Linux IP Networking

A Guide to the Implementation and Modification of the Linux Protocol Stack

Glenn Herrin

TR 00-04

Department of Computer Science
University of New Hampshire

May 31, 2000

Abstract

This document is a guide to understanding how the Linux kernel (version 2.2.14 specifically) implements networking protocols, focused primarily on the Internet Protocol (IP). It is intended as a complete reference for experimenters with overviews, walk-throughs, source code explanations, and examples. The first part contains an in-depth examination of the code, data structures, and functionality involved with networking. There are chapters on initialization, connections and sockets, and receiving, transmitting, and forwarding packets. The second part contains detailed instructions for modifying the kernel source code and installing new modules. There are chapters on kernel installation, modules, the *proc* file system, and a complete example.

Contents

Chapter 1
Introduction

This is version 1.0 of this document, dated May 31, 2000, referencing the Linux kernel version 2.2.14.

1.1 Background

Linux is becoming more and more popular as an alternative operating system. Since it is freely available to everyone as part of the open source movement, literally thousands of programmers are constantly working on the code to implement new features, improve existing ones, and fix bugs and inefficiencies in the code. There are many sources for learning more about Linux, from the source code itself (downloadable from the Internet) to books to ``HOW-TOs'' and message boards maintained on many different subjects.

This document is an effort to bring together many of these sources into one coherent reference on and guide to modifying the networking code within the Linux kernel. It presents the internal workings on four levels: a general overview, more specific examinations of network activities, detailed function walk-throughs, and references to the actual code and data structures. It is designed to provide as much or as little detail as the reader desires. This guide was written specifically about the Linux 2.2.14 kernel (which has already been superseded by 2.2.15) and many of the examples come from the Red Hat 6.1 distribution; hopefully the information provided is general enough that it will still apply across distributions and new kernels. It also focuses almost exclusively on TCP/UDP, IP, and Ethernet - which are the most common but by no means the only networking protocols available for Linux platforms.

As a reference for kernel programmers, this document includes information and pointers on editing and recompiling the kernel, writing and installing modules, and working with the */proc* file system. It also presents an example of a program that drops packets for a selected host, along with analysis of the results. Between the descriptions and the examples, this should answer most questions about how Linux performs networking operations and how you can modify it to suit your own purposes.

This project began in a Computer Science Department networking lab at the University of New Hampshire as an effort to institute changes in the Linux kernel to experiment with different routing algorithms. It quickly became apparent that blindly hacking the kernel was not a good idea, so this document was born as a research record and a reference for future programmers. Finally it became large enough (and hopefully useful enough) that we decided to generalize it, formalize it, and release it for public consumption.

As a final note, Linux is an ever-changing system and truly mastering it, if such a thing is even possible, would take far more time than has been spent putting this reference together. If you notice any misstatements, omissions, glaring errors, or even typos (!) within this document, please contact the person who is currently maintaining it. The goal of this project has been to create a freely available and useful reference for Linux programmers.

1.2 Document Conventions

It is assumed that the reader understands the C programming language and is acquainted with common network protocols. This is not vital for the more general information but the details within this document are intended for experienced programmers and may be incomprehensible to casual Linux users.

Almost all of the code presented requires superuser access to implement. Some of the examples can create security holes where none previously existed; programmers should be careful to restore their systems to a normal state after experimenting with the kernel.

File references and program names are written in a *slanted* font.

Code, command line entries, and machine names are written in a `typewriter` font.

Generic entries or variables (such as an output filename) and comments are written in an *italic* font.

1.3 Sample Network Example

There are numerous examples in this document that help clarify the presented material. For the sake of consistency and familiarity, most of them reference the sample network shown in Figure 1.1.

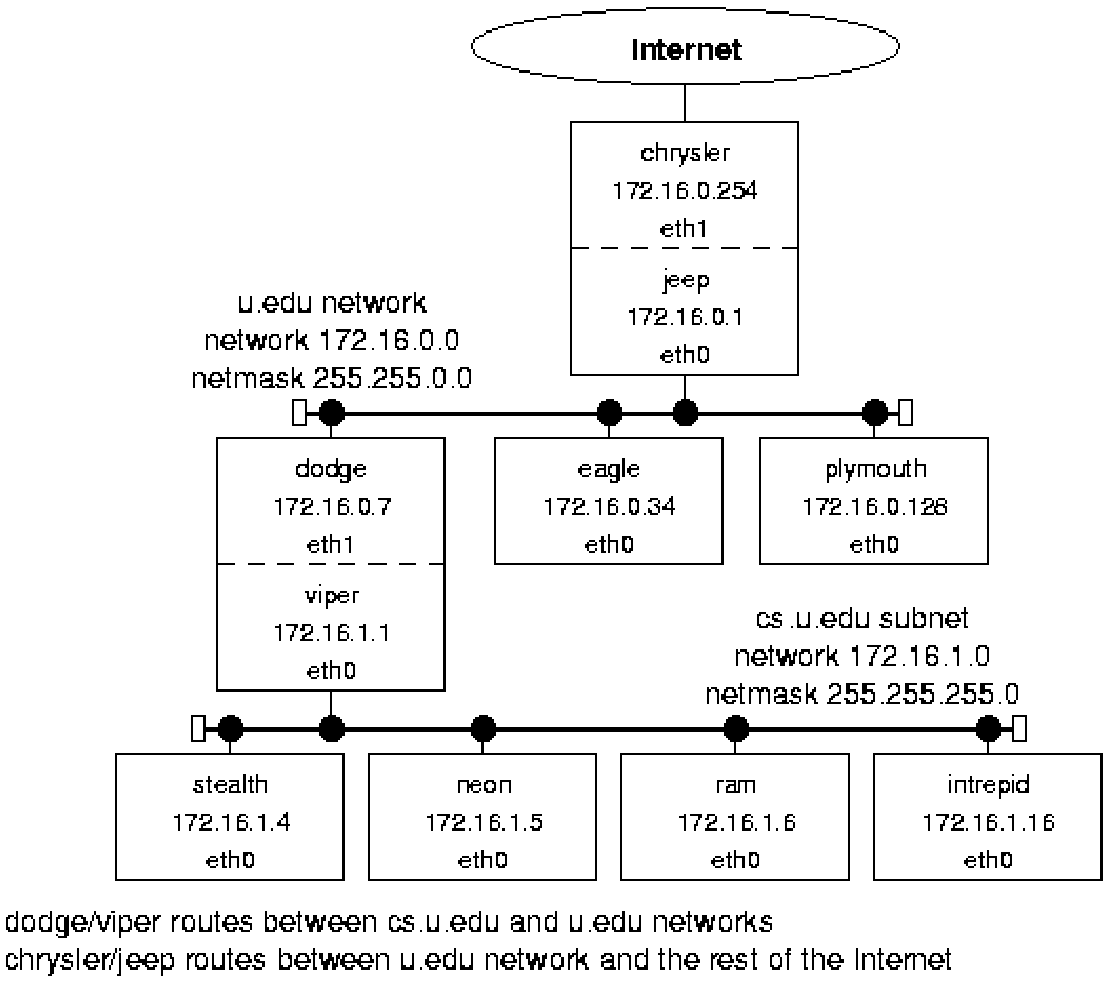


Figure 1.1: Sample network structure.

This network represents the computer system at a fictional unnamed University (U!). It has a router connected to the Internet at large (`chrysler`). That machine is connected (through the `jeep` interface) to the campus-wide network, `u.edu`, consisting of computers named for Chrysler owned car companies (`dodge`, `eagle`, etc.). There is also a LAN subnet for the computer science department, `cs.u.edu`, whose hosts are named after Dodge vehicle models (`stealth`, `neon`, etc.). They are connected to the campus network by the `dodge/viper` computer. Both the `u.edu` and `cs.u.edu` networks use Ethernet hardware and protocols.

This is obviously not a real network. The IP addresses are all taken from the block reserved for class B private networks (that are not guaranteed to be unique). Most real class B networks would have many more computers, and a network with only eight computers would probably not have a subnet. The connection to the Internet (through `chrysler`) would usually be via a T1 or T3 line, and that router would probably be a ``real'' router (i.e. a Cisco Systems hardware router) rather than a computer with two network cards. However, this example is realistic enough to serve its purpose: to illustrate the the Linux network implementation and the interactions between hosts, subnets, and networks.

1.4 Copyright, License, and Disclaimer

1.5 Acknowledgements

I wrote this document as part of my Master's project for the Computer Science Department of the University of New Hampshire. I would like to thank Professor Pilar de la Torre for setting up the project and Professor Radim Bartos for being both a sponsor and my advisor - giving me numerous pointers, much encouragement, and a set of computers on which to experiment. I would also like to credit the United States Army, which has been my home for 11 years and paid for my attendance at UNH.

Glenn Herrin
Major, United States Army
Primary Documenter and Researcher, Version 1.0
gherrin@cs.unh.edu

Chapter 2
Message Traffic Overview

This chapter presents an overview of the entire Linux messaging system. It provides a discussion of configurations, introduces the data structures involved, and describes the basics of IP routing.

2.1 The Network Traffic Path

The Internet Protocol (IP) is the heart of the Linux messaging system. While Linux (more or less) strictly adheres to the layering concept - and it is possible to use a different protocol (like ATM) - IP is almost always the nexus through which packets flow. The IP implementation of the network layer performs routing and forwarding as well as encapsulating data. See Figure 2.1 for a simplified diagram of how network packets move through the Linux kernel.

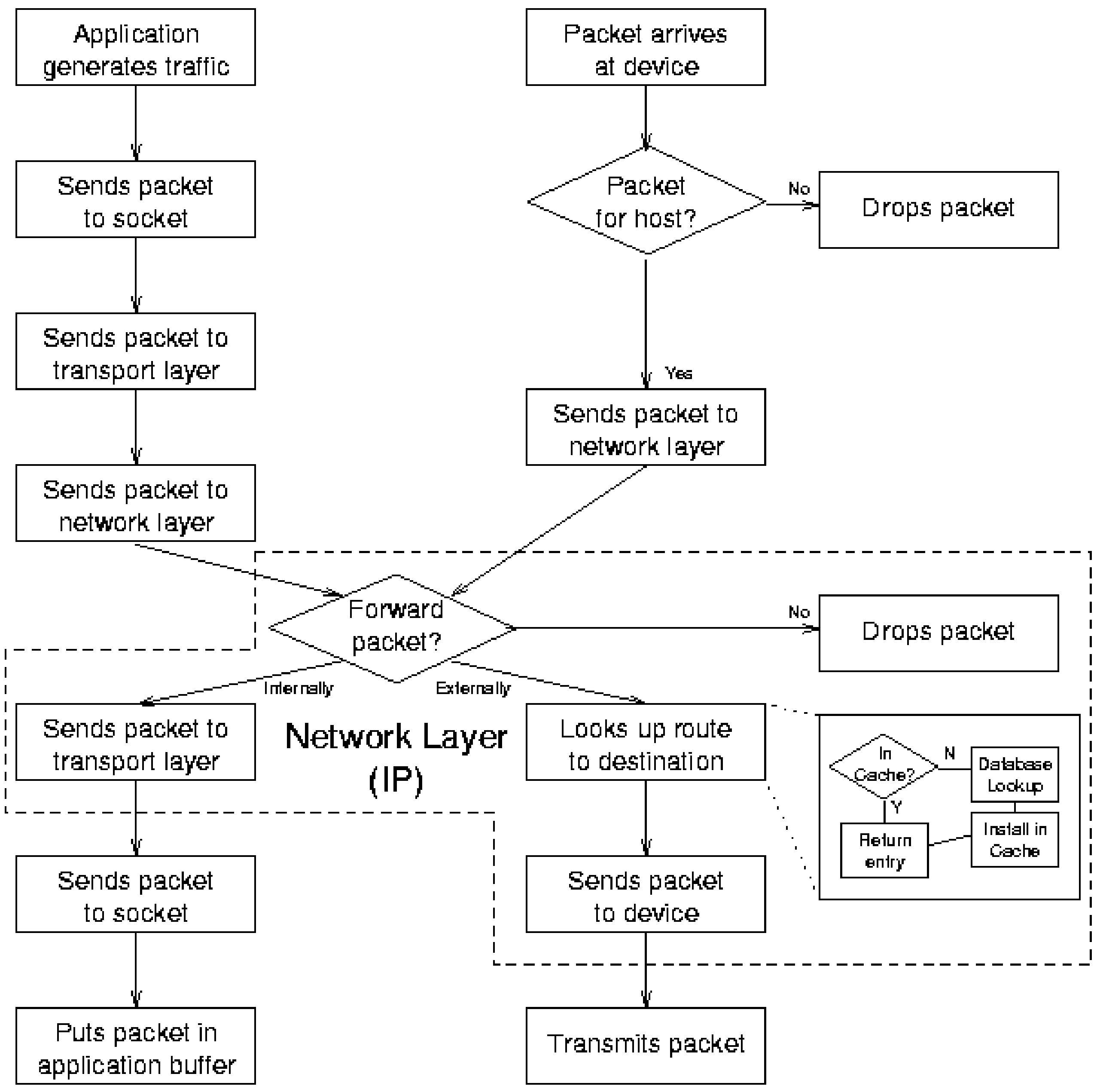


Figure 2.1: Abstraction of the Linux message traffic path.

When an application generates traffic, it sends packets through sockets to a transport layer (TCP or UDP) and then on to the network layer (IP). In the IP layer, the kernel looks up the route to the host in either the routing cache or its Forwarding Information Base (FIB). If the packet is for another computer, the kernel addresses it and then sends it to a link layer output interface (typically an Ethernet device) which ultimately sends the packet out over the physical medium.

When a packet arrives over the medium, the input interface receives it and checks to see if the packet is indeed for the host computer. If so, it sends the packet up to the IP layer, which looks up the route to the packet's destination. If the packet has to be forwarded to another computer, the IP layer sends it back down to an output interface. If the packet is for an application, it sends it up through the transport layer and sockets for the application to read when it is ready.

Along the way, each socket and protocol performs various checks and formatting functions, detailed in later chapters. The entire process is implemented with references and jump tables that isolate each protocol, most of which are set up during initialization when the computer boots. See Chapter 3 for details of the initialization process.

2.2 The Protocol Stack

Network devices form the bottom layer of the protocol stack; they use a link layer protocol (usually Ethernet) to communicate with other devices to send and receive traffic. Input interfaces copy packets from a medium, perform some error checks, and then forward them to the network layer. Output interfaces receive packets from the network layer, perform some error checks, and then send them out over the medium.

IP is the standard network layer protocol. It checks incoming packets to see if they are for the host computer or if they need to be forwarded. It defragments packets if necessary and delivers them to the transport protocols. It maintains a database of routes for outgoing packets; it addresses and fragments them if necessary before sending them down to the link layer.

TCP and UDP are the most common transport layer protocols. UDP simply provides a framework for addressing packets to ports within a computer, while TCP allows more complex connection based operations, including recovery mechanisms for packet loss and traffic management implementations. Either one copies the packet's payload between user and kernel space. However, both are just part of the intermediate layer between the applications and the network.

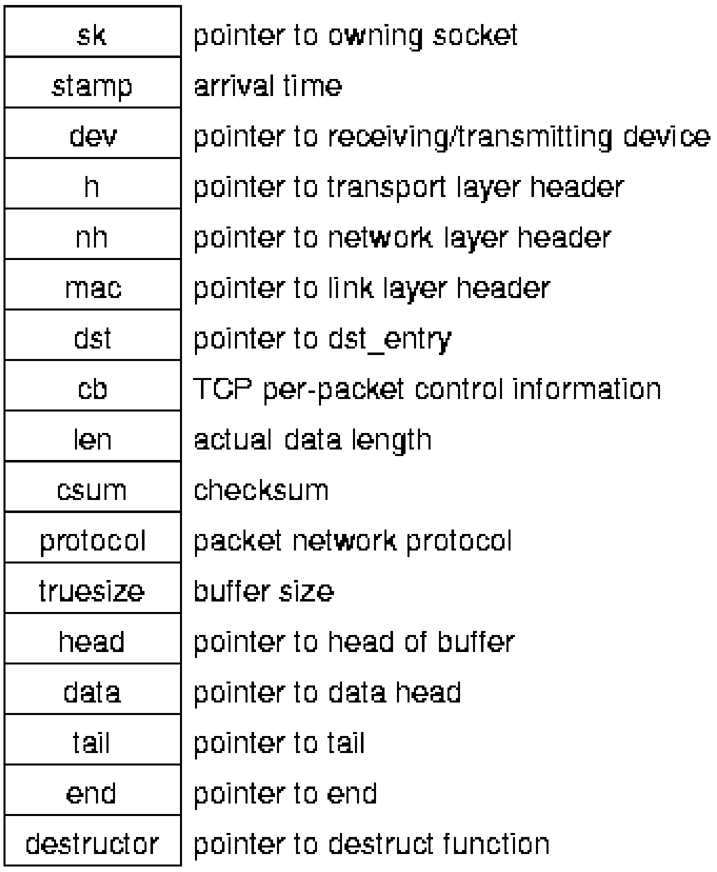
IP Specific INET Sockets are the data elements and implementations of generic sockets. They have associated queues and code that executes socket operations such as reading, writing, and making connections. They act as the intermediary between an application's generic socket and the transport layer protocol.

Generic BSD Sockets are more abstract structures that contain INET sockets. Applications read from and write to BSD sockets; the BSD sockets translate the operations into INET socket operations. See Chapter 4 for more on sockets.

Applications, run in user space, form the top level of the protocol stack; they can be as simple as two-way chat connection or as complex as the Routing Information Protocol (RIP - see Chapter 9).

2.3 Packet Structure

The key to maintaining the strict layering of protocols without wasting time copying parameters and payloads back and forth is the common packet data structure (a socket buffer, or `sk_buff` - Figure 2.2). Throughout all of the various function calls as the data makes it way through the protocols, the payload data is copied only twice; once from user to kernel space and once from kernel space to output medium (for an outbound packet).



sk	pointer to owning socket
stamp	arrival time
dev	pointer to receiving/transmitting device
h	pointer to transport layer header
nh	pointer to network layer header
mac	pointer to link layer header
dst	pointer to dst_entry
cb	TCP per-packet control information
len	actual data length
csum	checksum
protocol	packet network protocol
truesize	buffer size
head	pointer to head of buffer
data	pointer to data head
tail	pointer to tail
end	pointer to end
destructor	pointer to destruct function

Figure 2.2: Packet (`sk_buff`) structure.

This structure contains pointers to all of the information about a packet - its socket, device, route, data locations, etc. Transport protocols create these packet structures from output buffers, while device drivers create them for incoming data. Each layer then fills in the information that it needs as it processes the packet. All of the protocols - transport (TCP/UDP), internet (IP), and link level (Ethernet) - use the same socket buffer.

2.4 Internet Routing

The IP layer handles routing between computers. It keeps two data structures; a Forwarding Information Base (FIB) that keeps track of all of the details for every known route, and a faster routing cache for destinations that are currently in use. (There is also a third structure - the neighbor table - that keeps track of computers that are physically connected to a host.)

The FIB is the primary routing reference; it contains up to 32 zones (one for each bit in an IP address) and entries for every known destination. Each zone contains entries for networks or hosts that can be uniquely identified by a certain number of bits - a network with a netmask of 255.0.0.0 has 8 significant bits and would be in zone 8, while a network with a netmask of 255.255.255.0 has 24 significant bits and would be in zone 24. When IP needs a route, it begins with the most specific zones and searches the entire table until it finds a match (there should always be at least one default entry). The file */proc/net/route* has the contents of the FIB.

The routing cache is a hash table that IP uses to actually route packets. It contains up to 256 chains of current routing entries, with each entry's position determined by a hash function. When a host needs to send a packet, IP looks for an entry in the routing cache. If there is none, it finds the appropriate route in the FIB and inserts a new entry into the cache. (This entry is what the various protocols use to route, not the FIB entry.) The entries remain in the cache as long as they are being used; if there is no traffic for a destination, the entry times out and IP deletes it. The file */proc/net/rt_cache* has the contents of the routing cache.

These tables perform all the routing on a normal system. Even other protocols (such as RIP) use the same structures; they just modify the existing tables within the kernel using the `ioctl()` function. See Chapter 8 for routing details.

Chapter 3 Network Initialization

This chapter presents network initialization on startup. It provides an overview of what happens when the Linux operating system boots, shows how the kernel and supporting programs *ifconfig* and *route* establish network links, shows the differences between several example configurations, and summarizes the implementation code within the kernel and network programs.

3.1 Overview

Linux initializes routing tables on startup only if a computer is configured for networking. (Almost all Linux machines do implement networking, even stand-alone machines, if only to use the loopback device.) When the kernel finishes loading itself, it runs a set of common but system specific utility programs and reads configuration files, several of which establish the computer's networking capabilities. These determine its own address, initialize its interfaces (such as Ethernet cards), and add critical and known static routes (such as one to a router that connects it with the rest of the Internet). If the computer is itself a router, it may also execute a program that allows it to update its routing tables dynamically (but this is NOT run on most hosts).

The entire configuration process can be static or dynamic. If addresses and names never (or infrequently) change, the system administrator must define options and variables in files when setting up the system. In a more mutable environment, a host will use a protocol like the Dynamic Hardware Configuration Protocol (DHCP) to ask for an address, router, and DNS server information with which to configure itself when it boots. (In fact, in either case, the administrator will almost always use a GUI interface - like Red Hat's Control Panel - which automatically writes the configuration files shown below.)

An important point to note is that while most computers running Linux start up the same way, the programs and their locations are not by any means standardized; they may vary widely depending on distribution, security concerns, or whim of the system administrator. This chapter presents as generic a description as possible but assumes a Red Hat Linux 6.1 distribution and a generally static network environment.

3.2 Startup

When Linux boots as an operating system, it loads its image from the disk into memory, unpacks it, and establishes itself by installing the file systems and memory management and other key systems. As the kernel's last (initialization) task, it executes the *init* program. This program reads a configuration file (*/etc/inittab*) which directs it to execute a startup script (found in */etc/rc.d* on Red Hat distributions). This in turn executes more scripts, eventually including the network script (*/etc/rc.d/init.d/network*). (See Section 3.3 for examples of the script and file interactions.)

3.2.1 The Network Initialization Script

The network initialization script sets environment variables to identify the host computer and establish whether or not the computer will use a network. Depending on the values given, the network script turns on (or off) IP forwarding and IP fragmentation. It also establishes the default router for all network traffic and the device to use to send such traffic. Finally, it brings up any network devices using the *ifconfig* and *route* programs. (In a dynamic environment, it would query the DHCP server for its network information instead of reading its own files.)

