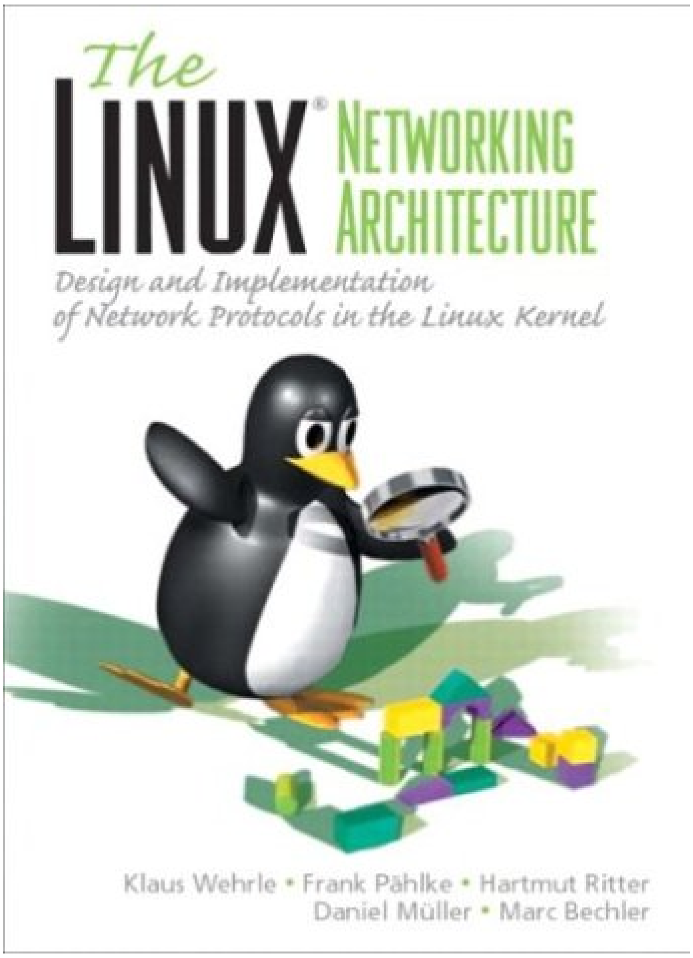
The Linux® Networking Architecture: Design and Implementation of Network Protocols in the Linux Kernel

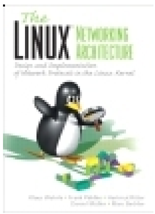
By Klaus Wehrle, Frank Pählke, Hartmut Ritter, Daniel Müller, Marc Bechler

Publisher: Prentice Hall
Pub Date: August 01, 2004
ISBN: 0-13-177720-3
Pages: 648
Supplier:Team FLY

Start Reading ▶


The
LINUX®
NETWORKING
ARCHITECTURE
Design and Implementation
of Network Protocols in the Linux Kernel
Klaus Wehrle • Frank Pählke • Hartmut Ritter
Daniel Müller • Marc Bechler

The most complete book on Linux networking by leading experts.



The Linux® Networking Architecture: Design and Implementation of

By Klaus Wehrle, Frank Pählke, Hartmut Ritter, Daniel Müller, Marc Bechler

Publisher: Prentice Hall
Pub Date: August 01, 2004
ISBN: 0-13-177720-3
Pages: 648

- Table of Contents
- Index

◂ Previous Next ▸

Copyright

An Alan R. Apt Book

Library of Congress Cataloging-in-Publication Data
CIP DATA AVAILABLE.

Vice President and Editorial Director, ECS: Marcia J. Horton

Publisher: Alan Apt

Associate Editor: Toni Dianne Holm

Editorial Assistant: Patrick Lindner

Vice President and Director of Production and Manufacturing, ESM: David W. Riccardi

Executive Managing Editor: Vince O'Brien

Managing Editor: Camille Trentacoste

Production Editor: Irwin Zucker

Director of Creative Services: Paul Belfanti

Creative Director: Carole Anson

Art Director and Cover Manager: Jayne Conte

Managing Editor, AV Management and Production: Patricia Burns

Art Editor: Gregory Dulles

Manufacturing Manager: Trudy Pisciotti

Manufacturing Buyer: Lisa McDowell

Marketing Manager: Pamela Hersperger

Translator: Angelika Shafir

Pearson Prentice Hall
Pearson Education, Inc.
Upper Saddle River, NJ 07458

Authorized translation from the German language edition entitled Linux Netzwerkarchitektur: Design und Implementierung von Netzwerkprotokollen im Linux-Kern published by Addison-Wesley, an imprint of Pearson Education Deutschland GmbH, München, ©2002.

