_route

Okay, the code didn't find the route quickly, so now the code has to look for it the hard way. This routine searches for the best match in the routing table for a given destination, and an optional device qualification.

21049: Allocate memory for a routing-table entry. If not enough memory was available, then lie—say you didn't find the route.

21056: If this route request is a local lookup, use the faster routine **fib_lookup_local** to find the specific route. Otherwise, use **fib_lookup**.

21061: If you find a matching route, increment its in-use count.

21067: The route may be marked "reject." (This dodge is sometimes used to prevent the propagation of Internet Assigned Numbers Authority (IANA) private network addresses onto the public network.) Alternatively, it may have been not-found. In either case, release the memory that had been allocated for the routing-table entry.

21077: If the Linux kernel was configured for the kerneld.o module, then issue a dynamic route request.

21088: Return without a route-entry pointer.

21091: Set the source address to the interface address. This action associates the destination address with the gateway address that should be used. If the destination address is the gateway address, then decrement the use count for the FIB (except if it's the loopback address) and get the FIB for the gateway.

21103: Here's another chance to fail. If no route can be found, then free the memory and propagate the status of the search.

21110: Fill in all the information about the route.

21123: Mark this route as a host route (as opposed to a network route).

21136: If you have the semaphore, and if you didn't limit your search to a specific interface device, then add the route to the cache for next time (so that you can go fast, of course!). If you don't have the lock, then so much for speed...but better luck next time.

21158: Return the route information.

fib_lookup_local

This routine scans the entire routing table, looking for the "longest" match for the given destination. The idea here is that the route that "works" (in other words, the one with the most high-order one-bits in the netmask) is the route to pick. This approach gets around the problem, which existed in older code, of choosing a nonuseful route (documented at lines 19758 through 19775).

19784: Search each zone list, resetting the success flag at each return to 0 (no match). If a hash table is associated with the zone list, use the hashing function to find the head of the sublist that might contain the desired route. Otherwise, use the main list for the zone.

19794: Search the selected list. If the destination doesn't match the route's destination, go to the next element. Ditto if a device was specified and the devices don't match.

19801: If this route is a gateway route, the code can shout "Eureka!" and return with that route.

19803: Did the code match a nongateway route? Then, remember this fact as the zone- list exam continues.

19805: If the code found a match in the zone list, don't bother looking in the next zone. The code already knows it's in trouble.

19808: If the code didn't find the destination at all, then it returns a failure report.

fib_lookup

This routine looks for any possible route to the destination. The primary difference between this routine and **fib_lookup_local** is that the route in the latter case is supposed to be on the local machine.

19834: Search the zone list. For each zone, if a hash table is present, use it to select the correct sublist for the desired destination. Otherwise, use the main list. Then, scan the selected list.

19844: If the destination doesn't match the request, or if the device (if any) specified by the caller doesn't match the request, then go to the next element on the list.

19849: Was a matching destination found? If so, then report the success by returning the route pointer.

19852: Were no matching destinations found? If so, then report the failure by returning the **NULL** pointer.