The script(s) involved in establishing networking can be very straightforward; it is entirely possible to have one big script that simply executes a series of commands that will set up a single machine properly. However, most Linux distributions come with a large number of generic scripts that work for a wide variety of machine setups. This leaves a lot of indirection and conditional execution in the scripts, but actually makes setting up any one machine much easier. For example, on Red Hat distributions, the */etc/rc.d/init.d/network* script runs several other scripts and sets up variables like `interfaces_boot` to keep track of which */etc/sysconfig/network-scripts/ifup* scripts to run. Tracing the process manually is very complicated, but simple modifications of only two configuration files (putting the proper names and IP addresses in the */etc/sysconfig/network* and */etc/sysconfig/network-scripts/ifcfg-eth0* files) sets up the entire system properly (and a GUI makes the process even simpler).

When the network script finishes, the FIB contains the specified routes to given hosts or networks and the routing cache and neighbor tables are empty. When traffic begins to flow, the kernel will update the neighbor table and routing cache as part of the normal network operations. (Network traffic may begin during initialization if a host is dynamically configured or consults a network clock, for example.)

3.2.2 *ifconfig*

The *ifconfig* program configures interface devices for use. (This program, while very widely used, is not part of the kernel.) It provides each device with its (IP) address, netmask, and broadcast address. The device in turn will run its own initialization functions (to set any static variables) and register its interrupts and service routines with the kernel. The *ifconfig* commands in the network script look like this:

```
ifconfig ${DEVICE} ${IPADDR} netmask ${NMASK} broadcast ${BCAST}
```

(where the variables are either written directly in the script or are defined in other scripts).

The *ifconfig* program can also provide information about currently configured network devices (calling with no arguments displays all the active interfaces; calling with the `-a` option displays all interfaces, active or not):

```
ifconfig
```

This provides all the information available about each working interface; addresses, status, packet statistics, and operating system specifics. Usually there will be at least two interfaces - a network card and the loopback device. The information for each interface looks like this (this is the `viper` interface):

```
eth0  Link encap:Ethernet  HWaddr 00:C1:4E:7D:9E:25
      inet addr:172.16.1.1  Bcast:172.16.1.255  Mask:255.255.255.0
      UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
      RX packets:389016 errors:16534 dropped:0 overruns:0 frame:24522
```

```
          TX packets:400845 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          Interrupt:11 Base address:0xcc00
```

A superuser can use *ifconfig* to change interface settings from the command line; here is the syntax:

> `ifconfig` *interface [aftype] options | address ...*

... and some of the more useful calls:

> `ifconfig eth0 down` - shut down `eth0`
> `ifconfig eth1 up` - activate `eth1`
> `ifconfig eth0 arp` - enable ARP on `eth0`
> `ifconfig eth0 -arp` - disable ARP on `eth0`
> `ifconfig eth0 netmask 255.255.255.0` - set the `eth0` netmask
> `ifconfig lo mtu 2000` - set the loopback maximum transfer unit
> `ifconfig eth1 172.16.0.7` - set the `eth1` IP address

Note that modifying an interface configuration can indirectly change the routing tables. For example, changing the netmask may make some routes moot (including the default or even the route to the host itself) and the kernel will delete them.

3.2.3 *route*

The *route* program simply adds predefined routes for interface devices to the Forwarding Information Base (FIB). This is not part of the kernel, either; it is a user program whose command in the script looks like this:

> `route add -net ${NETWORK} netmask ${NMASK} dev ${DEVICE}` -or-
> `route add -host ${IPADDR} ${DEVICE}`

(where the variables are again spelled out or defined in other scripts).

The *route* program can also delete routes (if run with the `del` option) or provide information about the routes that are currently defined (if run with no options):

```
route
```

This displays the Kernel IP routing table (the FIB, not the routing cache). For example (the `stealth` computer):

```
Kernel IP routing table
Destination    Gateway        Genmask         Flags Metric Ref Use Iface
172.16.1.4     *              255.255.255.255 UH    0      0     0 eth0
172.16.1.0     *              255.255.255.0   U     0      0     0 eth0
127.0.0.0      *              255.0.0.0       U     0      0     0 lo
default        viper.u.edu    0.0.0.0         UG    0      0     0 eth0
```

A superuser can use *route* to add and delete IP routes from the command line; here is the basic syntax:

> `route add` *[-net|-host] target [option arg]*
> `route del` *[-net|-host] target [option arg]*

... and some useful examples:

`route add -host 127.16.1.0 eth1` - adds a route to a host
`route add -net 172.16.1.0 netmask 255.255.255.0 eth0` - adds a network
`route add default gw jeep` - sets the default route through `jeep`
(Note that a route to `jeep` must already be set up)
`route del -host 172.16.1.16` - deletes entry for host `172.16.1.16`

3.2.4 Dynamic Routing Programs

If the computer is a router, the network script will run a routing program like *routed* or *gated*. Since most computers are always on the same hard-wired network with the same set of addresses and limited routing options, most computers do not run one of these programs. (If an Ethernet cable is cut, traffic simply will not flow; there is no need to try to reroute or adjust routing tables.) See Chapter 9 for more information about *routed*.

3.3 Examples

The following are examples of files for systems set up in three different ways and explanations of how they work. Typically every computer will execute a network script that reads configuration files, even if the files tell the computer not to implement any networking.

3.3.1 Home Computer

These files would be on a computer that is not permanently connected to a network, but has a modem for `ppp` access. (This section does not reference a computer from the general example.)

This is the first file the network script will read; it sets several environment variables. The first two variables set the computer to run networking programs (even though it is not on a network) but not to forward packets (since it has nowhere to send them). The last two variables are generic entries.

/etc/sysconfig/network

```
NETWORKING=yes
FORWARD_IPV4=false
HOSTNAME=localhost.localdomain
GATEWAY=
```

After setting these variables, the network script will decide that it needs to configure at least one network device in order to be part of a network. The next file (which is almost exactly the same on all Linux computers) sets up environment variables for the loopback device. It names it and gives it its (standard) IP address, network mask, and broadcast address as well as any other device specific variables. (The ONBOOT variable is a flag for the script program that tells it to configure this device when it boots.) Most computers, even those that will never connect to the Internet, install the loopback device for inter-process communication.

/etc/sysconfig/network-scripts/ifcfg-lo

```
DEVICE=lo
IPADDR=127.0.0.1
NMASK=255.0.0.0
NETWORK=127.0.0.0
BCAST=127.255.255.255
ONBOOT=yes
NAME=loopback
```

```
    BOOTPROTO=none
```

After setting these variables, the script will run the *ifconfig* program and stop, since there is nothing else to do at the moment. However, when the `ppp` program connects to an Internet Service Provider, it will establish a `ppp` device and addressing and routes based on the dynamic values assigned by the ISP. The DNS server and other connection information should be in an *ifcfg-ppp* file.

3.3.2 Host Computer on a LAN

These files would be on a computer that is connected to a LAN; it has one Ethernet card that should come up whenever the computer boots. These files reflect entries on the `stealth` computer from the general example.

This is the first file the network script will read; again the first variables simply determine that the computer will do networking but that it will not forward packets. The last four variables identify the computer and its link to the rest of the Internet (everything that is not on the LAN).

/etc/sysconfig/network

```
    NETWORKING=yes
    FORWARD_IPV4=false
    HOSTNAME=stealth.cs.u.edu
    DOMAINNAME=cs.u.edu
    GATEWAY=172.16.1.1
    GATEWAYDEV=eth0
```

After setting these variables, the network script will configure the network devices. This file sets up environment variables for the Ethernet card. It names the device and gives it its IP address, network mask, and broadcast address as well as any other device specific variables. This kind of computer would also have a loopback configuration file exactly like the one for a non-networked computer.

/etc/sysconfig/network-scripts/ifcfg-eth0

```
    DEVICE=eth0
    IPADDR=172.16.1.4
    NMASK=255.255.255.0
    NETWORK=172.16.1.0
    BCAST=172.16.1.255
    ONBOOT=yes
    BOOTPROTO=none
```

After setting these variables, the network script will run the *ifconfig program* to start the device. Finally, the script will run the *route* program to add the default route (`GATEWAY`) and any other specified routes (found in the */etc/sysconfig/static-routes file*, if any). In this case only the default route is specified, since all traffic either stays on the LAN (where the computer will use ARP to find other hosts) or goes through the router to get to the outside world.

3.3.3 Network Routing Computer

These files would be on a computer that serves as a router between two networks; it has two Ethernet cards, one for each network. One card is on a large network (WAN) connected to the Internet (through yet another router) while the other is on a subnetwork (LAN). Computers on the LAN that need to communicate with the rest of the Internet send traffic through this computer (and vice versa). These files reflect entries on the `dodge/viper` computer from the general example.

This is the first file the network script will read; it sets several environment variables. The first two simply determine that the computer will do networking (since it is on a network) and that this one will forward packets (from one network to the other). IP Forwarding is built into most kernels, but it is not active unless there is a 1 ``written'' to the */proc/net/ipv4/ip_forward* file. (One of the network scripts performs an `echo 1 > /proc/net/ipv4/ip_forward` if `FORWARD_IPV4` is true.) The last four variables identify the computer and its link to the rest of the Internet (everything that is not on one of its own networks).

/etc/sysconfig/network

```
NETWORKING=yes
FORWARD_IPV4=true
HOSTNAME=dodge.u.edu
DOMAINNAME=u.edu
GATEWAY=172.16.0.1
GATEWAYDEV=eth1
```

After setting these variables, the network script will configure the network devices. These files set up environment variables for two Ethernet cards. They name the devices and give them their IP addresses, network masks, and broadcast addresses. (Note that the BOOTPROTO variable remains defined for the second card.) Again, this computer would have the standard loopback configuration file.

/etc/sysconfig/network-scripts/ifcfg-eth0

```
DEVICE=eth0
IPADDR=172.16.1.1
NMASK=255.255.255.0
NETWORK=172.16.1.0
BCAST=172.16.1.255
ONBOOT=yes
BOOTPROTO=static
```

/etc/sysconfig/network-scripts/ifcfg-eth1

```
DEVICE=eth1
IPADDR=172.16.0.7
NMASK=255.255.0.0
NETWORK=172.16.0.0
BCAST=172.16.255.255
ONBOOT=yes
```

After setting these variables, the network script will run the *ifconfig* program to start each device. Finally, the script will run the *route* program to add the default route (`GATEWAY`) and any other specified routes (found in the */etc/sysconfig/static-routes file*, if any). In this case again, the default route is the only specified route, since all traffic will go on the network indicated by the network masks or through the default router to reach the rest of the Internet.

3.4 Linux and Network Program Functions

The following are alphabetic lists of the Linux kernel and network program functions that are most important to initialization, where they are in the source code, and what they do. The *SOURCES* directory shown represents the directory that contains the source code for the given network file. The executable files should come with any Linux distrbution, but the source code probably does not.

These sources are available as a package separate from the kernel source (Red Hat Linux uses the *rpm* package manager). The code below is from the *net-tools-1.53-1* source code package, 29 August 1999. The packages are available from the *www.redhat.com/apps/download* web page. Once downloaded, *root* can install the package with the following commands (starting from the directory with the package):

```
rpm -i net-tools-1.53-1.src.rpm
cd /usr/src/redhat/SOURCES
tar xzf net-tools-1.53.tar.gz
```

This creates a */usr/src/redhat/SOURCES/net-tools-1.53* directory and fills it with the source code for the *ifconfig* and *route* programs (among others). This process should be similar (but is undoubtably not exactly the same) for other Linux distributions.

3.4.1 *ifconfig*

```
devinet_ioctl() - net/ipv4/devinet.c (398)
  creates an info request (ifreq) structure and copies data from
      user to kernel space
  if it is an INET level request or action, executes it
  if it is a device request or action, calls a device function
  copies ifreq back into user memory
  returns 0 for success

>>> ifconfig main() - SOURCES/ifconfig.c (478)
  opens a socket (only for use with ioctl function)
  searches command line arguments for options
  calls if_print() if there were no arguments or the only argument
      is an interface name
  loops through remaining arguments, setting or clearing flags or
      calling ioctl() to set variables for the interface

if_fetch() - SOURCES/lib/interface.c (338)
  fills in an interface structure with multiple calls to ioctl() for
      flags, hardware address, metric, MTU, map, and address information

if_print() - SOURCES/ifconfig.c (121)
  calls ife_print() for given (or all) interface(s)
      (calls if_readlist() to fill structure list if necessary and
      then displays information about each interface)

if_readlist() - SOURCES/lib/interface.c (261)
  opens /proc/net/dev and parses data into interface structures
  calls add_interface() for each device to put structures into a list

inet_ioctl() - net/ipv4/af_inet.c (855)
  executes a switch based on the command passed
      [for ifconfig, calls devinet_ioctl()]

ioctl() -
  jumps to appropriate handler routine [= inet_ioctl()]
```

3.4.2 *route*

```
INET_rinput() - SOURCES/lib/inet_sr.c (305)
  checks for errors (cannot flush table or modify routing cache)
  calls INET_setroute()

INET_rprint() - SOURCES/lib/inet_gr.c (442)
  if the FIB flag is set, calls rprint_fib()
```

```
      (reads, parses, and displays contents of /proc/net/route)
    if the CACHE flag is set, calls rprint_cache()
      (reads, parses, and displays contents of /proc/net/rt_cache)

INET_setroute() - SOURCE/lib/inet_sr.c (57)
  establishes whether route is to a network or a host
  checks to see if address is legal
  loops through arguments, filling in rtentry structure
  checks for netmask conflicts
  creates a temporary socket
  calls ioctl() with rtentry to add or delete route
  closes socket and returns 0

ioctl() -
  jumps to appropriate handler routine [= ip_rt_ioctl()]

ip_rt_ioctl() - net/ipv4/fib_frontend.c (246)
  converts passed parameters to routing table entry (struct rtentry)
  if deleting a route:
    calls fib_get_table() to find the appropriate table
    calls the table->tb_delete() function to remove it
  if adding a route
    calls fib_net_table() to find an entry point
    calls the table->tb_insert() function to add the entry
  returns 0 for success

>>> route main() - SOURCES/route.c (106)
  calls initialization routines that set print and edit functions
  gets and parses the command line options (acts on some options
      directly by setting flags or displaying information)
  checks the options (prints a usage message if there is an error)
  if there are no options, calls route_info()
  if the option is to add, delete, or flush routes,
      calls route_edit() with the passed parameters
  if the option is invalid, prints a usage message
  returns result of

route_edit() - SOURCES/lib/setroute.c (69)
  calls get_aftype() to translate address family from text to a pointer
  checks for errors (unsupported or nonexistent family)
  calls the address family rinput() function [= INET_rinput()]

route_info() - SOURCES/lib/getroute.c (72)
  calls get_aftype() to translate address family from text to a pointer
  checks for errors (unsupported or nonexistent family)
  calls the address family rprint() function [= INET_rprint()]
```

Chapter 4 Connections

This chapter presents the connection process. It provides an overview of the connection process, a description of the socket data structures, an introduction to the routing system, and summarizes the implementation code within the kernel.

4.1 Overview

The simplest form of networking is a connection between two hosts. On each end, an application gets a socket, makes the transport layer connection, and then sends or receives packets. In Linux, a socket is actually composed of two socket structures (one that contains the other). When an application creates a socket, it is initialized but empty. When the socket makes a connection (whether or not this involves traffic with the other end) the IP layer determines the route to the distant host and stores that information in the socket. From that point on, all traffic using that connection uses that route - sent packets will travel through the correct device and the proper routers to the distant host, and received packets will appear in the socket's queue.

4.2 Socket Structures

There are two main socket structures in Linux: general BSD sockets and IP specific INET sockets. They are strongly interrelated; a BSD socket has an INET socket as a data member and an INET socket has a BSD socket as its owner.

BSD sockets are of type `struct socket` as defined in *include/linux/socket.h*. BSD socket variables are usually named `sock` or some variation thereof. This structure has only a few entries, the most important of which are described below.

- `struct proto_ops *ops` - this structure contains pointers to protocol specific functions for implementing general socket behavior. For example, `ops- > sendmsg` points to the `inet_sendmsg()` function.
- *struct inode *inode* - this structure points to the file inode that is associated with this socket.
- `struct sock *sk` - this is the INET socket that is associated with this socket.

INET sockets are of type `struct sock` as defined in *include/net/sock.h*. INET socket variables are usually named `sk` or some variation thereof. This structure has many entries related to a wide variety of uses; there are many hacks and configuration dependent fields. The most important data members are described below:

- `struct sock *next, *pprev` - all sockets are linked by various protocols, so these pointers allow the protocols to traverse them.
- `struct dst_entry *dst_cache` - this is a pointer to the route to the socket's other side (the destination for sent packets).
- `struct sk_buff_head receive_queue` - this is the head of the receive queue.
- `struct sk_buff_head write_queue` - this is the head of the send queue.
- `__u32 saddr` - the (Internet) source address for this socket.
- `struct sk_buff_head back_log,error_queue` - extra queues for a backlog of packets (not to be confused with the main backlog queue) and erroneous packets for this socket.
- `struct proto *prot` - this structure contains pointers to transport layer protocol specific functions. For example, `prot- > recvmsg` may point to the `tcp_v4_recvmsg()` function.
- `union struct tcp_op af_tcp; tp_pinfo` - TCP options for this socket.
- `struct socket *sock` - the parent BSD socket.
- Note that there are many more fields within this structure; these are only the most critical and non-obvious. The rest are either not very important or have self-explanatory names (e.g., `ip_ttl` is the IP Time-To-Live counter).

4.3 Sockets and Routing

Sockets only go through the routing lookup process once for each destination (at connection time). Because Linux sockets are so closely related to IP, they contain routes to the other end of a connection (in the `sock- >`

`sk- > dst_cache` variable). The transport protocols call the `ip_route_connect()` function to determine the route from host to host during the connection process; after that, the route is presumed not to change (though the path pointed to by the `dst_cache` may indeed change). The socket does not need to do continuous routing table look-ups for each packet it sends or receives; it only tries again if something unexpected happens (such as a neighboring computer going down). This is the benefit of using connections.

4.4 Connection Processes

4.4.1 Establishing Connections

Application programs establish sockets with a series of system calls that look up the distant address, establish a socket, and then connect to the machine on the other end.

```
/* look up host */
server = gethostbyname(SERVER_NAME);
/* get socket */
sockfd = socket(AF_INET, SOCK_STREAM, 0);
/* set up address */
address.sin_family = AF_INET;
address.sin_port = htons(PORT_NUM);
memcpy(&address.sin_addr,server->h_addr,server->h_length);
/* connect to server */
connect(sockfd, &address, sizeof(address));
```

The `gethostbyname()` function simply looks up a host (such as ``viper.cs.u.edu") and returns a structure that contains an Internet (IP) address. This has very little to do with routing (only inasmuch as the host may have to query the network to look up an address) and is simply a translation from a human readable form (text) to a computer compatible one (an unsigned 4 byte integer).

The `socket()` call is more interesting. It creates a socket object, with the appropriate data type (a `sock` for INET sockets) and initializes it. The socket contains inode information and protocol specific pointers for various network functions. It also establishes defaults for queues (incoming, outgoing, error, and backlog), a dummy header info for TCP sockets, and various state information.

Finally, the `connect()` call goes to the protocol dependent connection routine (e.g., `tcp_v4_connect()` or `udp_connect()`). UDP simply establishes a route to the destination (since there is no virtual connection). TCP establishes the route and then begins the TCP connection process, sending a packet with appropriate connection and window flags set.

4.4.2 Socket Call Walk-Through

- Check for errors in call
- Create (allocate memory for) socket object
- Put socket into INODE list
- Establish pointers to protocol functions (INET)
- Store values for socket type and protocol family
- Set socket state to closed
- Initialize packet queues

4.4.3 Connect Call Walk-Through

- Check for errors

- Determine route to destination:
    - Check routing table for existing entry (return that if one exists)
    - Look up destination in FIB
    - Build new routing table entry
    - Put entry in routing table and return it
- Store pointer to routing entry in socket
- Call protocol specific connection function (e.g., send a TCP connection packet)
- Set socket state to established

4.4.4 Closing Connections

Closing a socket is fairly straightforward. An application calls `close()` on a socket, which becomes a `sock_close()` function call. This changes the socket state to disconnecting and calls the data member's (INET socket's) release function. The INET socket in turn cleans up its queues and calls the transport protocol's close function, `tcp_v4_close()` or `udp_close()`. These perform any necessary actions (the TCP functions may send out packets to end the TCP connection) and then clean up any data structures they have remaining. Note that no changes are made for routing; the (now-empty) socket no longer has a reference to the destination and the entry in the routing cache will remain until it is freed for lack of use.