The author and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The author and publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The author and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

Previous Next

Preface

This book deals with the architecture of the network subsystem in the Linux kernel. The idea for this book was born at the Institute of Telematics at the University of Karlsruhe, Germany, where the Linux kernel has been used in many research projects and its network functionality is modified or enhanced, respectively, in a targeted way. For instance, new services and protocols were developed for the next-generation Internet, and their behavior was studied. In addition, existing protocols, such as the TCP transport protocol, were modified to improve their behavior and adapt them to the new situation in the Internet.

In the course of these research projects, it has been found that the Linux kernel is very suitable for studying new network functionalities, because it features a stable and extensive implementation of the TCP/IP protocol family. The freely available source code allows us to modify and enhance the functionality of protocol instances easily. In addition, the enhancement of the kernel functionality is very elegantly supported by the principle of the kernel modules. However, many studies and theses in this field showed that familiarization with the Linux network architecture, which is required before you can modify the behavior of a protocol instance, demands considerable work and time. Unfortunately, this is mainly due to the facts that the network subsystem of the Linux kernel is poorly documented and that there is no material that would explain and summarize the basic concepts.

Although there are a few books that deal with the Linux kernel architecture and introduce its basic concepts, none of these books includes a full discussion of the network implementation. This situation may be due to the following two reasons:

- The network subsystem in the Linux kernel is very complex. As mentioned above, it implements a large number of protocols, which is probably one good reason for the enormous success of Linux. Both [BoCe00] and [BBDK+01] mention that the description of all these protocols and their concepts would actually fill an entire book. Well, you are reading such a book now, and, as you can see, it has eventually turned out to be quite a large volume, although it describes only part of the network functionality, in addition to the basic concepts of the Linux network architecture.

- Operating-system developers normally deal with the classical topics of system architecture? for example, the management of memories, processes, and devices, or the synchronization of parallel activities in a system? rather than with the handling of network packets. As you go along in this book, you will surely notice that it has been written not by system developers, but by computer-science specialists and communication engineers.

While considering the facts that there was little documentation covering the Linux network architecture and that students had to familiarize themselves with it over and over again, we had the idea of creating a simple documentation of the Linux network architecture ourselves. Another wish that eventually led to the more extensive concept of this book was a stronger discussion of important communication issues: design and implementation of network protocols in real-world systems. Networking courses teach students the most important concepts and standards in the field of telecommunication, but the design and implementation of network functionality (mainly of network protocols) by use of computer-science concepts has enjoyed little attention in teaching efforts, despite the fact that this knowledge could have been used often within the scope of studies and theses. The authors consider the description of the implementation of the Linux network architecture and its structure, interfaces, and applied concepts a step towards strengthening the informatics component in networking classes.

The authors hope that this book will help to make the processes and structures of the Linux network architecture easier to understand, and, above all, that our readers will have fun dealing with it and perhaps learn a few things about the networking concept and its practical implementation.

The content of this book corresponds to our knowledge of the Linux network architecture. This knowledge is neither comprehensive nor exhaustive. Nevertheless, we have tried to represent the processes and structures of the Linux network architecture in a fashion as easily understandable and detailed as possible. We are thankful for all hints, suggestions for improvement, ideas, and comments, and we will try to consider them in later editions. Updated information about the Linux network architecture and this book is available online at http://www.Linux-netzwerkarchitektur.de.

Organization of this Book

Chapter 1 will deal intensively with the motivation behind Linux in general and the Linux network architecture in particular; Chapter 2 is an introduction into the basic mechanisms and components of the Linux kernel. To keep the volume of this book manageable, we will discuss only those components that are important for understanding the Linux network architecture. With regard to the other components of the Linux kernel, we refer our readers to other books (e.g., [BBDK+01]).

Chapter 3 is an introduction to the general architecture of communication systems and the functionality of protocols and protocol instances. It includes an introduction to the popular TCP/IP and ISO/OSI layering models.

Chapters 4 and 5 discuss fundamental concepts of the Linux network architecture, including the representation and management of network packets in the Linux kernel (see Socket Buffers?A class="docLink" HREF="0131777203_ch04.html#ch04">Chapter 4) and the concept of network devices (Chapter 5). Network devices form the links between the protocol instances on the higher layers and hide the particularities of the respective network adapters behind a uniform interface.