4.4.5 Close Walk-Through

- Check for errors (does the socket exist?)
- Change the socket state to disconnecting to prevent further use
- Do any protocol closing actions (e.g., send a TCP packet with the FIN bit set)
- Free memory for socket data structures (TCP/UDP and INET)
- Remove socket from INODE list

4.5 Linux Functions

The following is an alphabetic list of the Linux kernel functions that are most important to connections, where they are in the source code, and what they do. To follow function calls for creating a socket, begin with `sock_create()`. To follow function calls for closing a socket, begin with `sock_close()`.

```
destroy_sock - net/ipv4/af_inet.c (195)
  deletes any timers
  calls any protocols specific destroy functions
  frees the socket's queues
  frees the socket structure itself

fib_lookup() - include/net/ip_fib.h (153)
  calls tb_lookup() [= fn_hash_lookup()] on local and main tables
  returns route or unreachable error

fn_hash_lookup() - net/ipv4/fib_hash.c (261)
  looks up and returns route to an address

inet_create() - net/ipv4/af_inet.c (326)
  calls sk_alloc() to get memory for sock
  initializes sock structure:
    sets proto structure to appropriate values for TCP or UDP
    calls sock_init_data()
    sets family,protocol,etc. variables
    calls the protocol init function (if any)

  inet_release() - net/ipv4/af_inet.c (463)
    changes socket state to disconnecting
    calls ip_mc_drop_socket to leave multicast group (if necessary)
    sets owning socket's data member to NULL
    calls sk->prot->close() [=TCP/UDP_close()]

  ip_route_connect() - include/net/route.h (140)
    calls ip_route_output() to get a destination address
    returns if the call works or generates an error
    otherwise clears the route pointer and try again

  ip_route_output() - net/ipv4/route.c (1664)
    calculates hash value for address
    runs through table (starting at hash) to match addresses and TOS
    if there is a match, updates stats and return route entry
    else calls ip_route_output_slow()

  ip_route_output_slow() - net/ipv4/route.c (1421)
    if source address is known, looks up output device
    if destination address is unknown, sets up loopback
    calls fib_lookup() to find route in FIB
    allocates memory new routing table entry
    initializes table entry with source, destination, TOS, output device,
        flags
    calls rt_set_nexthop() to find next destination
    returns rt_intern_hash(), which installs route in routing table

  rt_intern_hash() - net/ipv4/route.c (526)
    loops through rt_hash_table (starting at hash value)
    if keys match, put rtable entry in front bucket
    else put rtable entry into hash table at hash

  >>> sock_close() - net/socket.c (476)
    checks if socket exists (could be null)
    calls sock_fasync() to remove socket from async list
    calls sock_release()

  >>> sock_create() - net/socket.c (571)
    checks parameters
    calls sock_alloc() to get an available inode for the socket and
        initialize it
    sets socket->type (to SOCK_STREAM, SOCK_DGRAM...)
    calls net_family->create() [= inet_create()] to build sock structure
    returns established socket

  sock_init_data() - net/core/sock.c (1018)
    initializes all generic sock values

  sock_release() - net/socket.c (309)
    changes state to disconnecting
    calls sock->ops->release() [= inet_release()]
    calls iput() to remove socket from inode list

  sys_socket() - net/socket.c (639)
    calls sock_create() to get and initialize socket
    calls get_fd() to assign an fd to the socket
    sets socket->file to fcheck() (pointer to file)
    calls sock_release() if anything fails

  tcp_close() - net/ipv4/tcp.c (1502)
  check for errors
  pops and discards all packets off incoming queue
  sends messages to destination to close connection (if required)

tcp_connect() - net/ipv4/tcp_output.c (910)
  completes connection packet with appropriate bits and window sizes set
  puts packet on socket output queue
  calls tcp_transmit_skb() to send packet, initiating TCP connection

tcp_v4_connect() - net/ipv4/tcp_ipv4.c (571)
  checks for errors
  calls ip_route_connect() to find route to destination
  creates connection packet
  calls tcp_connect() to send packet

udp_close() - net/ipv4/udp.c (954)
  calls udp_v4_unhash() to remove socket from socket list
  calls destroy_sock()

udp_connect() - net/ipv4/udp.c (900)
  calls ip_route_connect() to find route to destination
  updates socket with source and destination addresses and ports
  changes socket state to established
  saves the destination route in sock->dst_cache
```

Chapter 5
Sending Messages

This chapter presents the sending side of message trafficking. It provides an overview of the process, examines the layers packets travel through, details the actions of each layer, and summarizes the implementation code within the kernel.

5.1 Overview

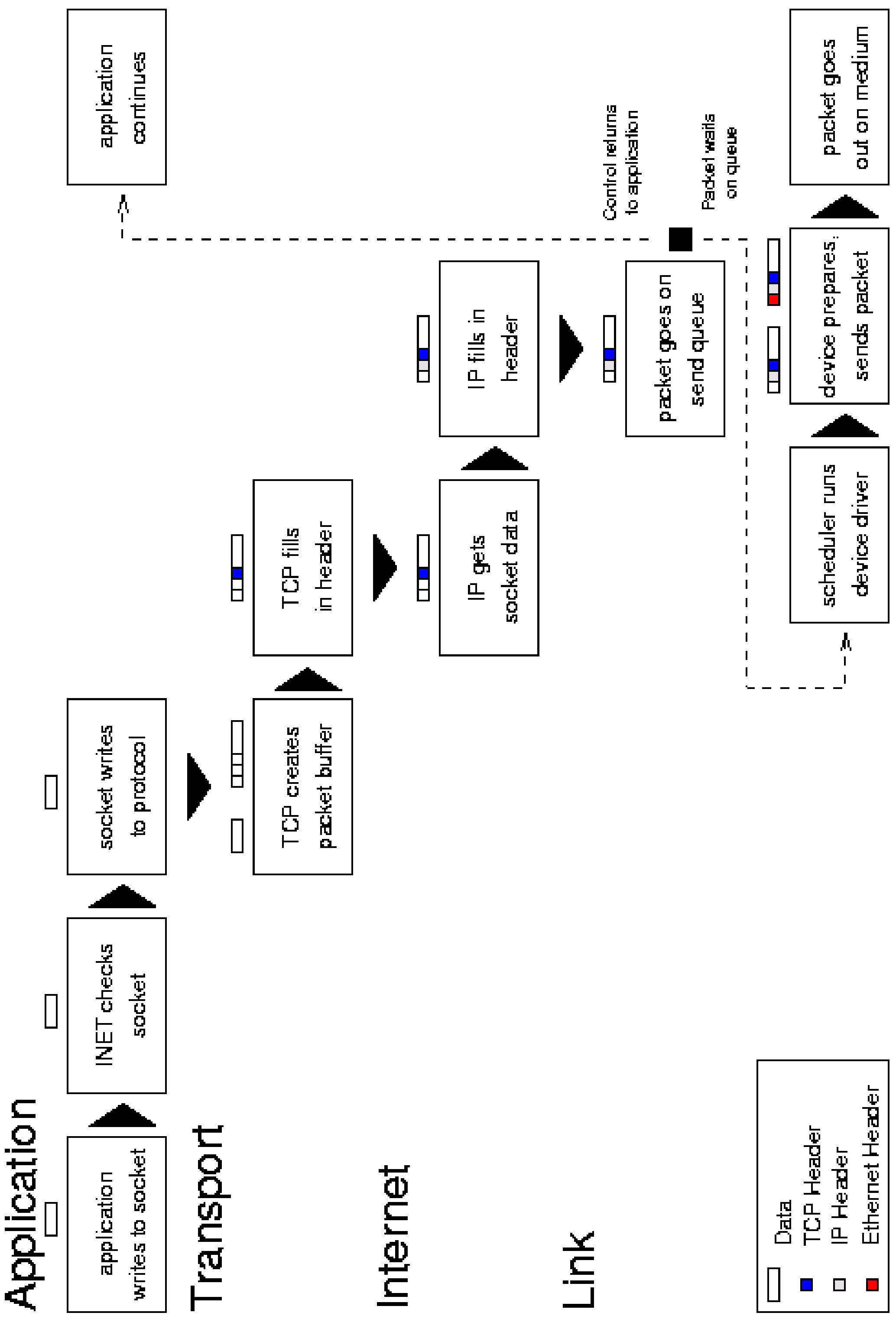

Application
Transport
Internet
Link
application writes to socket
INET checks socket
socket writes to protocol
TCP creates packet buffer
TCP fills in header
IP gets socket data
IP fills in header
packet goes on send queue
Control returns to application
Packet waits on queue
scheduler runs device driver
device prepares, sends packet
packet goes out on medium
application continues
Data
TCP Header
IP Header
Ethernet Header

Figure 5.1: Message transmission.

An outgoing message begins with an application system call to write data to a socket. The socket examines its own connection type and calls the appropriate send routine (typically INET). The send function verifies the status of the socket, examines its protocol type, and sends the data on to the transport layer routine (such as TCP or UDP). This protocol creates a new buffer for the outgoing packet (a socket buffer, or `struct sk_buff skb`), copies the data from the application buffer, and fills in its header information (such as port number, options, and checksum) before passing the new buffer to the network layer (usually IP). The IP send functions fill in more of the buffer with its own protocol headers (such as the IP address, options, and checksum). It may also fragment the packet if required. Next the IP layer passes the packet to the link layer function, which moves the packet onto the sending device's `xmit` queue and makes sure the device knows that it has traffic to send. Finally, the device (such as a network card) tells the bus to send the packet.

5.2 Sending Walk-Through

5.2.1 Writing to a Socket

- Write data to a socket (application)
- Fill in message header with location of data (socket)
- Check for basic errors - is socket bound to a port? can the socket send messages? is there something wrong with the socket?
- Pass the message header to appropriate transport protocol (INET socket)

5.2.2 Creating a Packet with UDP

- Check for errors - is the data too big? is it a UDP connection?
- Make sure there is a route to the destination (call the IP routing routines if the route is not already established; fail if there is no route)
- Create a UDP header (for the packet)
- Call the IP build and transmit function

5.2.3 Creating a Packet with TCP

- Check connection - is it established? is it open? is the socket working?
- Check for and combine data with partial packets if possible
- Create a packet buffer
- Copy the payload from user space
- Add the packet to the outbound queue
- Build current TCP header into packet (with ACKs, SYN, etc.)
- Call the IP transmit function

5.2.4 Wrapping a Packet in IP

- Create a packet buffer (if necessary - UDP)
- Look up route to destination (if necessary - TCP)
- Fill in the packet IP header
- Copy the transport header and the payload from user space
- Send the packet to the destination route's device output funtion

5.2.5 Transmitting a Packet

- Put the packet on the device output queue
- Wake up the device
- Wait for the scheduler to run the device driver
- Test the medium (device)
- Send the link header
- Tell the bus to transmit the packet over the medium

5.3 Linux Functions

The following is an alphabetic list of the Linux kernel functions that are most important to message traffic, where they are in the source code, and what they do. To follow function calls, begin with `sock_write()`.

```
dev_queue_xmit() - net/core/dev.c (579)
  calls start_bh_atomic()
  if device has a queue
    calls enqueue() to add packet to queue
    calls qdisc_wakeup() [= qdisc_restart()] to wake device
  else calls hard_start_xmit()
  calls end_bh_atomic()

DEVICE->hard_start_xmit() - device dependent, drivers/net/DEVICE.c
  tests to see if medium is open
  sends header
  tells bus to send packet
  updates status

inet_sendmsg() - net/ipv4/af_inet.c (786)
  extracts pointer to socket sock
  checks socket to make sure it is working
  verifies protocol pointer
  returns sk->prot[tcp/udp]->sendmsg()

ip_build_xmit - net/ipv4/ip_output.c (604)
  calls sock_alloc_send_skb() to establish memory for skb
  sets up skb header
  calls getfrag() [= udp_getfrag()]  to copy buffer from user space
  returns rt->u.dst.output() [= dev_queue_xmit()]

ip_queue_xmit() - net/ipv4/ip_output.c (234)
  looks up route
  builds IP header
  fragments if required
  adds IP checksum
  calls skb->dst->output() [= dev_queue_xmit()]

qdisc_restart() - net/sched/sch_generic.c (50)
  pops packet off queue
  calls dev->hard_start_xmit()
  updates status
  if there was an error, requeues packet

sock_sendmsg() - net/socket.c (325)
  calls scm_sendmsg() [socket control message]
  calls sock->ops[inet]->sendmsg() and destroys scm

>>> sock_write() - net/socket.c (399)
  calls socki_lookup() to associate socket with fd/file inode
  creates and fills in message header with data size/addresses
  returns sock_sendmsg()

tcp_do_sendmsg() - net/ipv4/tcp.c  (755)
  waits for connection, if necessary
  calls skb_tailroom() and adds data to waiting packet if possible
  checks window status
  calls sock_wmalloc() to get memory for skb
  calls csum_and_copy_from_user() to copy packet and do checksum
  calls tcp_send_skb()
```

```
tcp_send_skb() - net/ipv4/tcp_output.c (160)
  calls __skb_queue_tail() to add packet to queue
  calls tcp_transmit_skb() if possible

tcp_transmit_skb() - net/ipv4/tcp_output.c (77)
  builds TCP header and adds checksum
  calls tcp_build_and_update_options()
  checks ACKs,SYN
  calls tp->af_specific[ip]->queue_xmit()

tcp_v4_sendmsg() - net/ipv4/tcp_ipv4.c (668)
  checks for IP address type, opens connection, port addresses
  returns tcp_do_sendmsg()

udp_getfrag() - net/ipv4/udp.c (516)
  copies and checksums a buffer from user space

udp_sendmsg() - net/ipv4/udp.c (559)
  checks length, flags, protocol
  sets up UDP header and address info
  checks multicast
  fills in route
  fills in remainder of header
  calls ip_build_xmit()
  updates UDP status
  returns err
```

Chapter 6
Receiving Messages

This chapter presents the receiving side of message trafficking. It provides an overview of the process, examines the layers packets travel through, details the actions of each layer, and summarizes the implementation code within the kernel.

6.1 Overview

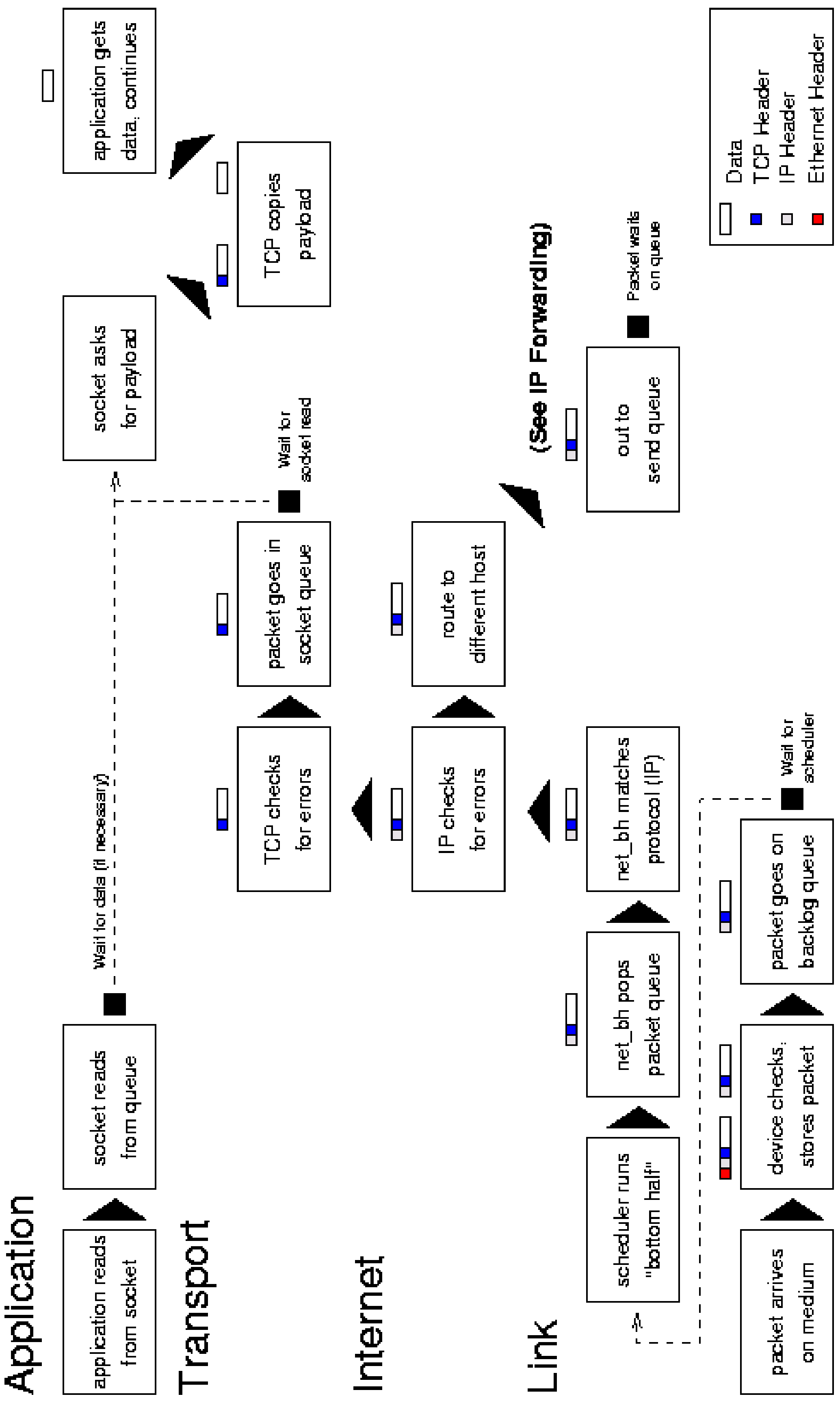


Figure 6.1: Receiving messages.

An incoming message begins with an interrupt when the system notifies the device that a message is ready. The device allocates storage space and tells the bus to put the message into that space. It then passes the packet to the link layer, which puts it on the backlog queue, and marks the network flag for the next ``bottom-half'' run.

The bottom-half is a Linux system that minimizes the amount of work done during an interrupt. Doing a lot of processing during an interrupt is not good precisely because it interrupts a running process; instead, interrupt handlers have a ``top-half'' and a ``bottom-half''. When the interrupt arrives, the top-half runs and takes care of any critical operations, such as moving data from a device queue into kernel memory. It then marks a flag that tells the kernel that there is more work to do - when the processor has time - and returns control to the current process. The next time the process scheduler runs, it sees the flag, does the extra work, and only then schedules any normal processes.

When the process scheduler sees that there are networking tasks to do it runs the network bottom-half. This function pops packets off of the backlog queue, matches them to a known protocol (typically IP), and passes them to that protocol's receive function. The IP layer examines the packet for errors and routes it; the packet will go into an outgoing queue (if it is for another host) or up to the transport layer (such as TCP or UDP). This layer again checks for errors, looks up the socket associated with the port specified in the packet, and puts the packet at the end of that socket's receive queue.

Once the packet is in the socket's queue, the socket will wake up the application process that owns it (if necessary). That process may then make or return from a `read` system call that copies the data from the packet in the queue into its own buffer. (The process may also do nothing for the time being if it was not waiting for the packet, and get the data off the queue when it needs it.)

6.2 Receiving Walk-Through

6.2.1 Reading from a Socket (Part I)

- Try to read data from a socket (application)
- Fill in message header with location of buffer (socket)
- Check for basic errors - is the socket bound to a port? can the socket accept messages? is there something wrong with the socket?
- Pass the message header with to the appropriate transport protocol (INET socket)
- Sleep until there is enough data to read from the socket (TCP/UDP)

6.2.2 Receiving a Packet

- Wake up the receiving device (interrupt)
- Test the medium (device)
- Receive the link header
- Allocate space for the packet
- Tell the bus to put the packet into the buffer
- Put the packet on the backlog queue
- Set the flag to run the network bottom half when possible
- Return control to the current process

6.2.3 Running the Network ``Bottom Half''

- Run the network bottom half (scheduler)
- Send any packets that are waiting to prevent interrupts (bottom half)
- Loop through all packets in the backlog queue and pass the packet up to its Internet reception protocol

- IP

- Flush the sending queue again
- Exit the bottom half

6.2.4 Unwrapping a Packet in IP

- Check packet for errors - too short? too long? invalid version? checksum error?
- Defragment the packet if necessary
- Get the route for the packet (could be for this host or could need to be forwarded)
- Send the packet to its destination handling routine (TCP or UDP reception, or possibly retransmission to another host)

6.2.5 Accepting a Packet in UDP

- Check UDP header for errors
- Match destination to socket
- Send an error message back if there is no such socket
- Put packet into appropriate socket receive queue
- Wake up any processes waiting for data from that socket

6.2.6 Accepting a Packet in TCP

- Check sequence and flags; store packet in correct space if possible
- If already received, send immediate ACK and drop packet
- Determine which socket packet belongs to
- Put packet into appropriate socket receive queue
- Wake up and processes waiting for data from that socket

6.2.7 Reading from a Socket (Part II)

- Wake up when data is ready (socket)
- Call transport layer receive function
- Move data from receive queue to user buffer (TCP/UDP)
- Return data and control to application (socket)

6.3 Linux Functions

The following is an alphabetic list of the Linux kernel functions that are most important to receiving traffic, where they are in the source code, and what they do. To follow functions calls from the network up, start with `DEVICE_rx()`. To follow functions calls from the application down, start with `sock_read()`.

```
>>> DEVICE_rx() - device dependent, drivers/net/DEVICE.c
  (gets control from interrupt)
  performs status checks to make sure it should be receiving
  calls dev_alloc_skb() to reserve space for packet
  gets packet off of system bus
  calls eth_type_trans() to determine protocol type
  calls netif_rx()
  updates card status
  (returns from interrupt)
```

```
inet_recvmsg() - net/ipv4/af_inet.c (764)
  extracts pointer to socket sock
  checks socket to make sure it is accepting
  verifies protocol pointer
  returns sk->prot[tcp/udp]->recvmsg()

ip_rcv() - net/ipv4/ip_input.c (395)
  examines packet for errors:
    invalid length (too short or too long)
    incorrect version (not 4)
    invalid checksum
  calls __skb_trim() to remove padding
  defrags packet if necessary
  calls ip_route_input() to route packet
  examines and handle IP options
  returns skb->dst->input() [= tcp_rcv,udp_rcv()]

net_bh() - net/core/dev.c (835)
  (run by scheduler)
  if there are packets waiting to go out, calls qdisc_run_queues()
      (see sending section)
  while the backlog queue is not empty
    let other bottom halves run
    call skb_dequeue() to get next packet
    if the packet is for someone else (FASTROUTED) put onto send queue
    loop through protocol lists (taps and main) to match protocol type
    call pt_prev->func() [= ip_rcv()] to pass packet to appropriate
        protocol
  call qdisc_run_queues() to flush output (if necessary)

netif_rx() - net/core/dev.c (757)
  puts time in skb->stamp
  if backlog queue is too full, drops packet
  else
    calls skb_queue_tail() to put packet into backlog queue
    marks bottom half for later execution

sock_def_readable() - net/core/sock.c (989)
  calls wake_up_interruptible() to put waiting process on run queue
  calls sock_wake_async() to send SIGIO to socket process

sock_queue_rcv_skb() - include/net/sock.h (857)
  calls skb_queue_tail() to put packet in socket receive queue
  calls sk->data_ready() [= sock_def_readable()]

>>> sock_read() - net/socket.c (366)
  sets up message headers
  returns sock_recvmsg() with result of read

sock_recvmsg() - net/socket.c (338)
  reads socket management packet (scm) or packet by
      calling sock->ops[inet]->recvmsg()

tcp_data() - net/ipv4/tcp_input.c (1507)
  shrinks receive queue if necessary
  calls tcp_data_queue() to queue packet
  calls sk->data_ready() to wake socket

tcp_data_queue() - net/ipv4/tcp_input.c (1394)
  if packet is out of sequence:
    if old, discards immediately
    else calculates appropriate storage location
    calls __skb_queue_tail() to put packet in socket receive queue
    updates connection state

  tcp_rcv_established() - net/ipv4/tcp_input.c (1795)
    if fast path
      checks all flags and header info
      sends ACK
      calls _skb_queue_tail() to put packet in socket receive queue
    else (slow path)
      if out of sequence, sends ACK and drops packet
      check for FIN, SYN, RST, ACK
      calls tcp_data() to queue packet
      sends ACK

  tcp_recvmsg() - net/ipv4/tcp.c (1149)
    checks for errors
    wait until there is at least one packet available
    cleans up socket if connection closed
    calls memcpy_toiovec() to copy payload from the socket buffer into
        the user space
    calls cleanup_rbuf() to release memory and send ACK if necessary
    calls remove_wait_queue() to wake process (if necessary)

  udp_queue_rcv_skb() - net/ipv4/udp.c (963)
    calls sock_queue_rcv_skb()
    updates UDP status (frees skb if queue failed)

  udp_rcv() - net/ipv4/udp.c (1062)
    gets UDP header, trims packet, verifies checksum (if required)
    checks multicast
    calls udp_v4_lookup() to match packet to socket
    if no socket found, send ICMP message back, free skb, and stop
    calls udp_deliver() [= udp_queue_rcv_skb()]

  udp_recvmsg() - net/ipv4/udp.c (794)
    calls skb_recv_datagram() to get packet from queue
    calls skb_copy_datagram_iovec() to move the payload from the socket buffer
        into the user space
    updates the socket timestamp
    fills in the source information in the message header
    frees the packet memory
```

Chapter 7
IP Forwarding

This chapter presents the pure routing side (by IP forwarding) of message traffic. It provides an overview of the process, examines the layers packets travel through, details the actions of each layer, and summarizes the implementation code within the kernel.