Chapter 6 gives an overview of the activity forms in the Linux network architecture and the flow of transmit and receive processes. In addition, this chapter introduces the interface to the higher-layer protocol instances.

Chapters 7 through 12 discuss protocols and mechanisms of the data link layer. More specifically, it describes the SLIP, PPP, and PPP-over-Ethernet protocols and how the ATM and Bluetooth network technologies are supported in Linux. Finally, we will describe how a Linux computer can be used as a transparent bridge.

Our discussion of the TCP/IP protocols starts with an overview of the TCP/IP protocol family in Chapter 13. We will begin with a brief history of the Internet, then give an overview of the different protocols within the TCP/IP protocol family. Chapter 14 will deal with the Internet Protocol and its mechanisms in detail. In addition, it introduces the IP options and the ICMP protocol. Chapters 15 through 23 discuss the following protocols and mechanisms on the network layer: ARP, routing, multicasting, traffic control, firewalls, connection tracking, NAT, KIDS, and IPv6.

Chapters 24 and 25 describe the TCP and UDP transport protocols, respectively. We will close our discussion of the kernel with an explanation of the socket interface, in Chapter 26, then end with a short overview of the programming of network functionality on the application level.

The appendix includes additional information and introduces tools facilitating your work with the Linux network architecture. The issues dealt with include the LXR source code browser, debugging work in the Linux kernel, and tools you can use to manage and monitor the Linux network architecture.

◀ Previous Next ▶

Additional Sources of Information

This section lists a few useful sources of information where you can find additional information about the Linux network architecture.

Magazines

- The Linux Magazine (http://www.Linux-mag.com) is probably the best-known Linux magazine. It features articles about all issues that are of interest when you deal with Linux. Of special interest is the Kernel Corner column, which regularly publishes articles about the architecture and implementation of components of the Linux kernel? most of them by developers themselves.

- Linux Focus (http://www.linuxfocus.org) is an online magazine publishing articles in many different languages. It also includes a Kernel Corner.

- The Linux Gazette (http://www.linuxgazette.com) is another online magazine dedicated to Linux.

Useful Links in the World Wide Web

- Linux Headquarters: http://www.linuxhq.com

- Linux Documentation Project: http://www.linuxdoc.org

- Linux Weekly News: http://www.lwn.net

Other Information

- Howtos include a lot of information about different Linux issues. Most deal with the configuration and installation of various Linux functionalities. Especially for the Linux kernel, there are also a few howto documents? for example, how to use locks in the kernel [Russ00b], and general information on hacking in the Linux kernel [Russ00c]. Of course, we should not forget to mention the networking howto, which includes a wealth of tips and information about configuring the network functionality in Linux [Drak00].

- The source code of the current kernels is found at `ftp.kernel.org`. There are also mirrors of this FTP server, a list of which can be found at http://www.kernel.org/mirrors/.

- Information about components and drivers of the Linux kernel are also included directly in the source code of a kernel version, in the Documentation subdirectory. In addition, the file Documentation/kernel-docs.txt includes a list of current information about the Linux kernel? for example, documentation, links, and books. (It's worth taking a look at this file!)

◀ Previous Next ▶

Previous Next

Conventions Used in this Book

This book uses the following typographical conventions to emphasize various elements of the Linux kernel, source texts, and other things.

Functions

A gray bar denotes important functions. A bar describes the function name on the left and the file name (within the kernel's source-code tree) on the right.

When giving a function name in such a place and throughout the body of this book, we normally leave out the parameters, because they would take up much space and impair the readability and text flow.

In general, when introducing a function, we describe the entire parameter set and give a brief description. The variable type is normally left out. For example, the description of the function `int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt)` from the file `net/ipv4/ip_input.c` is denoted as follows:

`ip_rcv()` net/ipv4/ip_input.c

Throughout the body of this book, we would then refer to this function as `ip_rcv()` or `ip_rcv(skb, dev, pt)`.

Variables, Function Names, Source Text Excerpts, and so on

A `sans-serif` font is used for excerpts from the source code, variable and function names, and other keywords referred to in the text.

Commands, Program Names, and so on

A `sans-serif` font is used for the names of programs and command-line tools. Parameters that should be passed unchanged are also printed in `sans-serif`; those parameters that have to be replaced by values are printed in `sans-serif italic`.

Direct input in the command line is often denoted by a leading shell prompt? for example,

Files, Directories, Web Links, and so on

A `sans-serif` font is used for files and directories. We generally give the relative path in the kernel source code for files of the Linux kernel (e.g., `net/ivp4/ip_input.c`). Web links are also printed in `sans-serif` font (e.g., http://www.Linux-netzwerkarchitektur.de).

Other Conventions

Italic text denotes emphasis, or an introduction to a key term or concept.

Previous Next

◂ Previous Next ▸

Acknowledgments

Many people's contributions were indispensable in the creation and production of this book. First and foremost, we would like to thank all students who studied the structure of the Linux network architecture in their papers and theses. They contributed enormously to collecting knowledge about the Linux network architecture at the Institute of Telematics:

Nasieh Abdel-Haq, Paul Burczek, Michael Conrad, Frank Dinies, Paul Hankes Drielsma, Jérôme Freilinger, Carolin Gärtner, Stefan Götz, Karsten Hahn, Artur Hecker, Tobias Hinkel, Michael Hofele, Verena Kahmann, Vera Kießling, Stefan Klett, Andreas Kramm, Jan Kratt, Eckehardt Luhm, David Metzler, Ulrich Mohr, Rainer Müller, Sven Oberländer, Vincent Oberle, Jan Oetting, Torsten Pastoors, Christian Pick, Christian Schneider, Steffen Schober, Marcus Schöller, Achim Settelmeier, Uwe Walter, and Jürgen Walzenbach.

The authors wrote this book mainly for their students.

Much appreciation is due to Professor Gerhard Krüger, who has always supported our activities, given us the freedom necessary to write this book, and assisted us with valuable advice. His support also allowed us to procure a Linux test network, which served as the basis for our research activities at the Institute of Telematics in the field of services for the next-generation Internet.

Our special thanks go to all the folks at the publishing houses who published the original German version of this book and the English translation that you are currently reading. Particularly, we would like to thank our editors, Sylvia Hasselbach and Toni Holm. Their admirable patience helped shepherd us through this book-writing process. The English translation was done by Angelika Shafir, whom we would also like to thank in this place. We also thank all the people who read the manuscript, especially Mark Doll, Sebastian Döweling, Thomas Geulig, Thorsten Sandfuchs, Marcus Schöller, Bernhard Thurm, Uwe Walter, Anja Wehrle, Kilian Weniger, and Friederike Daenecke.

Last but not least, we gratefully acknowledge the support, encouragement, and patience of our families and friends.

KARLSRUHE · BERKELEY · BERLIN · BRAUNSCHWEIG

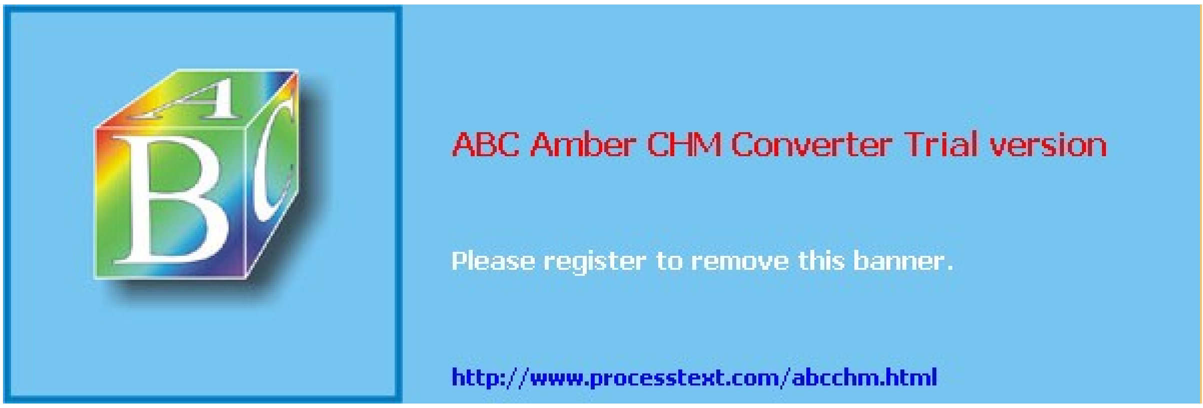
KLAUS WEHRLE · FRANK PÄHLKE · HARTMUT RITTER · DANIEL MÜLLER · MARC BECHLER

◂ Previous Next ▸

◂ Previous Next ▸

Part I: The Linux Kernel

◂ Previous Next ▸



◀ Previous Next ▶

Chapter 1. Motivation

Digital data transmission and processing form the basis of our today's information society. Within a short time, the Internet has penetrated all areas of our daily lives, and most of us can surely not imagine everyday life without it. With its new services, it offers us ways to communicate, fascinating all social strata, but corporations and organizations also use the possibilities of the Internet as a basis for internal exchange of information and for communication and handling business with customers and partners.