7.1 Overview

See Figure 7.1 for an abstract diagram of the the forwarding process. (It is essentially a combination of the receiving and sending processes.)

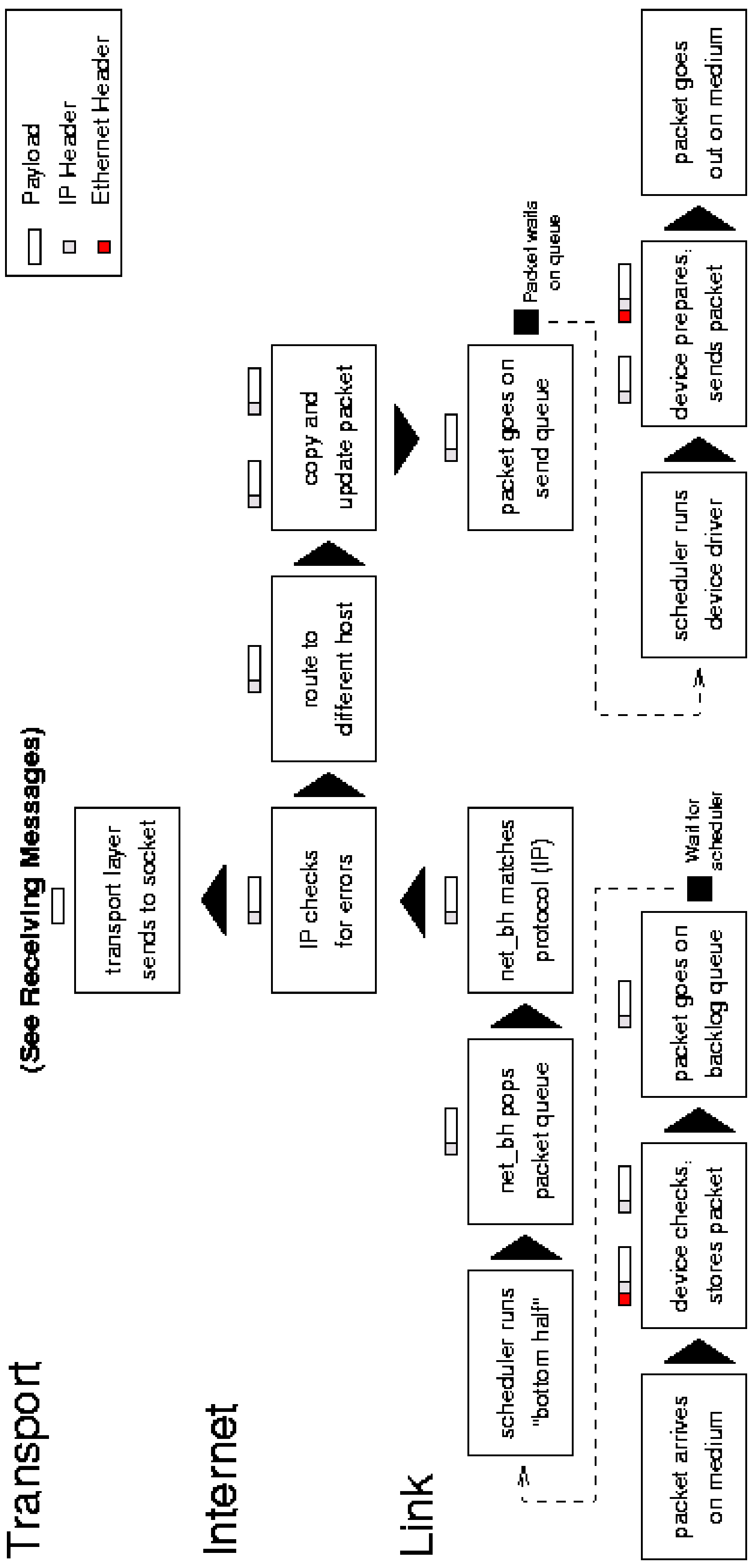

Payload
IP Header
Ethernet Header
Transport
Internet
Link
(See Receiving Messages)
transport layer sends to socket
IP checks for errors
route to different host
copy and update packet
packet goes on send queue
Packet waits on queue
net_bh matches protocol (IP)
net_bh pops packet queue
scheduler runs "bottom half"
packet arrives on medium
device checks; stores packet
packet goes on backlog queue
Wait for scheduler
scheduler runs device driver
device prepares; sends packet
packet goes out on medium

Figure 7.1: IP forwarding.

A forwarded packet arrives with an interrupt when the system notifies the device that a message is ready. The device allocates storage space and tells the bus to put the message into that space. It then passes the packet to the link layer, which puts it on the backlog queue, marks the network flag for the next ``bottom-half'' run, and returns control to the current process.

When the process scheduler next runs, it sees that there are networking tasks to do and runs the network ``bottom-half''. This function pops packets off of the backlog queue, matches them to IP, and passes them to the receive function. The IP layer examines the packet for errors and routes it; the packet will go up to the transport layer (such as TCP or UDP if it is for this host) or sideways to the IP forwarding function. Within the forwarding function, IP checks the packet and sends an ICMP message back to the sender if anything is wrong. It then copies the packet into a new buffer and fragments it if necessary.

Finally the IP layer passes the packet to the link layer function, which moves the packet onto the sending device's `xmit` queue and makes sure the device knows that it has traffic to send. Finally, the device (such as a network card) tells the bus to send the packet.

7.2 IP Forward Walk-Through

7.2.1 Receiving a Packet

- Wake up the receiving device (interrupt)
- Test the medium (device)
- Receive the link header
- Allocate space for the packet
- Tell the bus to put the packet into the buffer
- Put the packet on the backlog queue
- Set the flag to run the network bottom half when possible
- Return control to the current process

7.2.2 Running the Network ``Bottom Half''

- Run the network bottom half (scheduler)
- Send any packets that are waiting to prevent interrupts (net_bh)
- Loop through all packets in the backlog queue and pass the packet up to its Internet reception protocol - IP
- Flush the sending queue again
- Exit the bottom half

7.2.3 Examining a Packet in IP

- Check packet for errors - too short? too long? invalid version? checksum error?
- Defragment the packet if necessary
- Get the route for the packet (could be for this host or could need to be forwarded)
- Send the packet to its destination handling routine (retransmission to another host in this case)

7.2.4 Forwarding a Packet in IP

- Check TTL field (and decrement it)
- Check packet for improper (undesired) routing
- Send ICMP back to sender if there are any problems
- Copy packet into new buffer and free old one
- Set any IP options

- Fragment packet if it is too big for new destination
- Send the packet to the destination route's device output function

7.2.5 Transmitting a Packet

- Put the packet on the device output queue
- Wake up the device
- Wait for the scheduler to run the device driver
- Test the medium (device)
- Send the link header
- Tell the bus to transmit the packet over the medium

7.3 Linux Functions

The following is an alphabetic list of the Linux kernel functions that are most important to IP forwarding, where they are in the source code, and what they do. To follow the functions calls, start with DEVICE_rx().

```
dev_queue_xmit() - net/core/dev.c (579)
  calls start_bh_atomic()
  if device has a queue
    calls enqueue() to add packet to queue
    calls qdisc_wakeup() [= qdisc_restart()] to wake device
  else calls hard_start_xmit()
  calls end_bh_atomic()

DEVICE->hard_start_xmit() - device dependent, drivers/net/DEVICE.c
  tests to see if medium is open
  sends header
  tells bus to send packet
  updates status

>>> DEVICE_rx() - device dependent, drivers/net/DEVICE.c
  (gets control from interrupt)
  performs status checks to make sure it should be receiving
  calls dev_alloc_skb() to reserve space for packet
  gets packet off of system bus
  calls eth_type_trans() to determine protocol type
  calls netif_rx()
  updates card status
  (returns from interrupt)

ip_finish_output() - include/net/ip.h (140)
  sets sending device to output device for given route
  calls output function for destination [= dev_queue_xmit()]

ip_forward() - net/ipv4/ip_forward.c (72)
  checks for router alert
  if packet is not meant for any host, drops it
  if TTL has expired, drops packet and sends ICMP message back
  if strict route cannot be followed, drops packet and sends ICMP
      message back to sender
  if necessary, sends ICMP message telling sender packet is redirected
  copies and releases old packet
  decrements TTL
  if there are options, calls ip_forward_options() to set them
  calls ip_send()
```

```
ip_rcv() - net/ipv4/ip_input.c (395)
  examines packet for errors:
    invalid length (too short or too long)
    incorrect version (not 4)
    invalid checksum
  calls __skb_trim() to remove padding
  defrags packet if necessary
  calls ip_route_input() to route packet
  examines and handle IP options
  returns skb->dst->input() [= ip_forward()]

ip_route_input() - net/ipv4/route.c (1366)
  calls rt_hash_code() to get index for routing table
  loops through routing table (starting at hash) to find match for packet
  if it finds match:
    updates stats for route (time and usage)
    sets packet destination to routing table entry
    returns success
  else
    checks for multicasting addresses
    returns result of ip_route_input_slow() (attempted routing)

ip_route_output_slow() - net/ipv4/route.c (1421)
  if source address is known, looks up output device
  if destination address is unknown, set up loopback
  calls fib_lookup() to find route
  allocates memory new routing table entry
  initializes table entry with source, destination, TOS, output device,
      flags
  calls rt_set_nexthop() to find next destination
  returns rt_intern_hash(), which installs route in routing table

ip_send() - include/net/ip.h (162)
  calls ip_fragment() if packet is too big for device
  calls ip_finish_output()

net_bh() - net/core/dev.c (835)
  (run by scheduler)
  if there are packets waiting to go out, calls qdisc_run_queues()
      (see sending section)
  while the backlog queue is not empty
    let other bottom halves run
    call skb_dequeue() to get next packet
    if the packet is for someone else (FASTROUTED) put onto send queue
    loop through protocol lists (taps and main) to match protocol type
    call pt_prev->func() [= ip_rcv()] to pass packet to appropriate
        protocol
  call qdisc_run_queues() to flush output (if necessary)

netif_rx() - net/core/dev.c (757)
  puts time in skb->stamp
  if backlog queue is too full, drops packet
  else
    calls skb_queue_tail() to put packet into backlog queue
    marks bottom half for later execution

qdisc_restart() - net/sched/sch_generic.c (50)
  pops packet off queue
  calls dev->hard_start_xmit()
  updates status
  if there was an error, requeues packet
```

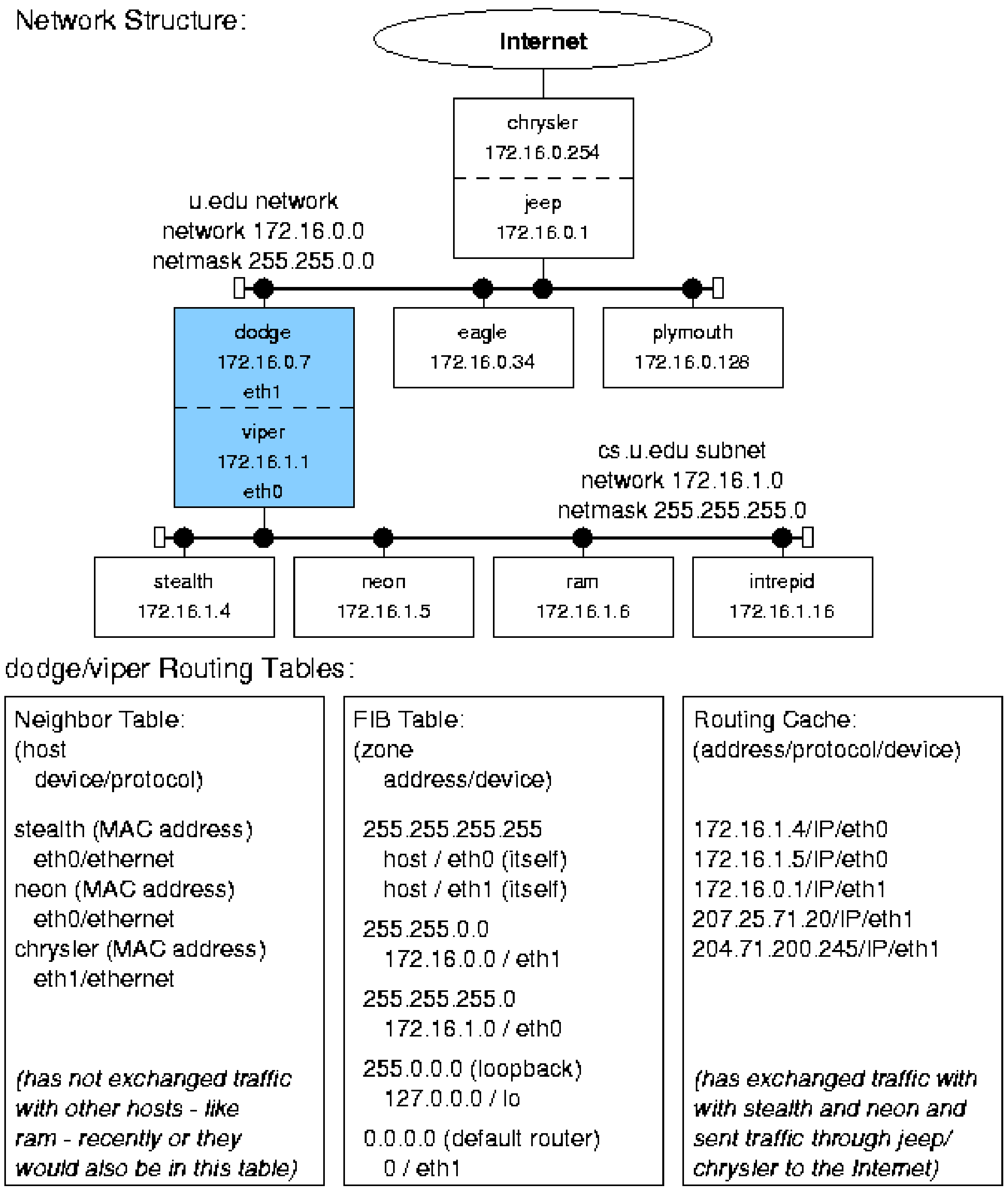
```
rt_intern_hash() - net/ipv4/route.c (526)
  puts new route in routing table
```

Chapter 8
Basic Internet Protocol Routing

This chapter presents the basics of IP Routing. It provides an overview of how routing works, examines how routing tables are established and updated, and summarizes the implementation code within the kernel.

8.1 Overview

Linux maintains three sets of routing data - one for computers that are directly connected to the host (via a LAN, for example) and two for computers that are only indirectly connected (via IP networking). Examine Figure 8.1 to see how entries for a computer in the general example might look.



Neighbor Table:
(host
device/protocol)

stealth (MAC address)
eth0/ethernet
neon (MAC address)
eth0/ethernet
chrysler (MAC address)
eth1/ethernet

(has not exchanged traffic with other hosts - like ram - recently or they would also be in this table)

FIB Table:
(zone
address/device)

255.255.255.255
host / eth0 (itself)
host / eth1 (itself)

255.255.0.0
172.16.0.0 / eth1

255.255.255.0
172.16.1.0 / eth0

255.0.0.0 (loopback)
127.0.0.0 / lo

0.0.0.0 (default router)
0 / eth1

Routing Cache:
(address/protocol/device)

172.16.1.4/IP/eth0
172.16.1.5/IP/eth0
172.16.0.1/IP/eth1
207.25.71.20/IP/eth1
204.71.200.245/IP/eth1

(has exchanged traffic with with stealth and neon and sent traffic through jeep/ chrysler to the Internet)

Figure 8.1: General routing table example.

The neighbor table contains address information for computers that are physically connected to the host (hence the name ``neighbor''). It includes information on which device connects to which neighbor and what protocols to use in exchanging data. Linux uses the Address Resolution Protocol (ARP) to maintain and update this table; it is dynamic in that entries are added when needed but eventually disappear if not used again within a certain time. (However, administrators can set up entries to be permanent if doing so makes sense.)

Linux uses two complex sets of routing tables to maintain IP addresses: an all-purpose Forwarding Information Base (FIB) with directions to every possible address, and a smaller (and faster) routing cache with data on frequently used routes. When an IP packet needs to go to a distant host, the IP layer first checks the routing cache for an entry with the appropriate source, destination, and type of service. If there is such an entry, IP uses it. If not, IP requests the routing information from the more complete (but slower) FIB, builds a new cache entry with that data, and then uses the new entry. While the FIB entries are semi-permanent (they usually change only when routers come up or go down) the cache entries remain only until they become obsolete (they are unused for a ``long'' period).

8.2 Routing Tables

Note: within these tables, there are references to variables of types such as `u32` (host byte order) and `__u32` (network byte order). On the Intel architecture they are both equivalent to `unsigned int`s and in point of fact they are translated (using the `ntohl` function) anyway; the type merely gives an indication of the order in which the value it contains is stored.

8.2.1 The Neighbor Table

The Neighbor Table (whose structure is shown in Figure 8.2) contains information about computers that are physically linked with the host computer. (Note that the source code uses the European spelling, ``neighbour''.) Entries are not (usually) persistent; this table may contain no entries (if the computer has not passed any network traffic recently) or may contain as many entries as there are computers physically connected to its network (if it has communicated with all of them recently). Entries in the table are actually other table structures which contain addressing, device, protocol, and statistical information.

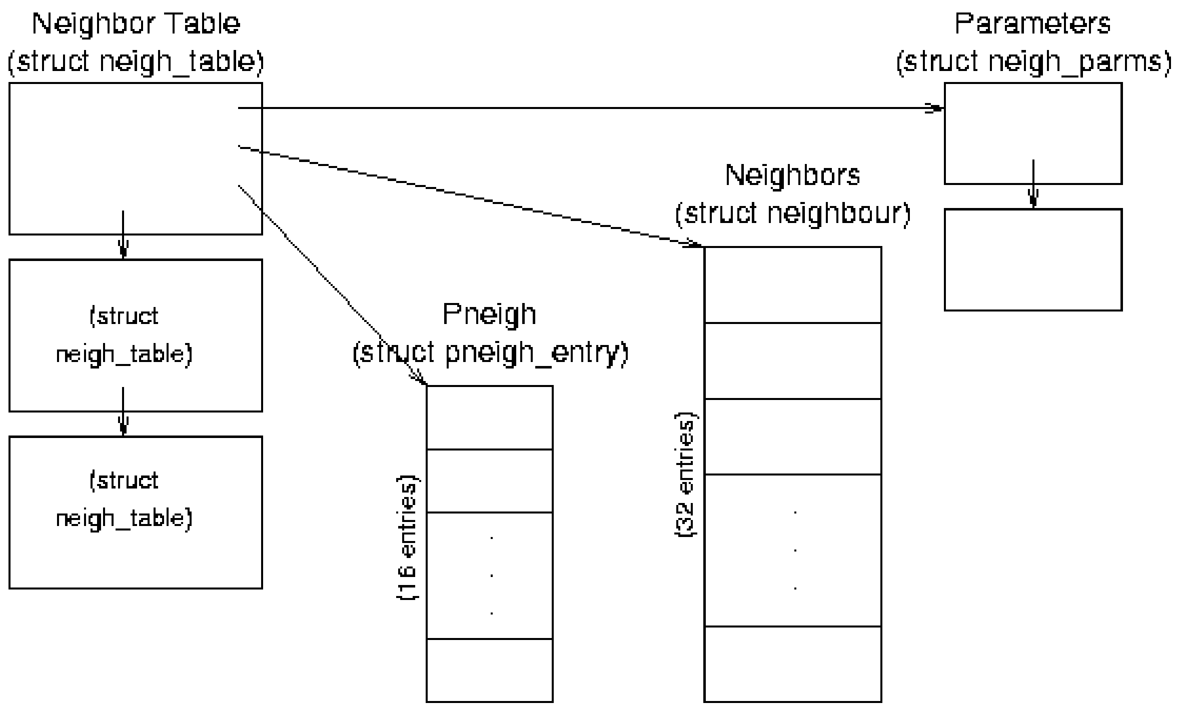


Figure 8.2: Neighbor Table data structure relationships.

`struct neigh_table *neigh_tables` - this global variable is a pointer to a list of neighbor tables; each table contains a set of general functions and data and a hash table of specific information about a set of neighbors. This is a very detailed, low level table containing specific information such as the approximate transit time for messages, queue sizes, device pointers, and pointers to device functions.

Neighbor Table (`struct neigh_table`) Structure - this structure (a list element) contains common neighbor information and table of neighbor data and pneigh data. All computers connected through a single type of connection (such as a single Ethernet card) will be in the same table.

- `struct neigh_table *next` - pointer to the next table in the list.
- `struct neigh_parms parms` - structure containing message travel time, queue length, and statistical information; this is actually the head of a list.
- `struct neigh_parms *parms_list` - pointer to a list of information structures.
- `struct neighbour *hash_buckets[]` - hash table of neighbors associated with this table; there are `NEIGH_HASHMASK+1` (32) buckets.
- `struct pneigh_entry *phash_buckets[]` - hash table of structures containing device pointers and keys; there are `PNEIGH_HASHMASK+1` (16) buckets.
- Other fields include timer information, function pointers, locks, and statistics.

Neighbor Data (`struct neighbour`) Structure - these structures contain the specific information about each neighbor.

- `struct device *dev` - pointer to the device that is connected to this neighbor.
- `__u8 nud_state` - status flags; values can be incomplete, reachable, stale, etc.; also contains state information for permanence and ARP use.
- `struct hh_cache *hh` - pointer to cached hardware header for transmissions to this neighbor.
- `struct sk_buff_head arp_queue` - pointer to ARP packets for this neighbor.
- Other fields include list pointers, function (table) pointers, and statistical information.

8.2.2 The Forwarding Information Base

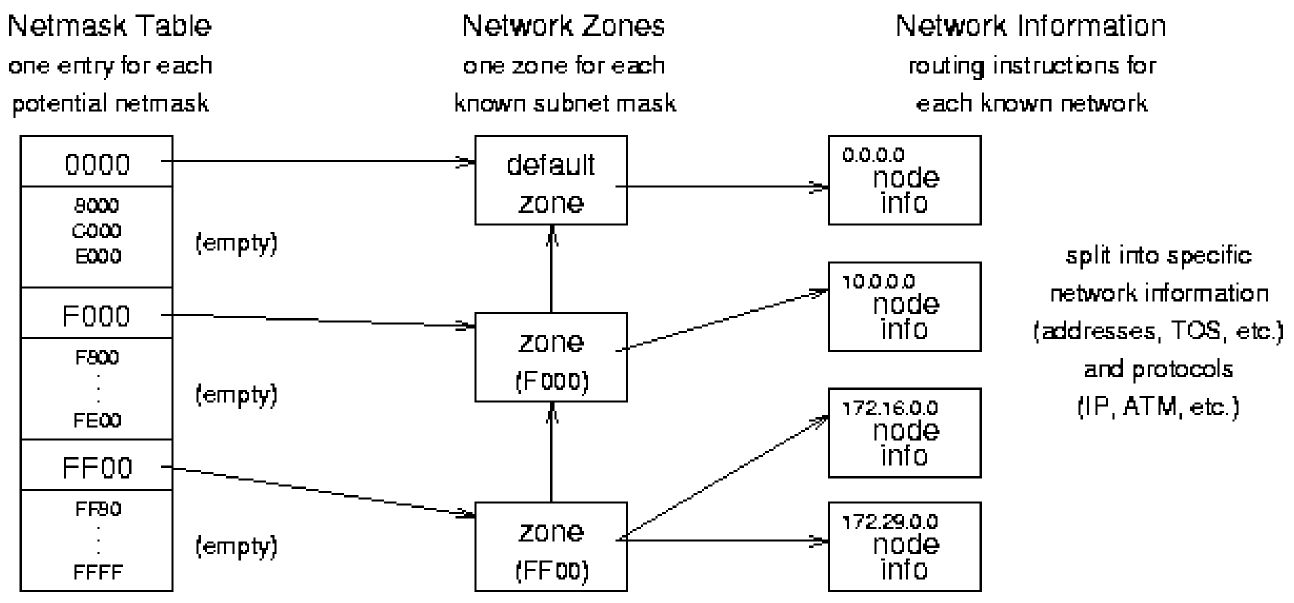


Figure 8.3: Forwarding Information Base (FIB) conceptual organization.

The Forwarding Information Base (FIB) is the most important routing structure in the kernel; it is a complex structure that contains the routing information needed to reach any valid IP address by its network mask. Essentially it is a large table with general address information at the top and very specific information at the bottom. The IP layer enters the table with the destination address of a packet and compares it to the most specific netmask to see if they match. If they do not, IP goes on to the next most general netmask and again compares the two. When it finally finds a match, IP copies the ``directions'' to the distant host into the routing cache and sends the packet on its way. See Figures 8.3 and 8.4 for the organization and data structures used in the FIB - note that Figure 8.3 shows some different FIB capabilities, like two sets of network information for a single zone, and so does not follow the general example.)

`struct fib_table *local_table, *main_table` - these global variables are the access points to the FIB tables; they point to table structures that point to hash tables that point to zones. The contents of the `main_table` variable are in */proc/net/route*.

FIB Table `fib_table` Structure - *include/net/ip_fib.h* - these structures contain function jump tables and each points to a hash table containing zone information. There are usually only one or two of these.

- `int (*tb_functions)()` - pointers to table functions (lookup, delete, insert, etc.) that are set during initialization to `fn_hash_function()`.
- `unsigned char tb_data[0]` - pointer to the associated FIB hash table (despite its declaration as a character array).
- `unsigned char tb_id` - table identifier; 255 for `local_table`, 254 for `main_table`.
- `unsigned tb_stamp`

Netmask Table `fn_hash` Structure - *net/ipv4/fib_hash.c* - these structures contain pointers to the individual zones, organized by netmask. (Each zone corresponds to a uniquely specific network mask.) There is one of these for each FIB table (unless two tables point to the same hash table).

- `struct fn_zone *fn_zones[33]` - pointers to zone entries (one zone for each bit in the mask; `fn_zone[0]` points to the zone for netmask 0x0000, `fn_zone[1]` points to the zone for 0x8000, and `fn_zone[32]` points to the zone for 0xFFFF.
- `struct fn_zone *fn_zone_list` - pointer to first (most specific) non-empty zone in the list; if there is an entry for netmask 0xFFFF it will point to that zone, otherwise it may point to zone 0xFFF0 or 0xFF00 or 0xF000 etc.

Network Zone `fn_zone` Structure - *net/ipv4/fib_hash.c* - these structures contain some hashing information and pointers to hash tables of nodes. There is one of these for each known netmask.

- `struct fn_zone *fz_next` - pointer to the next non-empty zone in the hash structure (the next most general netmask; e.g., `fn_hash- > fn_zone[28]- > fz_next` might point to `fn_hash- > fn_zone[27]`).
- `struct fib_node **fz_hash` - pointer to a hash table of nodes for this zone.
- `int fz_nent` - the number of entries (nodes) in this zone.
- `int fx_divisor` - the number of buckets in the hash table associated with this zone; there are 16 buckets in the table for most zones (except the first zone - 0000 - the loopback device).
- `u32 fz_hashmask` - a mask for entering the hash table of nodes; 15 (0x0F) for most zones, 0 for zone 0).
- `int fz_order` - the index of this zone in the parent `fn_hash` structure (0 to 32).
- `u32 fz_mask` - the zone netmask defined as `~((1<<(32-fz_order))-1)`; for example, the first (zero) element is 1 shifted left 32 minus 0 times (0x10000), minus 1 (0xFFFF), and complemented (0x0000). The second element has a netmask of 0x8000, the next 0xC000, the next 0xE000, 0xF000, 0xF800, and so on to the last (32d) element whose mask is 0xFFFF.

Network Node Information `fib_node` Structure - *net/ipv4/fib_hash.c* - these structures contain the information unique to each set of addresses and a pointer to information about common features (such as device and protocols); there is one for each known network (unique source/destination/TOS combination).

- `struct fib_node *fn_next` - pointer to the next node.
- `struct fib_info *fn_info` - pointer to more information about this node (that is shared by many nodes).
- `fn_key_t fn_key` - hash table key - the least significant 8 bits of the destination address (or 0 for the loopback device).
- Other fields include specific information about this node (like `fn_tos` and `fn_state`).

Network Protocol Information (`fib_info`) Structure - *include/net/ip_fib.h* - these structures contain protocol and hardware information that are specific to an interface and therefore common to many potential zones; several networks may be addressable through the same interface (like the one that leads to the rest of the Internet). There is one of these for each interface.

- `fib_protocol` - index to a network protocol (e.g., IP) used for this route.
- `struct fib_nh fib_nh[0]` - contains a pointer to the device used for sending or receiving traffic for this route.
- Other fields include list pointers and statistical and reference data (like `fib_refcnt` and `fib_flags`.

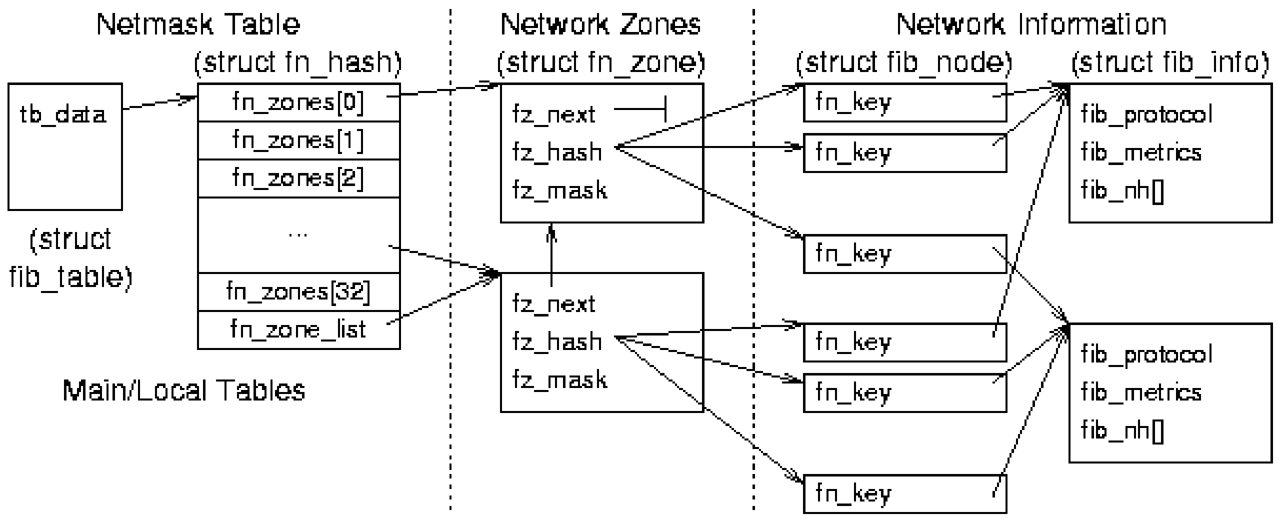


Figure 8.4: Forwarding Information Base (FIB) data relationships.

FIB Traversal Example:

1. `ip_route_output_slow()` (called because the route is not in the routing cache) sets up an `rt_key` structure with a source address of 172.16.0.7, a destination address of 172.16.0.34, and a TOS of 2.
2. `ip_route_output_slow()` calls `fib_lookup()` and passes it the key to search for.
3. `fib_lookup()` calls `local_table- > tb_lookup()` (which is a reference to the `fn_hash_lookup` function) to make the local table find the key.
4. `fn_hash_lookup()` searches the local table's hash table, starting in the most specific zone - 24 (netmask 255.255.255.0 dotted decimal) (pointed to by the `fn_zone_list` variable).
5. `fz_key()` builds a test key by ANDing the destination address with the zone netmask, resulting in a key value 172.16.0.0.
6. `fz_chain()` performs the hash function (see `fn_hash()`) and ANDs this value with the zone's `fz_hashmask` (15) to get an index (6) into the zone's hash table of nodes. Unfortunately, this node is empty; there are no possible matches in this zone.
7. `fn_hash_lookup()` moves to the next non-empty zone - 16 (netmask 255.255.0.0 dotted decimal) (pointed to by the current zone's `fz_next` variable).
8. `fz_key()` builds a new test key by ANDing the destination address with this zone's netmask, resulting

in a key value of 172.16.0.0.

9. `fz_chain()` performs the hash function and ANDs this value with the zone's `fz_hashmask` (15) to get an index (10) into the zone's hash table of nodes. There is a node in that slot.
10. `fn_hash_lookup()` compares its search key to the node's key. They do not match, but the search key value is less than that of the node key, so it moves on to the next node.
11. `fn_hash_lookup()` compares its search key to the new node's key. These do match, so it does some error checking and tests for an exact match with the node and its associated info variable.
12. Since everything matches, `fn_hash_lookup()` fills in a `fib_result` structure with all the information about this route. (Otherwise it would continue checking more nodes and more zones until it finds a match or fails completely.)
13. `ip_route_output_slow()` takes the `fib_result` structure and, assuming everything is in order, creates a new routing cache entry from it.

8.2.3 The Routing Cache

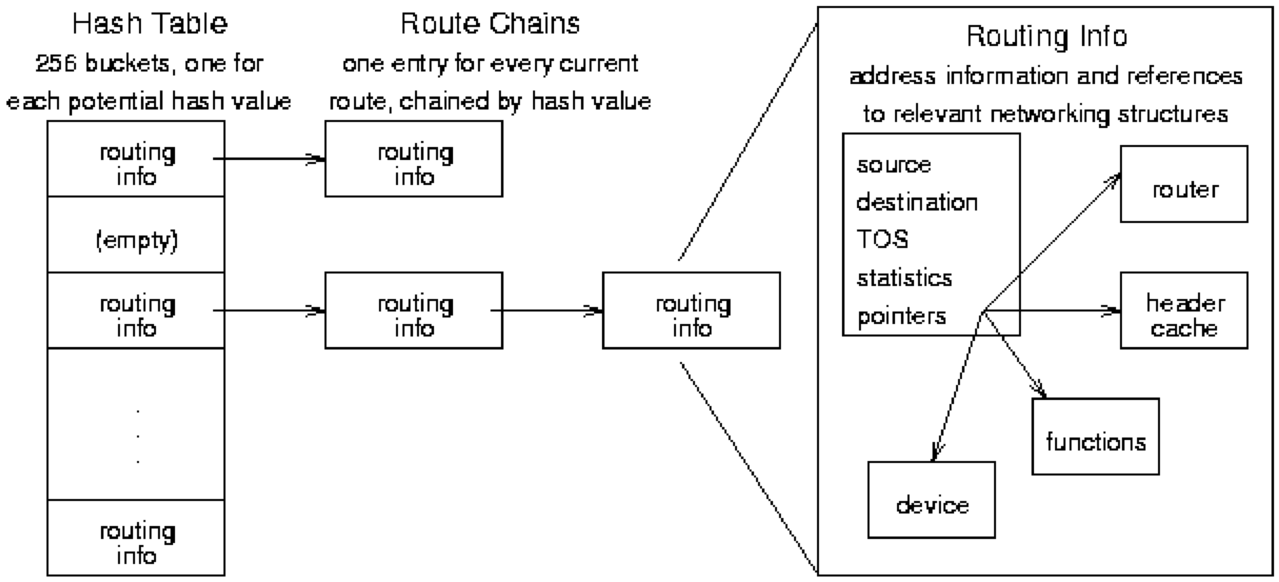


Figure 8.5: Routing Cache conceptual organization.

The routing cache is the fastest method Linux has to find a route; it keeps every route that is currently in use or has been used recently in a hash table. When IP needs a route, it goes to the appropriate hash bucket and searches the chain of cached routes until finds a match, then sends the packet along that path. (See Section 8.2.2 for what happens when the route is not yet in the cache.) Routes are chained in order, most frequently used first, and have timers and counters that remove them from the table when they are no longer in use. See Figure 8.5 for an abstract overview and Figures 8.6 and 8.7 for detailed diagrams of the data structures.

`struct rtable *rt_hash_table[RT_HASH_DIVISOR]` - this global variable contains 256 buckets of (pointers to) chains of routing cache (`rtable`) entries; the hash function combines the source address, destination address, and TOS to get an entry point to the table (between 0 and 255). The contents of this table are listed in */proc/net/rt_cache*.

Routing Table Entry (`rtable`) Structure - *include/net/route.h* - these structures contain the destination cache entries and identification information specific to each route.

- `union < struct dst_entry dst; struct rtable* rt_next) > u` - this is an entry in the table; the union structure allows quick access to the next entry in the table by overusing the `rtable`'s next field to point to the next cache entry if required.

- `__u32 rt_dst` - the destination address.
- `__u32 rt_src` - the source address.
- `rt_int iif` - the input interface.
- `__u32 rt_gateway` - the address of the neighbor to route through to get to a destination.
- `struct rt_key key` - a structure containing the cache lookup key (with src, dst, iif, oif, tos, and scope fields)
- Other fields contain flags, type, and other miscellaneous information.

Destination Cache (`dst_entry`) Structure - *include/net/dst.h* - these structures contain pointers to specific input and output functions and data for a route.

- `struct device *dev` - the input/output device for this route.
- `unsigned pmtu` - the maximum packet size for this route.
- `struct neighbor *neighbor` - a pointer to the neighbor (next link) for this route.
- `struct hh_cache *hh` - a pointer to the hardware header cache; since this is the same for every outgoing packet on a physical link, it is kept for quick access and reuse.
- `int (*input)(struct sk_buff*)` - a pointer to the input function for this route (typically `tcp_recv()`).
- `int (*output)(struct sk_buff*)` - a pointer to the output function for this route (typically `dev_queue_xmit()`).
- `struct dst_ops *ops` - a pointer to a structure containing the family, protocol, and check, reroute, and destroy functions for this route.
- Other fields hold statistical and state information and links to other routing table entries.

Neighbor Link (`neighbor`) Structure - *include/net/neighbor.h* - these structures, one for each host that is exactly one hop away, contain pointers to their access functions and information.

- `struct device *dev` - a pointer to device that is physically connected to this neighbor.
- `struct hh_cache *hh` - a pointer to the hardware header that always precedes traffic sent to this neighbor.
- `int (*output)(struct sk_buff*)` - a pointer to the output function for this neighbor (typically `dev_queue_xmit()`?).
- `struct sk_buff_head arp_queue` - the first element in the ARP queue for traffic concerning this neighbor - incoming or outgoing?
- `struct neigh_ops *ops` - a pointer to a structure that containing family data and and output functions for this link.
- Other fields hold statistical and state information and references to other neighbors.

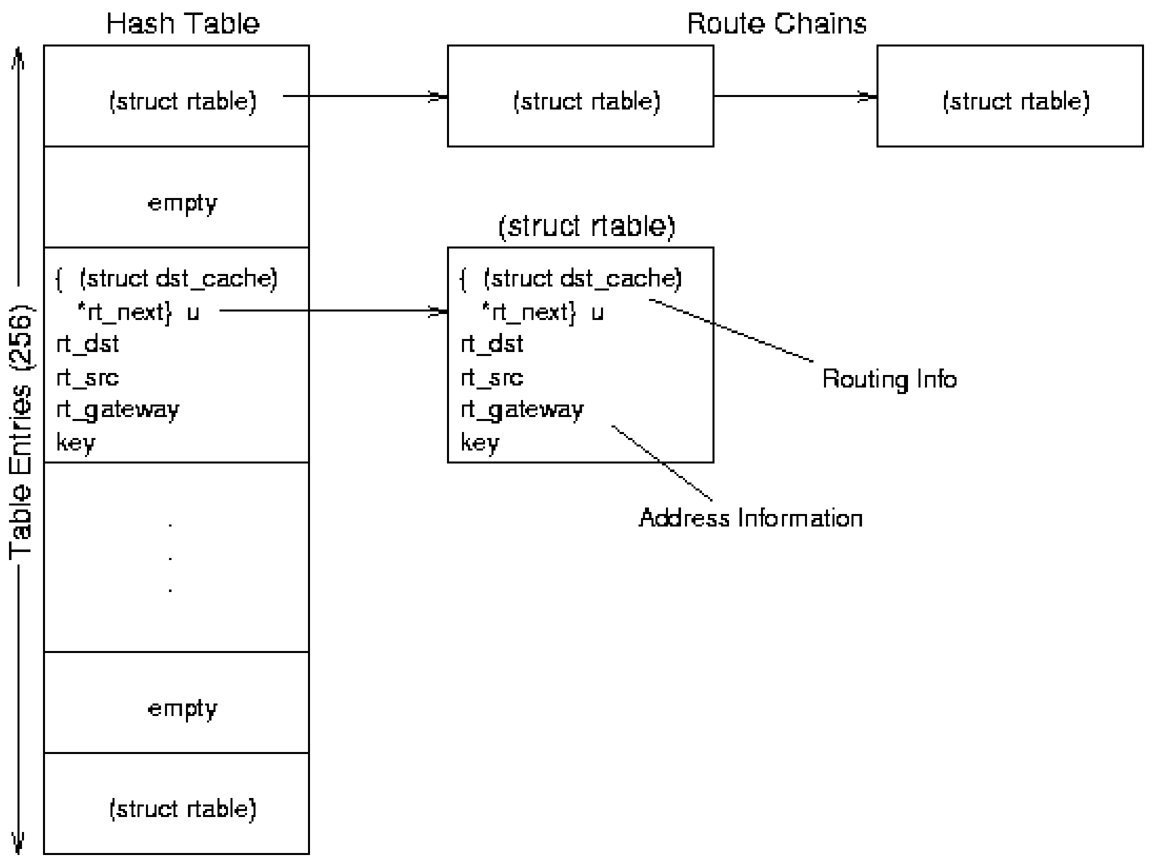


Figure 8.6: Routing Cache data structure relationships.

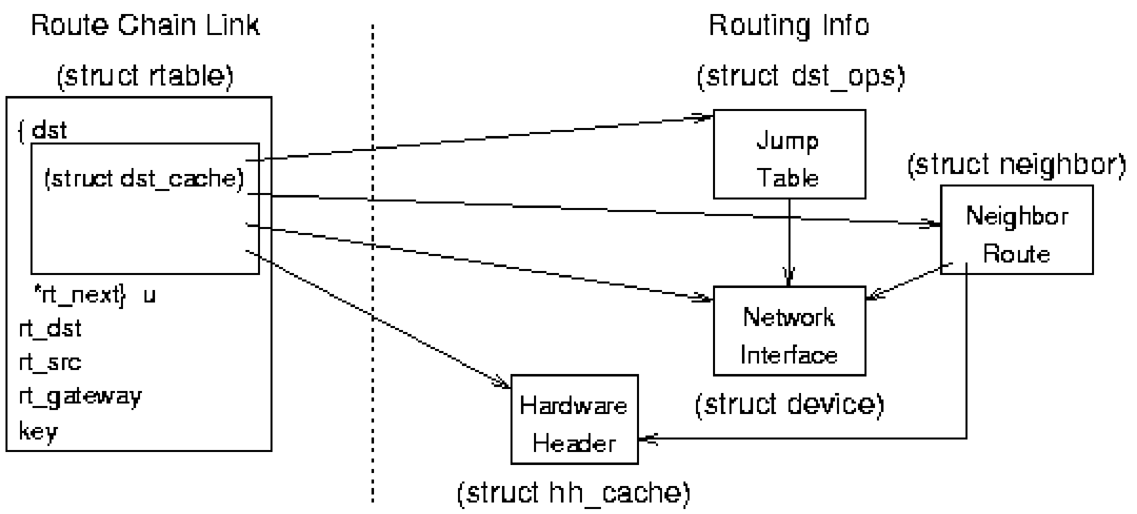


Figure 8.7: Destination Cache data structure relationships.