The technique of the Internet has been developed during the past twenty years; the actual boom began with the introduction of the World Wide Web at the beginning of the nineties. Development has progressed since then; new protocols and standards have been integrated, improving now both the functionality and the security in the "global net."

As developments in the Internet progressed, so did the technologies of the underlying network: The first e-mails were sent over telephone lines at 1200 bits/s in the eighties, but we can now communicate over gigabit or terabit lines. In addition, new technologies for mobile communication are emerging, such as UMTS and Bluetooth.

All these technologies have one thing in common: They are integral parts of digital communication systems, allowing spatial communication and interaction of distributed applications and their users. Modern communication systems decompose these extremely complex tasks into several layers, and the instances of these layers interact via predefined protocols to supply the desired service.

Telematics[1] is a field that handles both the development and research of telecommunication systems (and their basic mechanisms) and the implementation and realization of these systems by using means of computer science. This means that, in addition to the design of communication systems and protocols, the implementation of these mechanisms is an important task within the telematics discipline. Unfortunately, many universities and academic institutions neglect this point. For example, during coverage of the basics and the current standards with regard to communication protocols in detail, only very little knowledge is conveyed as to how these principles can be used (e.g., which basic principles of computer science can be used when implementing communication protocols).

With this book, the authors? who themselves teach computer-science students? attempt to contribute to promoting the computer-science component in telematics. Using the Linux operating system as an example, which the authors employ mainly for research purposes, in addition to the usual office applications (e-mail, World Wide Web, word processing, etc.), we will introduce the practical realization of communication systems and communication protocols. Essentially, the structuring of the network subsystem in the Linux kernel, the structuring of interfaces between network components, and the applied software methods will be used to show the reader various ways to implement protocols and network functionality.

In addition to its teaching use, of course, this book is also intended to address all those interested in the architecture of the network subsystem in the Linux kernel, taking a look behind the scenes at this poorly documented part of the Linux kernel. The following section discusses the Linux operating system and the reasons for its use in offices, companies, networks, and research.

[1] Telematics is the subdiscipline of informatics that deals with the design and implementation of telecommunication systems by use of information technologies.

◀ Previous Next ▶

◄ Previous Next ►

1.1 The Linux Operating System

Linux is a freely available multiuser, multitasking, multiprocessor, and multiplatform UNIX operating system. Its popularity and the number of users increase continually, making Linux an increasingly serious factor in the operating-systems market. Thanks to the freely available source code that everybody can obtain over the Internet and to the fact that everybody can participate in and contribute to the further development of the Linux system, many developers, all over the world, are constantly busy further developing this system, removing existing errors, and optimizing the system's performance.

The fact that most developers do this very time-consuming work for free in their spare time is a sign of the great fun working with Linux and mainly with the Linux kernel can be. As we progress in this book, we will try to pass some of this enthusiasm on to our readers. The large number of research projects at the University of Karlsruhe that have used, enhanced, or modified the Linux network architecture experienced a high motivation of all participating students. The reason was mainly that this offered them a way to participate in the "Linux movement."

The development of Linux was initiated by a student by the name of Linus B. Torvalds, in 1991. At that time, he worked five months on his idea of a new PC-based UNIX-like operating system, which he eventually made available for free on the Internet. It was intended to offer more functions than the Minix system designed by Andrew S. Tanenbaum, which was developed for teaching purposes only [Tane95]. With his message in the Minix newsgroup (see page 1), he set a movement in motion, the current result of which is one of the most stable and widely developed UNIX operating systems. Back then, Linus Torvalds planned only the development of a purely experimental system, but his idea further developed during the following years, so that Linux is now used successfully by many private people, corporations, and scientists alike. Mainly, the interoperability with other systems (Apple, MS-Windows) and the ability to run on many different platforms (Intel x86, MIPS, PA-RISC, IA64, Alpha, ARM, Sparc, PowerPC, M68, S390) make Linux one of the most popular operating systems.

Not only the extensive functionality of Linux, but also the freely accessible source code of this operating system, have convinced many private people and companies to use Linux. In addition, the German government, with its program for the support of open-source software, promotes the use of freely available programs with freely available source code. The main reason for this is seen not in the low procurement cost, but in the transparency of the software used. In fact, anyone can view the source code and investigate its functionality. Above all, anyone can check what? perhaps security-relevant? functionalities or errors are contained in an application or operating system. Especially with commercial systems and applications, there are often speculations that they could convey information about the user or the installed applications to the manufacturer.

You do not have such fears with freely developed software, where such a behavior would be noticed and published quickly. Normally, several developers work concurrently on an open-source project in a distributed way over the Internet, monitoring themselves implicitly. After all, free software is not aimed at maximizing the profit of a company or its shareholders. Its goal is to develop high-quality software for everybody. Linux is a very good example showing that freely developed software is not just the hobby of a handful of freaks, but leads to serious and extremely stable applications.

The authors of this book use Linux mainly for research work in the network area. The freely available source texts allow us to implement and evaluate new theories and protocols in real-world networks. For example, Linux was used to study various modifications of the TCP transport protocol [WeRW01, Ritt01], to develop a framework for the KIDS QoS support [Wehr01b], and to develop the high-resolution UKA-APIC timer [WeRi00].

◄ Previous Next ►

1.2 What is Linux?

Originally, the term Linux described only the operating-system kernel that abstracts from the hardware of a system, offering applications a uniform interface. Over time, the term Linux has often come to mean the kernel (the actual Linux) together with the entire system environment, including the following components:

- the operating-system kernel (currently version 2.0, 2.2, or 2.4);
- the system programs (compiler, libraries, tools, etc.);
- the graphical user interface (e.g., XFree) and a window manager or an application environment (KDE, Gnome, FVWM, etc.);
- a large number of applications from all areas (editors, browsers, office applications, games, etc.).

Different components not forming part of the kernel originate largely from the GNU project of Free Software Foundation, which explains why the complete system environment is often called "GNU/Linux system." A characteristic common to the Linux kernel and GNU programs is that they may all be freely distributed under the GNU Public License (GPL), provided that the source text is made publicly available. To the extent that enhancements or modifications have been effected to the programs, then these are automatically governed by the GNU license (i.e., their source text must also be made freely available). Since the advent of Linux, this has had the effect that the system has been further developed free from corporate policy interests and that it has been more strongly oriented to word its users' needs than are other, commercial operating systems. Anyone can participate in the development and implement new capabilities, ones based on the freely available source texts. This means that Linux is always involved in the support of international standards, and no attempt is made to enforce corporate or proprietary standards to secure a market position.

Errors made during the development of a piece of software are normally removed quickly. In addition, there is a continual effort to keep the system performing as well as possible. This has become very clear in the example of the network implementation in the last kernel version: After it had become known that the performance of Linux in the area of protocol handling on multiprocessor systems suffers from a few flaws, the network part was extensively rewritten to remove these faults. This means that Linux is an example that clearly shows the benefits of open-source projects:

- stability,
- performance, and
- security.

◂ Previous Next ▸

1.3 Reasons for Using Linux

The previous section introduced the important properties and objectives of Linux as a free software project. This section will discuss a number of general properties of the Linux kernel, offering more reasons for its use:

- Linux supports preemptive multitasking: All processes run independently in different protected memory spaces, so that the failure of one process does not in any way impair the other processes. When a process claims too much computing time, its processor can be taken and allocated to another waiting application. Preemptive multitasking is a fundamental requirement for stable systems.

- Multiprocessor: Linux is one of the few operating systems supporting several processors in SMP (Symmetric MultiProcessing) operation. This means that several processes can be handled concurrently by different CPUs. Since kernel version 2.0, multiprocessor systems with Intel and Sparc processors are supported. Version 2.2 and the current Version 2.4 additionally improved the performance and parallelism in the Linux kernel.

- Multiuser: Several users can work concurrently in one system, when they are logged in over different consoles. In addition, users can work easily on several graphical user interfaces.

- Multiplatform: properties of: Linux was originally developed only for the personal computer (Intel 80386), but it runs on more than ten processor architectures today. The bandwidth of supported platforms extends from small digital personal assistants over the standard personal computer to mainframe architectures: Intel x86, MIPS, PA-RISC, IA64, Alpha, ARM, Sparc, PowerPC, M68, and so on.

- Linux is a UNIX system: It is compatible with the POSIX-1300.1 standard[2] and includes large parts of the functionality of UNIX System V and BSD. This means that you can use UNIX standard software under Linux.