Routing Cache Traversal Example:

1. `ip_route_output()` (called to find a route) calls `rt_hash_code()` with a source address of 172.16.1.1, a destination address of 172.16.1.6, and a TOS of 2.
2. `rt_hash_code()` performs a hash function on the source, destination, and TOS and ANDs the result with 255 to get an entry into the hash table (5).
3. `ip_route_output()` enters the hash table at index 5. There is an entry there, but the destination addresses do not match.

4. `ip_route_output()` moves to the next entry (pointed to by the `u.rt_next` field of the last entry). This one matches in every case - destination address, source address, `iif` of 0, matching `oif`, and acceptable TOS.
5. `ip_route_output()` updates the statistics in the newfound `dst_cache` structure of the table entry, sets a pointer for the calling function to refer to the route, and returns a 0 indicating success.

8.2.4 Updating Routing Information

Linux only updates routing information when necessary, but the tables change in different manners. The routing cache is the most volatile, while the FIB usually does not change at all.

The neighbor table changes as network traffic is exchanged. If a host needs to send something to an address that is on the local subnet but not already in the neighbor table, it simply broadcasts an ARP request and adds a new entry in the neighbor table when it gets a reply. Periodically entries time out and disappear; this cycle continues indefinitely (unless a route has been specifically marked as ARP permanent). The kernel handles most changes automatically.

The FIB on most hosts and even routers remains static; it is filled in during initialization with every possible zone to route through all connected routers and never changes unless one of the routers goes down. (See Chapter 9 for details on IP routing daemons). Changes have to come through external `ioctl()` calls to add or delete zones.

The routing cache changes frequently depending on message traffic. If a host needs to send packets to a remote address, it looks up the address in the routing cache (and FIB if necessary) and sends the packet off through the appropriate router. On a host connected to a LAN with one router to the Internet, every entry will point to either a neighbor or the router, but there may be many entries that point to the router (one for each distant address). The entries are created as connections are made and time out quickly when traffic to that address stops flowing. Everything is done with IP level calls to create routes and kernel timers to delete them.

8.3 Linux Functions

The following is an alphabetic list of the Linux kernel functions that are most important to routing, where they are in the source code, and what they do.

```
arp_rcv() - net/ipv4/arp.c (542)
  checks for errors (non-ARP device, no device, packet not for host,
      device type does not match, etc.)
  check for operation - only understands REPLY and REQUEST
  extracts data from packet
  check for bad requests - loopback or multicast addresses
  checks for duplicate address detection packet (sends reply if necessary)
  if the message is a request and ip_route_input() is true:
    if the packet is a local one:
      calls neigh_event_ns() to look up and update neighbor that sent packet
      checks for hidden device (does not reply if hidden)
      sends reply with the device address
    otherwise:
      calls neigh_event_ns() to look up and update neighbor that sent packet
      calls neigh_release()
      if necessary, calls arp_send() with the address
      otherwise calls pneigh_enqueue() and returns 0
  if the message is a reply:
    calls __neigh_lookup()
    checks to see if multiple ARP replies have come in; keeps only the
```

```
        fastest (first) one
    calls neigh_update() to update ARP entry
        calls neigh_release()
  frees the skbuffer and returns 0

arp_send() - net/ipv4/arp.c (434)
  checks to make sure device supports ARP
  allocates an skbuffer
  fills in buffer header information
  fills in the ARP information
  calls dev_queue_xmit() with the finished packet

arp_req_get() - net/ipv4/arp.c (848)
  calls __neigh_lookup() to find entry for given address
  copies data from neighbor entry to arpreq entry
  returns 0 if found or ENXIO if address not in ARP table

fib_get_procinfo() - net/ipv4/fib_frontend.c (109)
  prints header and results of main_table->fn_hash_get_info() for proc FS

fib_lookup() - include/net/ip_fib.h (153)
  calls tb_lookup() [= fn_hash_lookup()] on local_table and main_table
  if either one has an entry, it fills in fib_result and returns 0
  else returns unreachable error

fib_node_get_info() - net/ipv4/fib_semantics.c (971)
  prints fib_node and fib_info contents for proc FS

fib_validate_source() - net/ipv4/fib_frontend.c (191)
  tests incoming packet's device and address
  returns error code if something is wrong
  returns 0 if packet seems legal

fn_hash() - net/ipv4/fib_hash.c (108)
  performs a hash function on a destination address:
    u32 h = ntohl(daddr)>>(32 - fib_zone->fz_order);
    h ^= (h>>20);
    h ^= (h>>10);
    h ^= (h>>5);
    h &= FZ_HASHMASK(fz);   // FZ_HASHMASK is 15 for almost all zones

fn_hash_get_info() - net/ipv4/fib_hash.c (723)
  loops through zones in a FIB table printing fib_node_get_info() for
      proc FS

fn_hash_lookup() - net/ipv4/fib_hash.c (261)
  loops through the zones in the given table
    loops through the nodes in each zone (starting at the hash entry)
      if the netmasks (node and destination) match
          checks the TOS and node status
          calls fib_semantic_match() to check packet type
          fills in fib_result with success data and returns 0
  returns 1 if nothing matched

fn_new_zone() - net/ipv4/fib_hash.c (220)
  allocates memory (in kernel) for new zone
  allocates space for 16 node buckets for zone (except first zone -
      0.0.0.0 [loopback] - which only gets one)
  stores netmask (leftmost n bits on, where n is the position of the
      zone in the table)
  searches for more specific zone in parent table
  inserts zone into zone list (most specific zone is head)
```

```
    installs new zone into parent table
    returns new zone

  fz_chain() - net/ipv4/fib_hash.c (133)
    calls fn_hash() to get a hash value
    returns the fib_node in the fib_zone at the hash index

  ip_dev_find() - net/ipv4/fib_frontend.c (147)
    looks up and returns the device with a given address in the local table

  ip_route_connect() - include/net/route.h (140)
    calls ip_route_output() to get a destination address
    returns if the call works or generates an error
    otherwise clears the route pointer and try again

  ip_route_input() - net/ipv4/route.c (1366)
    calculates hash value for address
    runs through table (starting at hash) to find connection match
        (source, destination, TOS, and IIF/OIF)
    if there is a match, updates stats and returns routing entry
    else calls ip_route_input_slow()

  ip_route_input_slow() - net/ipv4/route.c (1097)
    creates a routing table cache key
    checks for special addresses (like loopback, broadcast, or errors)
    calls fib_lookup() to find route
    allocates memory for new routing table entry
    initializes table entry with source, destination, TOS, output device,
        flags
    calls fib_validate_source() to test packet source
    printks message and returns error if source is bad
    calls rt_set_nexthop() to find next destination (neighbor)
    returns rt_intern_hash(), which installs route in routing table

  ip_route_output() - net/ipv4/route.c (1664)
    calculates hash value for address
    runs through table (starting at hash) to find connection match
        (source, destination, TOS, and IIF/OIF)
    if there is a match, updates stats and returns routing entry
    else calls ip_route_output_slow()

  ip_route_output_slow() - net/ipv4/route.c (1421)
    creates a routing table cache key
    if source address is known, calls ip_dev_find to determine output device
    if destination address is unknown, set up loopback
    calls fib_lookup() to find route
    allocates memory for new routing table entry
    initializes table entry with source, destination, TOS, output device,
        flags
    calls rt_set_nexthop() to find next destination (neighbor)
    returns rt_intern_hash(), which installs route in routing table

  ip_rt_ioctl() - net/ipv4/fib_frontend.c (250)
    switches on SIOCADDRT or SIOCDELRT (returns EINVAL otherwise)
    verifies permission and copies argument to kernel space
    converts copied argument to an rtentry structure
    if deleting a route, calls fib_get_table() and table->delete()
    else calls fib_new_table() and table->insert()
    frees argument space and returns 0 for success

  neigh_event_ns() - net/core/neighbour.c (760)
    calls __neigh_lookup() to find up address in neighbor table
  calls neigh_update()
  returns pointer to designated neighbor

neigh_update() - net/core/neighbour.c (668)
  checks permissions to modify table
  checks neighbor status if this is not a new entry
  compares given address to cached one:
    if null or device has no address, uses current address
    if different, check override flag
  calls neigh_sync() to verify neighbor is still up
  updates neighbor contact time
  if old entry was valid and new one does not change address, returns 0
  if new address is different from old, replaces old with new
  if new and old states match, returns 0
  calls neigh_connect() or neigh_suspect() to make/check connection
  if old state was invalid:
    goes through packets in ARP queue, calling the neighbor output()
        function on each
    purges the ARP queue
  returns 0

rt_cache_get_info() - net/ipv4/route.c (191)
  prints header and all elements of rt_hash_table for proc FS

rt_hash_code() - net/ipv4/route.c (18)
  uses source address, destination address, and type of service to
      determine (and return) a hash value:
    hash = ((daddr&0xF0F0F0F0)>>4)|((daddr&0x0F0F0F0F)<<4);
    hash = hash^saddr^tos;
    hash = hash^(hash>>16);
    hash = (hash^(hash>>8)) & 0xFF;

rt_intern_hash() - net/ipv4/route.c (526)
  puts new route in routing table
```

Chapter 9 Dynamic Routing with *routed*

This chapter presents dynamic routing as performed by a router (as opposed to an end host computer). It provides an overview of how the *routed* program implements routing protocols under Linux, examines how it modifies the kernel routing tables, and summarizes the implementation code.

9.1 Overview

A normal host computer has very limited options for routing packets; a message is either for itself or another computer, and if it is for another computer there are a very limited number of options for sending it on. Usually such a host needs only to put a packet out on a LAN for a ``gateway'' computer (router) to pick up and send on its way. Linux usually does not maintain any metric (distance) information about routes, even though there are variables for storing it in the FIB. For simple end-host routing, the only important question is ``can I get there from here'', not ``which way is best?''

However, a router must make decisions on where to send traffic. There may be several routes to a destination, and the router must select one (based on distance, measured in hops or some other metric such as the nebulous quality of service). The Routing Information Protocol (RIP) is a simple protocol that allows routing computers to track the distance to various destinations and to share this information amongst themselves.

Using RIP, each node maintains a table that contains the distance from itself to other networks and the route along which it will send packets to that destination. Periodically the routers update each other; when shorter routes becomes apparent, the node updates its table. Updates are simply RIP messages with the destination address and metric components of this table. See Figure 9.1 for a diagram of an RIP routing table and an RIP packet.

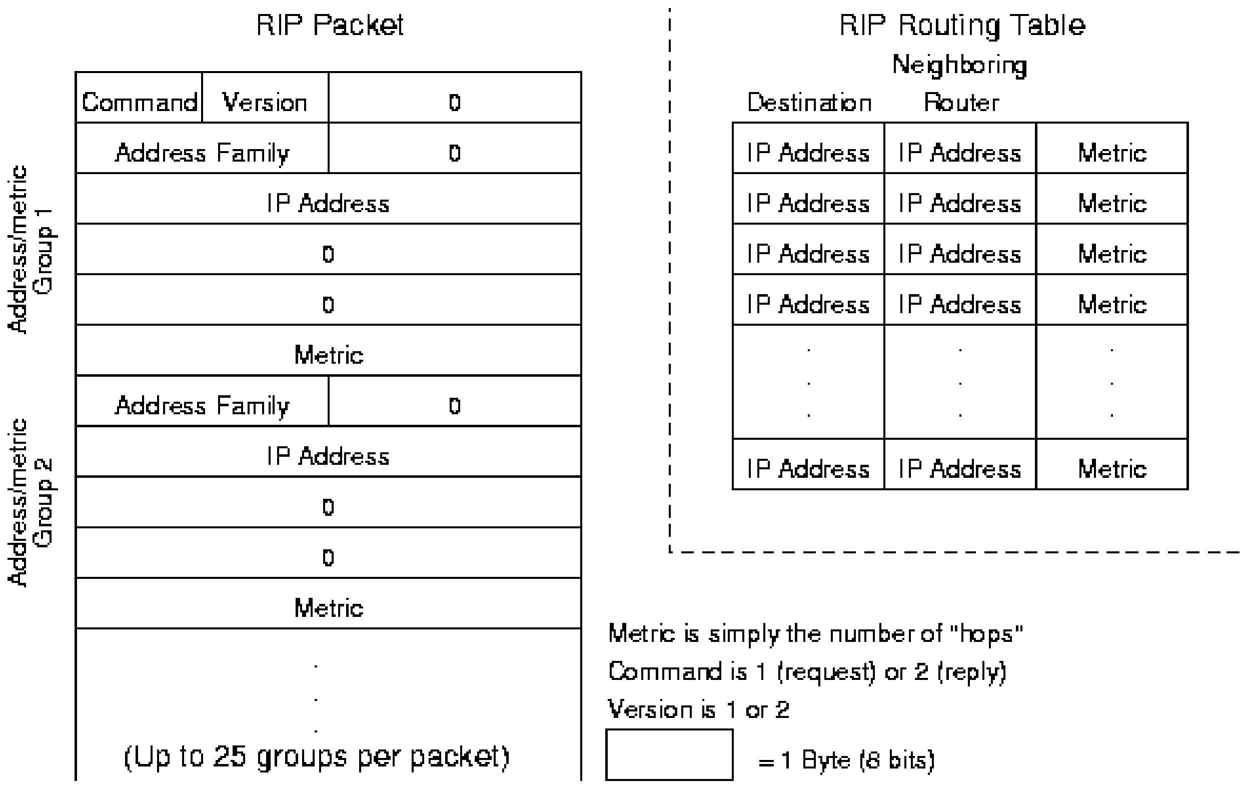


Figure 9.1: Routing Information Protocol packet and table.

9.2 How *routed* Works

routed is a widely available program for implementing RIP via UDP messages on POSIX computers. It is essentially a stand-alone program which uses `ioctl()` calls to get information from and update routing tables on the host machine.

9.2.1 Data Structures

routed maintains two identical data tables - one for hosts and one for networks. Each is a hash table with `ROUTEHASHSIZ` (32) buckets of chains of routing entries. The entries contain the RIP information (but can also line up with a `struct rtentry` so that *routed* can pass them to the kernel through `ioctl()` calls). Along with the basic destination, router, and metric information the entries store flags, state, and timer information.

9.2.2 Initialization

When *routed* begins, it performs various initialization actions and calls `ioctl()` to get interface information from the kernel. Next it sends out a RIP/UDP message requesting routing information from all neighboring routers. Finally it enters an infinite loop in which it waits for traffic or timers to make it do something.

9.2.3 Normal Operations

When RIP messages arrive (via a UDP socket), *routed* parses them and either modifies its table (for response messages) or sends information back to the requesting router (for requests). Sending information is simply a matter of looking up a destination in its own table, putting that information into a RIP packet, and sending it out through a UDP socket. Updating its table may have no impact (if there is no change or the change makes no difference) or it may result in a routing change. If the update reveals a shorter route to a destination, *routed* will update its own table and then call `ioctl()` to update the kernel's routing tables (the FIB).

When the update timer expires, every `TIMER_RATE` seconds, *routed* goes through every entry in both tables and updates their timers. Entries which are out of date are set to a distance of infinity (`HOPCNT_INFINITY`) and entries which are too old are deleted (only from the RIP table, not from the kernel's FIB). Finally, it sends an update to its neighboring routers. This update contains the new table information (response messages) for any entries which have changed since the last update.

routed leaves the actual routing to the normal kernel routing mechanisms; all it does is update the kernel's tables based on information from other routers and pass on its own routing information. The updates then change how the kernel routes packets, but *routed* itself does not actually do any routing.

9.3 *routed* Functions

The following is an alphabetic list of the *routed* program functions that are most important to routing, where they are in the source code, and what they do. The *SOURCES* directory shown represents the directory that contains the source code for the given network file.

The *routed* source is available as a package separate from the kernel source (Red Hat Linux uses the *rpm* package manager). The code below is from the *netkit-routed-0.10* source code package, 8 March 1997. This package is available from the *www.redhat.com/apps/download* web page; specifically this came from *www.redhat.com/swt/src/netkit-routed-0.10.src.html*. Once downloaded, *root* can install the package with the following commands (starting from the directory with the package):

```
rpm -i netkit-routed-0.10.src.rpm
cd /usr/src/redhat/SOURCES
tar xzf netkit-routed-0.10.tar.gz
```

This creates a */usr/src/redhat/SOURCES/netkit-routed-0.10* directory and fills it with the source code for the *routed* program. This process should be similar (but is undoubtably not exactly the same) for other Linux distributions.

```
ifinit() - SOURCES/routed/startup.c (88)
  opens a UDP socket
  calls ioctl(SIOCFIGCONF) to get interface configuration
  loops through interfaces:
    calls ioctl() to get flags, broadcast address, metric, and netmask
    creates a new interface structure
    copies info into interface structure
    calls addrouteforif() to add routing entry for interface
  sets supplier variable if necessary
  closes socket

process() - SOURCES/routed/main.c (298)
  starts a continuous loop:
    receives a packet (waits)
    verifies that packet is correct size
    calls rip_input() to handle (RIP) packet

rip_input() - SOURCES/routed/input.c (60)
  traces input if necessary
  checks packet to make sure protocol and address are supported
  checks for RIP version (cannot be 0)
  switch based on packet content -
    if packet is a request:
      checks request for validity
      if request is for all entries, calls supply()
      else looks up requested address, builds and sends response packet
    if packet is a trace on or off:
      verifies request came from a valid port
      if all is in order, sets trace to on or off
    if packet is a response:
      verifies response came from a router
      updates timer for interface
      loops through each entry in received packet:
        parses route information
        validates address family, host, and metric information
        updates hop count (adds metric in message to hop count to router
            that send message, subject to HOPCNT_INFINITY maximum)
        calls rtlookup() to find address in routing table
        if this seems to be a new route:
           calls rtfind() to look for an equivalent route
           if it really is new, calls rtadd() and returns
        calls rtchange() to modify route if necessary (new route or
           hopcount change)
        updates route timers
  if there were changes:
    sends an update if neccessary
    updates general update timer information

>>> routed main() - SOURCES/routed/main.c (78)
  opens routed log file
  calls getservbyname() to get UDP router
  sets up a UDP socket for RIP message traffic
  runs through command line arguments to set flags
  if not debugging, forks and runs program in new session (parent dies)
  calls rtinit() to initialize data tables
  calls ifinit() to fill in interface information
  calls toall() to request info from all other routers
  installs signal handlers for ALRM,HUP,TERM,INT,USR1,and USR2
  starts a continuous loop:
    if in need of update, sets up timer variables
    calls select() to wait for traffic
    if select() returns an error (other than EINTR), logs it
    if select() times out (time for update)
        calls toall() to broadcast update
        resets timer variables
    if there is traffic waiting on the socket, calls process()

rtadd() - SOURCES/routed/tables.c (138)
  verifies address family is in proper range
  calls family af_rtflags() function to set routing flags
  determines hash value for appropriate table (host or net)
  creates and fills in new rt_entry structure
  calls insque() to add entry to table
  calls rtioctl() to add entry to kernel table
  if call fails:
    if route should work, calls family af_format() to add destination
```

```
        and gateway to kernel tables
      if host is unreachable, removes and frees entry

  rtchange() - SOURCES/routed/tables.c (207)
    determines if change necessitates adding or deleting gateways
    calls rtioctl() to add and/or delete routes

  rtfind() - SOURCES/routed/tables.c (100)
    determines hash value for host table
    loops through table; returns entry if addresses are equal
    determines hash value for net table
    goes back to loop through table, this time returning entry if a call
        to family af_netmatch() function returns true
    returns null (0) if no match

  rtinit() - SOURCES/routed/tables.c (336)
    loops through the net hash table, setting forward and back pointers
    loops through the host hash table, setting forward and back pointers

  rtioctl() - SOURCES/routed/tables.c (346)
    fills in rtentry structure from parameters
    outputs trace actions if necessary
    calls ioctl(SIOCADDRT or SIOCDELRT) to update kernel table
    returns result of ioctl() call (or -1 for erroneous parameter)

  rtlookup() - SOURCES/routed/tables.c (65)
    determines hash value for address
    runs through host table looking for match
    if unsuccessful at first, tries again with net table
    returns pointer to entry or null (0)

  sndmsg() - SOURCES/routed/output.c (77)
    calls the appropriate family output function
    traces the packet if necessary

  supply() - SOURCES/routed/ouput.c (91)
    creates an RIP response message
    loops through the routing host table
      loops through the routing entries
        checks to see if routing host needs the entry
        if so, puts routing info into packet and sends it
    goes back and does it again with the routing net table

  timer() - SOURCES/routed/timer.c (56)
    updates timer variables
    loops through the host table
      updates timer information for each entry
      deletes entry if it is too old
      changes metric to infinity if it is getting old
    goes back and does it again with net table
    calls toall() if update is due

  toall() - SOURCES/routed/output.c (55)
    loops through interfaces:
      sets destination address to broadcast or specific address
      calls passed function [sndmsg() or supply()] with address
```

Chapter 10
Editing Linux Source Code

10.1 The Linux Source Tree

Linux source code is usually in the /usr/src directory (if installed). There may be many versions in different directory trees (such as *linux-2.2.5* or *linux-2.2.14*). There should be one soft link (*linux*) to the most current version of the code (i.e. *linux* → *linux-2.2.14*).

This is an overview of the Linux source directory structure (not all branches are shown:

/usr/src/linux/

- *arch* - architecture specific code, by processor
  - *i386* - code for Intel processors (including 486 and Pentium lines)
    - *boot* - location of newly compiled kernels
- *drivers* - code for drivers of all sorts
  - *block* - block device drivers (e.g., hard drives)
  - *cdrom* - CD ROM device drivers
  - *net* - network device drivers
  - *pci* - PCI bus drivers
- *fs* - code for different file systems (EXT2, MS-DOS, etc.)
- *include* - header files used throughout the code
  - *asm* → *asm-i386* - processor dependent headers
  - *config* - general configuration headers
  - *linux* - common headers
  - *net* - networking headers
- *kernel* - code for the kernel specific routines
- *lib* - code for errors, strings, and printf
- *mm* - code for memory management
- *modules* - object files and references for the kernel to load as required
- *net* - code for networking
  - *core* - protocol independent code
  - *ipv4* - code specific to IPv4
  - *packet* - protocol independent packet code
  - *sched* - code for scheduling network actions

10.2 Using EMACS Tags

The Linux source is obviously very large and spread throughout many files. A *TAGS* file allows you to quickly maneuver to a specific file in search of a reference.

10.2.1 Referencing with TAGS

Inside a file, move the cursor to a keyword you would like to look up (e.g., ``sock''). Press ``ESC'' ``.'' - EMACS will prompt for the tag to find (defaulting to the word your cursor is on); hit ``ENTER''. The first time you use it, you will have to specify which TAGS file to use (e.g., */usr/src/TAGS*). Next EMACS will automatically open the appropriate file (e.g., */usr/src/linux/include/linux/sock.h*) in a new buffer and put the cursor on the definition of that struct, #define, or function. If the definition it brings up is not the one you were looking for, press ``CTRL-U'' ``ESC'' ``.'' to bring up alternate references.

These tags work even as you make changes to the source files, though they will run slower as more and more changes are made. EMACS stores the tags in a file (defaulted to *TAGS*) with each reference, filename, and line number. If the tag is not at the stored line number, EMACS will search the file to find the new location.

10.2.2 Constructing TAGS files

If you need to start from scratch, follow the steps below.

The command to make a tags file is:

> `etags` *filename*

The command to append new information onto a tags file is:

> `etags -a` *filename*

These put the new tags into the file *TAGS* in the current directory. Filenames are stored as given, so absolute references will always refer to the same files while relative references depend on the position of the *TAGS* file. (Read the man page for *etags* for more information).

For example, to create a tags file for the *ipv4* source files, enter:

```
etags /usr/src/linux/net/ipv4/*.c
```

To add the header files, enter:

```
etags -a /usr/src/include/net/*.h
```

The *TAGS* file will now contain quick references to all the C source code and header information in those directories.

10.3 Using vi tags

The vi editor also supports the use of tags files (and creates them with the *gctags* command, which works almost exactly like the *etags* command shown above).

10.4 Rebuilding the Kernel

(See the Linux-kernel-HOWTO for more detailed instructions.)

This is a quick step-by-step guide to recompiling and installing a kernel from scratch.

1. Go to the top of the source directory (*/usr/src/linux*). If there is not already a historical copy of a working *.config* file (such as the current one), MAKE ONE. Until you have enough experience that you no longer need this guide, do not overwrite anything until you have made sure there is a copy to which you can revert. (On the other hand, once you have a stable kernel version, there is no reason to keep old ones around. Even a development system should probably only have an original working version, a last known stable version, and a current version.

2. Run `make xconfig` (`make config` and `make menuconfig` also work, but xconfig is by far the user-friendliest). Configure the system as desired; there is help available for most options. The config file

should default to the current settings, so you should only have to change the things you want to add or take out. As a general rule, select ``Y'' for essential or frequently used features (like the ext2 file system), ``M'' for things that are sometimes useful (like sound drivers), and ``N'' for things that do not apply (like amateur radio support). If in doubt, consult the help text or include something as a module.

3. Run `make dep` to make sure the options you heve selected will compile properly. This make take a few minutes as the computer checks all of the dependencies. If all goes well, the `make` program will simply exit; if there is a problem, it will display error messages and stop.

4. Run `make clean` to remove old object files IF you want to recompile everything. This obviously will make the compilation process take longer.

5. Run `make bzImage` to build the new kernel. (`make zImage` and `make boot` also build kernel images, but the bzImage will compile into the most compact file. If you are using one of these two methods for some reason, you may get a ``kernel too big'' error when you run *lilo* - try again with a `bzImage`.) This will take quite some time, depending on available memory.

6. Run `make modules` to build any modules (not included in the main kernel image).

7. Rename the old modules if necessary:

   ```
   mv /lib/modules/2.2.xx /lib/modules/2.2.xx-old
   ```

   (Note that you will not have to do this if you are compiling a completely new version; the old ones will still be in */lib/modules/2.2.xx* when you build version 2.2.*yy*.)

8. Run `make modules_install` to install the new modules. You must do this even if you built a monolithic kernel (one with no modules). (Note that there may be a Red Hat *module-info* text file or link in the boot directory; it is not terribly important and this does not update it.)

9. Copy the new kernel to the */boot* directory and change the kernel link (usually *vmlinuz*):

   ```
   cp arch/i386/boot/bzImage /boot/vmlinuz-2.2.xx
   ln -sf /boot/vmlinuz-2.2.xx /boot/vmlinuz
   ```

10. Copy the new *System.map* file to the */boot* directory and change the map link:

    ```
    cp System.map /boot/System.map-2.2.xx
    ln -sf /boot/System.map-2.2.xx /boot/System.map
    ```

11. Create a new *initrd* file if there are any SCSI devices on the computer:

    ```
    /sbin/mkinitrd /boot/initrd-2.2.xx.img 2.2.xx
    ```

12. Edit the file */etc/lilo.conf* to install the new kernel; copy the block for the old kernel (`image=vmlinuz`) and change the existing one to keep it as an option. For example, rename the image to `vmlinuz-2.2.xx-old` and change the label to `stable`. This way you can always reboot to the current (presumably stable) kernel if your changes cause problems.

13. Run `/sbin/lilo` to install the new kernel as a boot option.

14. Reboot the computer with the new kernel.

15. If the new kernel does not work properly, boot the old kernel and reconfigure the system before trying

again.

10.5 Patching the Kernel Source

Linux is a constantly changing operating system; updates can be released every few months. There are two ways to install a new kernel version: downloading the new source in its entirety or downloading patches and applying them.

Downloading the entire source may be preferable to guarantee everything works properly. To do so, download the latest kernel source and install (*untar*) it. Note that this will (probably) be a complete distribution, not a machine-specific one, and will contain a lot of extra code. Much of this can be deleted, but the configuration Makefiles rely on some for information. If space is an issue, delete the **.c* and **.h* files in the non-i386 *arch/* and *include/asm-** directories, but tread lightly.

Downloading patches may be quicker to do, but is somewhat harder. Because of distribution variations, changes you have made, or other modifications the patches may not quite work properly. You must apply patch files in order (to go from 2.2.12 to 2.2.14, first apply patch 2.2.13 then apply 2.2.14). Nevertheless, patches may be preferable because they work on an existing directory tree.

Once you have downloaded a patch (and unzipped it, if necessary), simply put it in the directory above *linux* (e.g., */usr/src/*) and run the patch program to install it:

```
patch -Np0 -verbose -r rejfile < patch-2.2.xx
```
(where xx is the patch version)

The `-N` option ignores patches that are already applied, and the `-p0` assumes the patch wants to apply itself to a source in a *linux* directory. The `-r rejfile` option puts all the patch rejects into one file (*rejfile*) - which may or may not be what you want to do. If you have not kept the entire source distribution, you will have to skip many changes (for different processor architectures) by simply hitting ``ENTER'' at the ``patch which file'' and ``ignore patch'' prompts. Once you are comfortable with the process, run it without the `-verbose` and `-r rejfile` options.

Once you have a new kernel version, follow the instructions on rebuilding the kernel to actually start using it. You probably will not have to change any of the configurations options, but you will almost definitely want to run `make clean` to remove any old object files.

Chapter 11
Linux Modules

This chapter presents the Linux module system. It provides an overview of how modules work, describes how to install and remove them, and presents an example program.

11.1 Overview

Linux kernels more recent than 2.0 can be (and usually) are modularized. There is a portion of the kernel that remains in memory constantly (the most frequently used processes, such as the scheduler) but other processes are only loaded when needed. An MS-DOS file system for reading disks, for example, might be loaded only on mounting such a disk and then unloaded when no longer needed. This keeps the space the kernel requires at any one time small while allowing it to do more and more. It is still possible to put everything into one ``monolithic'' kernel that will not need modules, but that is usually done only for special purpose machines (where all the required processes are known in advance).

Another advantage of modules is that the kernel can load and unload them dynamically (and automatically with the *kerneld* daemon). This means that a (super) user can load a module, test it, unload it, and debug it repeatedly without having to reboot the computer. This document assumes that the user has superuser access (you must be `root` to install and remove modules) and the kernel is configured for modules. (With a monolithic kernel, it is possible to set configuration options not to even allow modules.)

11.2 Writing, Installing, and Removing Modules

11.2.1 Writing Modules

Modules are just like any other programs except that they run in kernel space. As such, they must define `MODULE` and include *module.h* and any other kernel header files that define functions or variables they use. Modules can be quite simple (as the example shows) but they can also be quite complex, such as device drivers and entire file systems.

This is the general module format:

```
#define MODULE
#include <linux/module.h>
/* ... other required header files ...  */

/*
 *  ... module declarations and functions ...
 */

int init_module() {
  /* code kernel will call when installing module */
}

void cleanup_module() {
  /* code kernel will call when removing module */
}
```

Modules that use the kernel source must be compiled with *gcc* with the option `-I/usr/src/linux/include`; this ensures that the files included will be from the proper source tree.

Note that not all kernel variables are exported for modules to use, even if the code declares them to be `extern`. The */proc/ksyms* file or *ksyms* program display the exported symbols (not many of which are useful for networking). Recent Linux kernels export both the symbol and its version number using the `EXPORT_SYMBOL(x)` macro. For user created variables, use the `EXPORT_SYMBOL_NOVERS(x)` macro instead or the linker will not retain the variable in the kernel symbol table. Module writers may also want to use the `EXPORT_NO_SYMBOLS` macro; modules export all of their variables by default.

11.2.2 Installing and Removing Modules

Installing and removing modules is as simple as calling a program with the name of the compiled module. (You must be a superuser to install or remove a module.)

The *insmod* program installs a module; it first links the module with the kernel's exported symbol table to resolve references and then installs the code in kernel space.

```
/sbin/insmod module_name
```

The *rmmod* program removes an installed module and any references that it has exported.

```
/sbin/rmmod module_name
```

The *lsmod* program lists all the currently installed modules:

```
/sbin/lsmod
Module       Size  Used by
cdrom       13368   0 (autoclean) [ide-cd]
3c59x       19112   1 (autoclean)
```

11.3 Example

This is a complete example of a very simple module.

simple_module.c

```
/* simple_module.c
 *
 * This program provides an example of how to install a trivial module
 *   into the Linux kernel.  All the module does is put a message into
 *   the log file when it is installed and removed.
 *
 */

#define MODULE
#include <linux/module.h>
/* kernel.h contains the printk function */
#include <linux/kernel.h>

/************************************************************** init_module
 * the kernel calls this function when it loads the module */
int init_module() {
  printk("<1>The simple module installed itself properly.\n");
  return 0;
}  /* init_module */

/*********************************************************** cleanup_module
 * the kernel calls this function when it removes the module */
void cleanup_module() {
  printk("<1>The simple module is now uninstalled.\n");
}  /* cleanup_module */
```

This is the *Makefile*:

```
# Makefile for simple_module

CC = gcc -I/usr/src/linux/include/config

CFLAGS = -O2 -D__KERNEL__ -Wall

simple_module.o: simple_module.c

install:
        /sbin/insmod simple_module

remove:
        /sbin/rmmod simple_module
```

To use (must be `root`):

```
root# make
root# make install
root# make remove
root# tail /var/log/messages
... kernel: The simple module installed itself properly.
... kernel: The simple module is now uninstalled.
```

Chapter 12
The *proc* File System

This chapter presents the virtual *proc* file system. It provides an overview of how the file system works, shows how the existing network code uses the system, and details how to write and use *proc* entries from programs.

12.1 Overview

The *proc* file system is so named because it is found in the */proc* directory on most Linux machines. NOT including the *proc* FS is a configuration option, but the system is a powerful tool of which many programs make frequent use. While designed to appear as a file system with directory structures and inodes, it is in fact a construct of registered functions which provide information about important variables.

The *proc* directory has many subdirectories - one for each running process and others for subsystems such as file systems, interfaces, terminals, and networking (*/proc/net*). There are also many files in the main */proc* directory itself - *interrupts*, *ioports*, *loadavg*, and *version* to name a few. Within each process subdirectory (named for the process number) are files that describe the process' command line, current working directory, status, and so on.

The kernel traps *proc* file access and instead of executing ``normal'' file operations on them calls special (individually registered) functions instead. When a file in the */proc* directory is ``created'', it is registered with a set of functions that tell the kernel what to do when the file is read from or written to. Most entries only allow reads and they simply print out the state of certain system variables for use by other programs or for perusal by knowledgeable users.

The only tricky thing about using *proc* files is that the kernel calls the information generation function each and every time the file is read; subsequent reads of a changing file without copying and buffering the results may yield very different results. The best way to use a *proc* file is to read it into a `PAGE_SIZE`-byte buffer. This will read the entire entry at once and the buffer will then allow consistent random accesses.

12.2 Network *proc* Files

This is a list of the most important files in the */proc/net/* directory, what they contain, and a reference to the function and file that creates them. Note that there are many other interesting *proc* entries, such as the */proc/sys* files, */proc/ksyms*, and */proc/modules* to name only a few.

arp

displays the neighbor table (`arp_tbl`); the IP and hardware addresses, hardware type, device, and flags. (`arp_get_info()` : *net/ipv4/arp.c* 988)

dev
: displays reception and transmission statistics for each registered interface

dev_stat
: displays number of received packets dropped and throttle and FASTROUTE statistics (`dev_proc_stats()` : *net/core/dev.c* 1228)

netstat
: displays sync cookie, pruning, and ICMP statistics (`netstat_get_info()` : *net/ipv4/proc.c* 355)

raw
: displays address, queue, and timeout information for each open RAW socket from `struct proto raw_prot` (`get__netinfo()` : *net/ipv4/proc.c* 165)

route
: displays the FIB table (`main_table`); the interface, address, gateway, flags, and usage information. (`fib_get_procinfo()`) : *net/ipv4/fib_frontend.c* 109)

rt_cache
: displays the routing cache (`rt_hash_table`); the interface, address, gateway, usage, source, and other information. (`rt_cache_get_info()` : *net/ipv4/route.c* 191)

sockstat
: displays number of sockets that have been used and some statistics on how many were TCP, UDP, and RAW (`afinet_get_info()` : *net/ipv4/proc.c* 244)

tcp
: displays address, queue, and timeout information for each open TCP socket from `struct proto tcp_prot` (`get__netinfo()` : *net/ipv4/proc.c* 165)

udp
: displays address, queue, and timeout information for each open UDP socket from `struct proto udp_prot` (`get__netinfo()` : *net/ipv4/proc.c* 165)

12.3 Registering *proc* Files

This section describes the simplest method for registering a read-only *proc* ``file'' entry (available only in Linux 2.0 and later releases). It is possible to create a more fully functional entry by defining `file_operations` and `inode_operations` structures. However, that method is significantly more complicated than the one presented here; look in the source code for details on implementing fully functional entry. The method described below - defining a function and then registering and unregistering the function - provides most of the functionality required for testing and tracking system resources. Only the kernel can register a *proc* file; users can do so by building and installing kernel modules (though only `root` can install and remove modules). These procedures assume that the Linux source is installed and the kernel is compiled to use modules.

12.3.1 Formatting a Function to Provide Information

`static int` *read_proc_function* `(char *buf,char **start,off_t offset,int len,int unused)`

This is the function that the Linux kernel will call whenever it tries to read from the newly created *proc* ``file''. The only parameter that is usually significant is `buf` - a pointer to the buffer the kernel makes available for storing information. The others normally will not change. (*read_proc_function* is of course the name of the new function.)

Typically this function prints out a header, iterates through a list or table printing its contents (using the normal `sprintf` routine), and returns the length of the resulting string. The only limitation is that the buffer (`buf`) is at most `PAGE_SIZE` bytes (this is at least 4KB).

For an example of this kind of function, look at the `fib_get_procinfo()` function beginning on line 109 of *net/ipv4/fib_frontend.c*. This function displays the contents of the main FIB table.

12.3.2 Building a *proc* Entry

Because this is part of the file system, the entry needs an inode. This is easily constructed using a `struct proc_dir_entry`:

```
#include <linux/proc_fs.h>
struct proc_dir_entry new_proc_entry = {
   0,                       // low_ino - inode number (0 for dynamic)
   5,                       // namelen  - length of entry name
   "entry",                 // name
   S_IFREG | S_IRUGO,       // mode
   1,                       // nlinks
   0,                       // uid - owner
   0,                       // gid - group
   0,                       // size - not used
   NULL,                    // ops - inode operations (use default)
   &read_proc_function      // read_proc - address of read function
                            // leave rest blank!
}
```

The contents of this block can be used as shown by simply replacing the `namelen`, `name`, and `read_proc_function` fields with the desired values. Note that many of the kernel defined entries have predefined inode numbers (like `PROC_NET_ROUTE`, part of an enumeration defined in *include/linux/proc_fs.h*.

For an example of this kind of entry, look at the `__init_func()` function beginning on line 607 of *net/ipv4/fib_frontend.c*. This functions calls `proc_net_register()` (described below) with a newly created `proc_dir_entry` structure.

12.3.3 Registering a *proc* Entry

Once the read function and the inode entry are ready, all that remains is to register the new ``file'' with the *proc* system.

```
int proc_register(struct proc_dir_entry *dir, struct proc_dir_entry *entry)
int proc_net_register(struct proc_dir_entry *entry)
```

`dir` is a pointer to the directory in which the entry belongs - `&proc_root` and `proc_net` (defined in *include/proc_fs.h*) are probably the most useful. `entry` is a pointer to the entry itself, as created above. These two functions are identical except that `proc_net_register` automatically uses the */proc/net* directory. They return either 0 (success) or EAGAIN (if there are no available inodes).

12.3.4 Unregistering a *proc* Entry

When an entry is no longer needed, it should be deleted by unregistering it.

```
int proc_unregister(struct proc_dir_entry *dir,int inode)
int proc_net_unregister(int inode)
```

`dir` is the *proc* directory in which the file resides, and `inode` is the inode number of the file. (The inode is available in the entry's `struct proc_dir_entry.low_ino` field if it is not a constant.) Again, these functions are identical except that `proc_net_unregister` automatically uses the */proc/net* directory. They return either 0 (success) or EINVAL (if there is no such entry).

12.4 Example

This is a complete example of a module that installs a simple *proc* entry.

simple_entry.c

```
/* simple_entry.c
 *
 * This program provides an example of how to install an entry into the
 *   /proc File System.  All this entry does is display some statistical
 *   information about IP.
 */

#define MODULE
#include <linux/module.h>
/* proc_fs.h contains proc_dir_entry and register/unregister prototypes */
#include <linux/proc_fs.h>
/* ip.h contains the ip_statistics variable */
#include <net/ip.h>


/************************************************************ show_ip_stats
 * this function is what the /proc FS will call when anything tries to read
 *   from the file /proc/simple_entry - it puts some of the kernel global
 *   variable ip_statistics's contents into the return buffer */
int show_ip_stats(char *buf,char **start,off_t offset,int len,int unused) {
  len = sprintf(buf,"Some IP Statistics:\nIP Forwarding is ");
  if (ip_statistics.IpForwarding)
    len += sprintf(buf+len,"on\n");
  else
    len += sprintf(buf+len,"off\n");
  len += sprintf(buf+len,"Default TTL:  %lu\n",ip_statistics.IpDefaultTTL);
  len += sprintf(buf+len,"Frag Creates: %lu\n",ip_statistics.IpFragCreates);
  /* this could show more.... */
  return len;
}  /* show_ip_stats */

/**************************************************************** test_entry
 * this structure is a sort of registration form for the /proc FS; it tells
 *   the FS to allocate a dynamic inode, gives the "file" a name, and gives
 *   the address of a function to call when the file is read  */
struct proc_dir_entry test_entry = {
  0,                     /* low_ino - inode number (0 for dynamic)  */
  12,                    /* namelen - length of entry name          */
  "simple_entry",        /* name                                    */
  S_IFREG | S_IRUGO,     /* mode                                    */
  1,                     /* nlinks                                  */
  0,                     /* uid - owner                             */
  0,                     /* gid - group                             */
  0,                     /* size - not used                         */
  NULL,                  /* ops - inode operations (use default)    */
  &show_ip_stats         /* read_proc - address of read function    */
                         /* leave rest blank!                       */
};
```

```
/************************************************************ init_module
 * this function installs the module; it simply registers a directory entry
 *   with the /proc FS  */
int init_module() {
  /* register the function with the proc FS */
  int err = proc_register(&proc_root,&test_entry);
  /* put the registration results in the log */
  if (!err)
    printk("<1> simple_entry: registered with inode %d.\n",
          test_entry.low_ino);
  else
    printk("<1> simple_entry: registration error, code %d.\n",err);
  return err;
}  /* init_module */

/********************************************************* cleanup_module
 * this function removes the module; it simply unregisters the directory
 *   entry from the /proc FS  */
void cleanup_module() {
  /* unregister the function from the proc FS */
  int err = proc_unregister(&proc_root,test_entry.low_ino);
  /* put the unregistration results in the log */
  if (!err)
    printk("<1> simple_entry: unregistered inode %d.\n",
          test_entry.low_ino);
  else
    printk("<1> simple_entry: unregistration error, code %d.\n",err);
}  /* cleanup_module */
```

This is the *Makefile*:

```
# Makefile for simple_entry

CC = gcc -I/usr/src/linux/include

CFLAGS = -O2 -D__KERNEL__ -Wall

simple_entry.o: simple_entry.c

install:
        /sbin/insmod simple_entry

remove:
        /sbin/rmmod simple_entry
```

To use (must be `root`):

```
root# make
root# make install
root# cat /proc/simple_entry
Some IP Statistics:
IP Forwarding is on
Default TTL:  64
Frag Creates: 0
root# make remove
root# tail /var/log/messages
... kernel: simple_entry: registered with inode 4365.
... kernel: simple_entry: unregistered inode 4365.
```

Chapter 13
Example - Packet Dropper

This sample experiment inserts a routine into the kernel that selectively drops packets to a given host. It discusses the placement of the code, outlines the data from an actual trial, presents a lightweight analysis of the results, and includes the code itself.

13.1 Overview

This program is implemented as a module that, while installed, compares each outgoing packet's destination address to a given target. If they match, it randomly drops a percentage of those packets. It does this for all IP traffic, no matter where it was generated and what transport protocol it uses. Implementing this requires a modification to the kernel (to allow a module access to the transmission functions) and a module that takes advantage of that modification.