  [2] Portable Operating System Interface based on UniX? POSIX 1300.1 defines a minimum interface that each UNIX-like operating system must offer.

- Rich network functionality: The Linux network architecture makes available an extensive choice of network protocols and functionalities in the networking area. The development of the Internet and its services is inseparably linked to UNIX systems. This is why the properties of the TCP/IP protocol family and its behavior can best be studied and controlled in a UNIX system. Other PC operating systems would be unsuitable for this, especially those with source code not publicly available.

- Open source: The source code of the entire Linux kernel is freely available and can be used according to the GNU Public License. A large number of programmers work on the further development of the Linux kernel all over the world, continually enhancing and improving it. Linux is distributed over the Internet so that each user can test the kernel and make improvements or enhancements. The development of Linux in this dimension would not have been possible without the Internet.

  Formerly, users had to put up with defects in software they purchased; Linux now allows everyone to remove such defects. And it really works. An often heard criticism has been that the driver support for Linux is one of its major problems. This situation has changed dramatically during the past years. For instance, all actually available network cards are supported by Linux. In fact, we can rely to the Linux community to such an extent that there will soon be a matching driver for each new device.

- Efficient network implementation: Meanwhile, the Linux kernel makes available a well-structured implementation of the network functionality, which will be our main focus of discussion in the next 27 chapters of this book. The functions can be adapted to the special requirements of the desired system and meet the specifications of the Internet Engineering Task Force (IETF), IEE, and ISO better than many other systems.

  In the creation of a new kernel, its desired functionality can be individually configured. For instance, you can enable a large number of optimization options or add specific functionalities

Chapter 2. The Kernel Structure

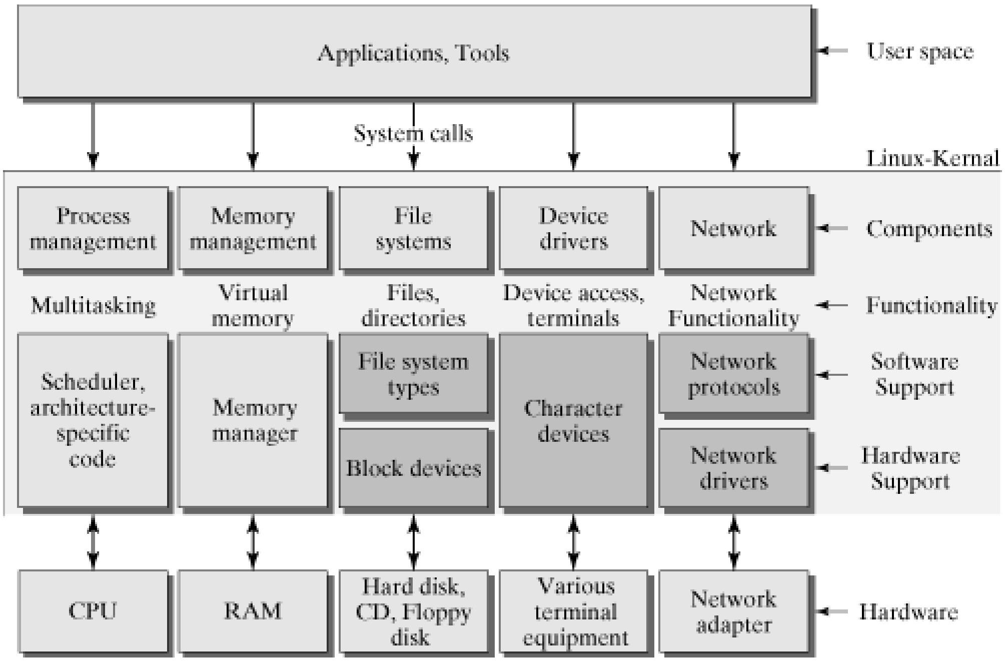
This chapter deals with the basic architecture of the Linux kernel and its components. It provides an overview of the most important areas of the kernel, such as the different forms of activity in the kernel, memory management, device drivers, timers, and modules. Each of these issues will be discussed briefly in this book, to give you an insight into the tasks and processes of each component. Detailed information about each of these issues is found in other books and references. A choice of corresponding sources is given in the bibliography, where we particularly recommend [RuCo01], [BBDK+01], and [BoCe00].

The goal of this chapter is to describe the framework in which the Linux network architecture operates. All areas described below offer basic functions required to offer network services in the first place. This is the reason why knowing them is an essential prerequisite for an understanding of the implementation of the Linux networking architecture.