13.2 Considerations

Code Placement

This code could be built directly into the kernel or it could be designed as a module:

- Kernel - this is conceptually much simpler; simply adding some code to the kernel is a fairly easy matter. However, it makes semi-permanent changes and takes a long time to debug, since the entire kernel must be recompiled, installed, and rebooted for every change.
- Module - this is much safer and easier since the (super) user can install, remove, and debug modules quite painlessly. However, it requires access to the kernel that is not always available - even from a module. The kernel does not always export the variables that a module may need to access. (See the discussion on the *ksyms* program in Chapter 11.)
- Both - this is the best method; by performing a few minor modifications to the kernel code to export necessary variables and make use of a module only if it is loaded, a user can recompile the kernel once and then perform tests and experiments with modules. This still has the disadvantage of opening potential security holes on a system, but since only the experimenter knows how they are implemented, this is a minimal risk.

Protocol Level

This code could be implemented at many levels:

- Device Driver - this is a possibility since all traffic comes through the device. However, this breaks the layering protocols and requires hacking a (presumably) stable hardware driver.
- Generic Device Functions - this is the best choice, since this is the lowest level through which all traffic travels (specifically the `dev_queue_xmit()` and `netif_rx()` functions). It still violates the protocol layering, but all of the modifications can be made in one section of code.
- IP Protocol - this is conceptually the right place to insert a special function, either in the input, routing, or output routines. However, this is unsuitable precisely because there are three different routines in the implementation that a packet might go through - `ip_forward()` (forwarded packets), `ip_queue_xmit()` (TCP packets), or `ip_build_xmit()` (UDP packets). See the coding sections in Chapters 5 and 7 to see how these routines interact. These functions would be a good choice for inserting a special-purpose dropper, but not one that affects all traffic.

- Transport Protocol - these routines would be appropriate for affecting specific traffic types (such as UDP only) but are not useful for this example.

13.3 Experimental Systems and Benchmarks

This example was implemented on two computers that are connected through a single router as shown in Figure 13.1; the router runs the modifed kernel and packet dropper module. In the general example, this represents traffic flowing between `neon` and `eagle`, with `dodge/viper` dropping packets for `eagle`.

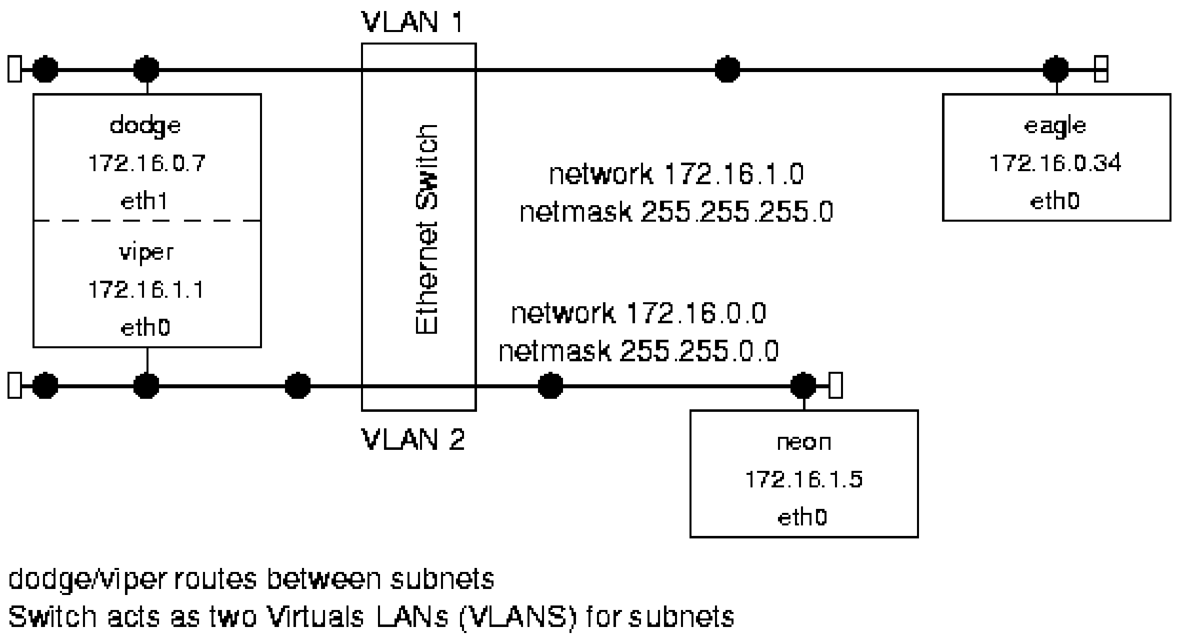


Figure 13.1: Experimental system setup.

The switch is a Cisco Catalyst 2900 set up with Virtual LANs (VLANs) for each ``subnetwork'' (one for the source computer and one for the destination computer, with the routing computer acting as the router between the two. The switch operates entirely on the link level and is essentially invisible for routing purposes.

The routing computer (`dodge/viper`) is a Dell Optiplex GX1 with a Pentium II/350 processor and 128M of RAM. It has three 3Com 3c59x Ethernet cards with 10Mbps connections to the switch.

One host computer (`neon`) is an AST Premmia GX with a Pentium/100 processor and 32M of RAM. It has an AMD Lance Ethernet card with a 10Mbps connection to the switch.

The other host computer (`eagle`) is a Dell Optiplex XL590 with a Pentium/90 processor and 32M of RAM. It has a 3Com 3c509 Ethernet card with a 10Mbps connection to the switch.

All computers have the Red Hat 6.1 distribution of Linux; the source and destination computers have standard recompiled version 2.2.14 kernels, while the router uses either a standard (2.2.14) kernel or a slightly modified one as indicated.

The first benchmark is a ``ping-pong'' test that establishes a TCP connection and then repeatedly sends packets back and forth. It returns a total transmission time (from start to finish, not including making and closing the connection); dividing the time by the number of iterations yields an average Round Trip Time (RTT). This test was run with 20,000 iterations of 5 byte packets and 5,000 iterations of 500 byte packets.

The second benchmark is a ``blast'' test that establishes a TCP connection and then sends data from a source to a destination. It returns a total transmission time (from start to finish, not including making and closing the connection); multiplying the number of packets by the size of the packets and dividing by the time yields the throughput. This test was run with 50,000 5 byte packets, 5,000 500 byte packets, and 1,000 1500 byte packets.

The benchmarks were run on both machines (i.e., from `neon` to `eagle` and from `eagle` to `neon`), but in both cases only packets to `eagle` were dropped. In each trial the blast test was run once with default settings (100 packets of 1 byte each) before running the performance tests ``for record'' to ensure that the routing cache and any protocol tables were in a normalized state. The complete suite was run ten times to capture variations between trials (the averages are presented here). None of the machines (including the router) were running any other user programs beyond a login shell and the appropriate module, client, or server programs (not even X Windows).

13.4 Results and Preliminary Analysis

13.4.1 Standard Kernel

These are the reference standards; these routines were run with the two computers directly connected (NOT routed) and while the router had an unmodified Linux 2.2.14 kernel. The error rate on such a direct connection is near zero.

ping-pong

```
                           Mean Time (sec)     Average RTT (millisec)
               Drop Rate   20K@5    5K@500         20K@5    5K@500
Direct -
 neon to eagle:   ---      17.24     28.98          0.86     5.80
 eagle to neon:   ---      17.20     28.99          0.86     5.80
Routed -
 neon to eagle:  (0.0%)    24.53     48.59          1.23     9.72
 eagle to neon:  (0.0%)    24.36     48.46          1.22     9.69
```

blast

```
                               Mean Time (sec)        Throughput (Mbits/sec)
               Drop Rate   50K*5  10K*500  1K*1500   50K*5  10K*500  1K*1500
Direct -
 neon to eagle:   ---       0.56    3.19     1.89     3.55    6.26    6.36
 eagle to neon:   ---       0.78    3.03     1.77     2.58    6.61    6.76
Routed -
 neon to eagle:  (0.0%)     0.56    3.19     1.92     3.60    6.27    6.26
 eagle to neon:  (0.0%)     0.77    3.19     1.93     2.60    6.27    6.23
```

13.4.2 Modified Kernel Dropping Packets

These are the experimental results. The drop rate of 0.0% provides a reference for measuring the overhead of calling the test and random functions without dropping any packets.

ping-pong

```
                           Mean Time (sec)     Average RTT (millisec)
               Drop Rate   20K@5    5K@500         20K@5    5K@500
neon to eagle:    0.0%     25.55     49.12          1.28     9.82
```

	0.1%	29.87	51.11	1.49	10.22
	0.5%	44.78	58.07	2.24	11.61
	1.0%	65.37	68.77	3.27	13.75
	5.0%	245.51	160.09	12.28	32.02
	10.0%	506.03	290.77	25.30	58.15
eagle to neon:	0.0%	25.53	49.21	1.28	9.84
	0.1%	29.08	50.92	1.45	10.18
	0.5%	45.87	59.21	2.29	11.84
	1.0%	66.19	68.66	3.31	13.73
	5.0%	235.68	156.94	11.78	31.39
	10.0%	519.61	297.02	25.98	59.40

blast

		Mean Time (sec)			Throughput (Mbits/sec)		
	Drop Rate	50K*5	10K*500	1K*1500	50K*5	10K*500	1K*1500
neon to eagle:	0.0%	0.55	3.19	1.91	3.64	6.26	6.27
	0.1%	0.55	3.07	1.93	3.62	6.51	6.21
	0.5%	0.55	2.95	1.76	3.64	6.77	6.82
	1.0%	0.55	2.87	1.75	3.65	6.96	6.87
	2.5%	0.59	3.36	2.04	3.38	5.59	5.90
	5.0%	0.63	4.63	2.71	3.19	4.31	4.43
	10.0%	1.06	7.08	5.11	1.89	2.83	2.35
	20.0%	3.43	30.35	18.55	0.58	0.66	0.65
eagle to neon:	0.0%	0.79	3.21	1.93	2.53	6.23	6.23
	0.1%	0.77	3.22	1.89	2.59	6.20	6.35
	0.5%	0.80	3.24	1.88	2.51	6.17	6.39
	1.0%	0.77	3.24	1.91	2.60	6.17	6.27
	2.5%	0.79	3.17	1.90	2.53	6.31	6.33
	5.0%	0.78	3.17	1.91	2.57	6.31	6.29
	10.0%	0.81	3.85	2.51	2.48	5.20	4.78
	20.0%	2.02	4.06	2.51	0.99	4.92	4.78

13.4.3 Preliminary Analysis

What follows is an elementary examination of the results. It is NOT intended as an exhaustive analysis, and indeed the experiment was not extensive enough to provide hard data from which to draw definite conclusions. However, this does demonstrate the multitude of factors involved and the effects that a few lines of code can have on a network. Further analysis, if desired, is left as an exercise for the reader.

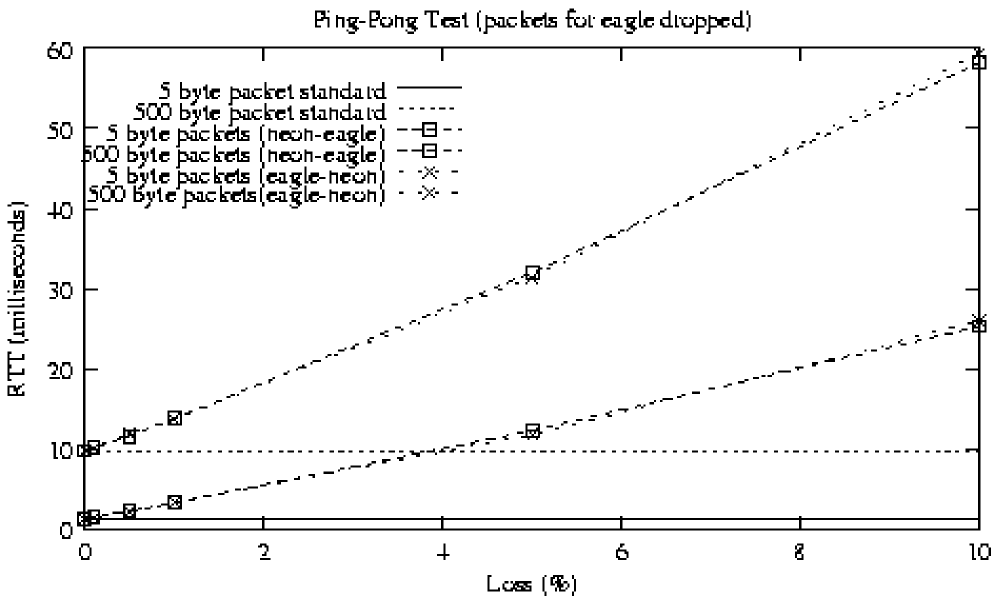


Figure 13.2: Ping-pong benchmark results.

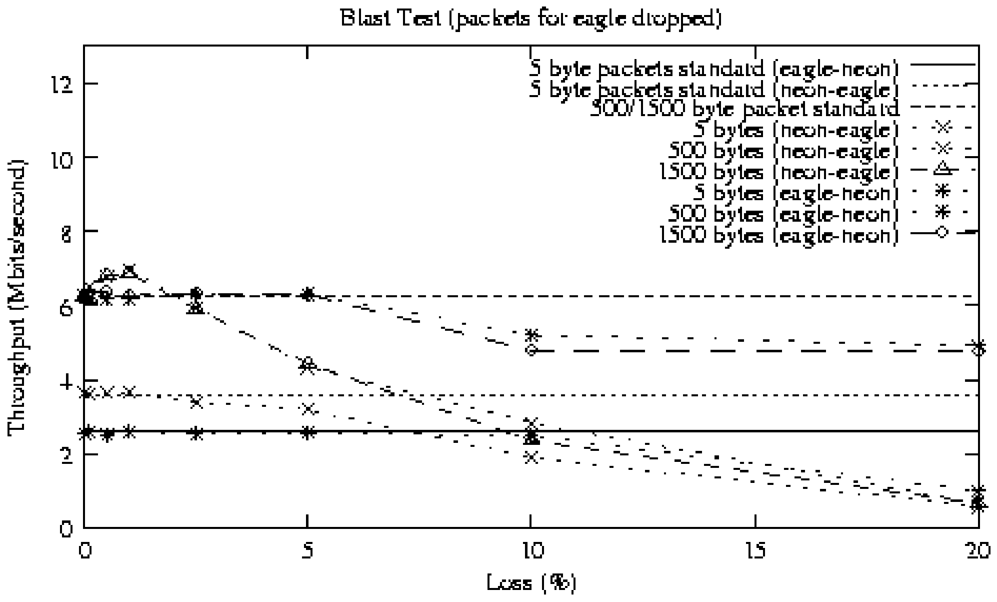


Figure 13.3: Blast benchmark results.

The kernel modifications and module insertion had a small but measurable impact on a TCP connection (measured by the increased RTT). For very small packets, this difference was approximately 0.05 msec; for large packets it was 0.10 msec. Why should there be a difference? Note that the direction of travel and packet size made a large difference on the throughput. This is an indication that processor speed and layering overhead are affecting the RTT; for a 1500 byte packet, 66 bytes of wrappers (20 for TCP, 20 for IP, and at least 26 for Ethernet) are not very significant - but for a 5 byte packet, that overhead is very large. Assume that the actual ``cost'' of inserting the module the delay for the larger packets, 0.10 msec.

Dropping packets from a TCP connection resulted in a fairly linear drop in performance on the ping-pong test; see the graph in Figure 13.2. This is as expected; when either a packet or acknowledgement is lost, the sender pauses and then sends again. The RTT is also very close (certainly within the expected experimental error) no matter which machine is the ``source''; again this is because the benchmark tests the behavior of both machines at the same time.

At low packet sizes, the throughput was very different depending on which way data was sent. This is because one machine (`eagle`) was slower than the other. For a large number of very small packets, the chokepoint in the network is not the medium or the interface, but the speed at which the processor can build and send packets. However, for larger packet sizes, the throughput (for low error rate) for both sources is similar; in this case the network is the limiting factor, not the processor.

The most surprising result is the apparent peak in throughput when the loss rate is approximately 1% - better even than no loss at all (for blasted data; loss of ACKs sent from the receiver to the source had little impact). This is a very counter-intuitive finding; why should losing packets speed up the throughput? A 1% error might be just enough to prevent a TCP exponential back-off algorithm from slowing the traffic rate. The immediate ACK that the receiver sends when an out-of-sequence packet arrives might include window size information that keeps the sender from pausing. Interrupts caused by out-of-sequence packets might result in the scheduler running the benchmark process more frequently, emptying the buffer window and again keeping the sender from pausing. There are many potential causes; determining the real one would take much more study - but would be very interesting.

13.5 Code

13.5.1 Kernel

The following code adds a trapdoor to the kernel. It creates a function that will be called (if it exists) from within the `dev_queue_xmit()` function and exports it so that modules will be able to use it. These lines are added directly to the source code; the kernel then has to be recompiled. installed, and booted. Note that the kernel still functions normally (albeit with one extra comparison) while no test module is installed.

net/core/dev.c (after line 579)

```
...
int *test_function(struct sk_buff *)=0;                  /* new */

int dev_queue_xmit(struct sk_buff *skb)...

    ...struct Qdisc  *q;

    if (test_function && (*test_function)(skb)) {         /* new */
        kfree_skb(skb);                                   /* new */
        return 0;                                         /* new */
    }                                                     /* new */

#ifdef CONFIG_NET_PROFILE...
```

net/netsyms.c (after line 544)

```
...
extern int (*test_function)(struct sk_buff *);            /* new */
EXPORT_SYMBOL_NOVERS(test_function);                      /* new */
EXPORT_SYMBOL(register_gifconf);...
```

13.5.2 Module

The following is the code for the packet dropping module itself. On installation, it calculates a percentage cut-off and puts an address into the function pointer defined above. From then on, any packets sent through `dev_queue_xmit()` will also pass through the `packet_dropper` function, which compares the destination address to a hard coded one. If they match and a random number comes up below the calculated cut-off, it drops the packet; otherwise the packets pass through untouched. When the module is removed, it resets the function pointer to 0 (`null`) again. (Note that this not very robust code depends on two byte short integers for simplicity. The function `get_random_bytes()` is only accessible to the kernel - or modules, of course - and provides random numbers that are ``merely cryptographically strong''.)

packet_dropper.c

```
/* packet_dropper.c
 *
 * This program provides an example of how to install a module into a
 *   slightly modified kernel that will randomly drop packets for a specific
 *   (hard-coded) host.
 *
 * See linux/drivers/char/random.c for details of get_random_bytes().
 *
 * Usage (must be root to use):
 *   /sbin/insmod packet_dropper
 *   /sbin/rmmod packet_dropper
```

```
  */

#define MODULE
#define MAX_UNSIGNED_SHORT 65535

#include <linux/module.h>
#include <linux/skbuff.h>  /* for struct sk_buff */
#include <linux/ip.h>      /* for struct iphdr */

extern int (*test_function)(struct sk_buff *);        /* calling function */
extern void get_random_bytes(void *buf, int nbytes); /* random function */
unsigned short cutoff;                               /* drop cutoff */
float rate   = 0.050;                                /* drop percentage */
__u32 target = 0x220010AC;                           /* 172.16.0.34 */

/************************************************************ packet_dropper
 * this is what dev_queue_xmit will call while this module is installed */
int packet_dropper(struct sk_buff *skb) {
  unsigned short t;
  if (skb->nh.iph->daddr == target) {
    get_random_bytes(&t,2);
    if (t <= cutoff) return 1;    /* drop this packet */
  }
  return 0;                       /* continue with normal routine */
}  /* packet_dropper */

/*************************************************************** init_module
 * this function replaces the null pointer with a real one */
int init_module() {
  EXPORT_NO_SYMBOLS;
  cutoff = rate * MAX_UNSIGNED_SHORT;
  test_function = packet_dropper;
  printk("<1> packet_dropper: now dropping packets\n");
  return 0;
}  /* init_module */

/************************************************************ cleanup_module
 * this function resets the function pointer back to null */
void cleanup_module() {
  test_function = 0;
  printk("<1> packet_dropper: uninstalled\n");
}  /* cleanup_module */
```

Chapter 14 Additional Resources

14.1 Internet Sites

Linux Documentation Project
http://metalab.unc.edu/mdw/index.html
Linux Headquarters
http://www.linuxhq.com
Linux HOWTOs
ftp://metalab.unc.edu/pub/Linux/docs/HOWTO
Linux Kernel Hackers' Guide

http://metalab.unc.edu/mdw/LDP/khg/HyperNews/get/khg.html

Linux Router Project
: http://www.linuxrouter.org

New TTCP
: http://users.leo.org/~bartel

Red Hat Software
: http://www.redhat.com

Requests for Comment
: http://www.rfc-editor.org/isi.html

14.2 Books

Computer Networks
Tanenbaum, Andrew, Prentice-Hall Inc., Upper Saddle River, NJ, 1996.

High Speed Networks
Stallings, William, Prentice-Hall Inc., Upper Saddle River, NJ, 1998.

Linux Core Kernel Commentary
Maxwell, Scott, CoriolisOpen Press, Scottsdale, AZ, 1999.

Linux Device Drivers
Rubini, Alessandro, O'Reilly & Associates, Inc., Sebastopol, CA, 1998.

Linux Kernel Internals
Beck, Michael, et al., Addison-Wesley, Harlow, England, 1997.

Running Linux
Welsh, Matt, Dalheimer, Matthias, and Kaufman, Lar, O'Reilly & Associates, Inc., Sebastopol, CA, 1999.

Unix Network Programming, Vol. 1 (2d Ed.)
Stevens, W. Richard, Prentice-Hall Inc., Upper Saddle River, NJ, 1998.

Chapter 15 Acronyms

ARP	Address Resolution Protocol
ATM	Asynchronous Transfer Mode (a protocol)
BSD	Berkeley Software Distribution
DHCP	Dynamic Hardware Configuration Protocol
DNS	Domain Name Server
FIB	Forwarding Information Base
GUI	Graphical User Interface
ICMP	Internet Control Message Protocol

INET
: Internet

IP
: Internet Protocol

ISP
: Internet Service Provider

LAN
: Local Area Network

LDP
: Linux Documentation Project

lo
: Loopback (device or interface)

MTU
: Maximum Transfer Unit

PPP
: Point-to-Point Protocol

RARP
: Reverse Address Resolution Protocol

RIP
: Routing Information Protocol

RTT
: Round Trip Time

TCP
: Transmission Control Protocol

UDP
: User Datagram Protocol

UNH
: University of New Hampshire

VLAN
: Virtual Local Area Network

WAN
: Wide Area Network

File translated from TEX by TTH, version 2.70.
On 31 May 2000, 23:35.