Figure 2-1 shows the structure of the Linux kernel. The kernel can be divided into six different sections, each possessing a clearly defined functionality and offering this functionality to the other kernel components. This organization is reflected also in the kernel's source code, where each of these sections is structured in its own subtree.

Figure 2-1. Structure of the Linux kernel according to [RuCo01].

[View full size image]



Here we briefly describe these components.

- Process management: This area is responsible for creating and terminating processes and other activities of the kernel (software interrupts, tasklets, etc.). In addition, this is the area where interprocess communication (signals, pipes, etc.) takes place. The scheduler is the main component of process management. It handles all active, waiting, and blocked processes and takes care that all application processes obtain their fair share of the processor's computing time.

- Memory management: The memory of a computer is one of the most important resources. A computer's performance strongly depends on the main memory it is equipped with. In addition, memory management is responsible for allowing each process its own memory section, which has to be protected against access by other processes.

- File systems: In UNIX, the file system assumes a central role. In contrast to other operating systems (e.g., Windows NT), almost everything is handled over the file-system interface. For

◀ Previous Next ▶

2.1 Monolithic Architectures and Microkernels

In contrast to current operating-system developments tending toward a microkernel architecture, the Linux operating system is based on a monolithic kernel. In microkernel architectures, such as the Mach kernel [Tane95] or the Windows NT kernel, the operating system kernel represents merely the absolute necessary minimum of functionality. Good examples are interprocess communication (IPC) and memory management (MM). Building on the microkernel, the remaining functionality of the operating system is moved to independent processes or threads, running outside the operating system kernel. They use a defined interface to communicate with the microkernel, generally via system calls.

In monolithic kernels, to which the Linux kernel belongs, the entire functionality is concentrated in one (large) kernel. In addition to the basic mechanisms known from microkernels, the Linux operating system kernel also includes device drivers, file system drivers, most instances of the network protocols, and much more. (See Figure 2-1.) Compared to microkernel architectures, the use of a monolithic kernel has both benefits and drawbacks, as we will see below.

The benefits include the fact that the entire functionality of the operating system is concentrated in the kernel, allowing the system to work more efficiently. You can access resources directly from within the kernel, so costly system calls and context changes are needed less frequently. One major drawback is that the source code for the operating system kernel can quickly become rather complex, even messy, because no defined interfaces are required within the kernel. In addition, the development of new drivers can be made more difficult by the lack of an interface definition. For example, if you install a new device, you have to retranslate the entire kernel to ensure that this device driver can be compiled with the kernel, a need avoided by microkernel architectures.

That Linux is based on a monolithic operating-system kernel is due to historical reasons. A system that had not been planned to become such a big project, at the beginning, has continually been developed further, so that it became impossible, at some point in time, to migrate to a microkernel architecture. However, since Version 2.0, Linux has made a step towards microkernel architectures. More specifically, the possibility was created of moving certain functionalities into modules, which are loaded into the kernel at runtime, from which they can be removed again.

This removed an important drawback of monolithic kernels and opened the way to loading drivers or other functionalities at runtime. In addition, modularization offers another benefit: Uniform interfaces are defined. This feature had previously been characteristic only of microkernel architectures. Linux has a number of such interfaces, allowing the kernel to be dynamically enhanced by a number of functionalities. This very flexibility and openness of its interfaces is one of the most important benefits of Linux.

Table 2-1 shows a selection of the most important interfaces, including the pertinent methods used to register and unregister functionalities.

Table 2-1. Interfaces in the Linux kernel to embed new functionalities.

Functionality	Functions for Dynamic Registration
Character devices	`(un)register_chrdev( )`
Block devices	`(un)register_blkdev( )`
Binary formats	`(un)register_binfmt( )`
File systems	`(un)register_filesystem( )`
Serial interfaces	`(un)register_serial( )`
Network adapters	`(un)register_netdev( )`

◀ Previous Next ▶

2.2 Activities in the Linux Kernel

Linux is a multitasking system. This means that several application processes can be active, and several applications can be used, simultaneously. In multiprocessor systems, which have been supported since kernel Version 2.0, even several applications or their processes can be processed in parallel. However, a process is not the only form of activity you can execute in the Linux kernel.

2.2.1 Processes and System Calls

Processes are normally activities that are started to run a specific application, and they are terminated once the application is through. Creating, controlling, and destroying of processes are tasks handled by the kernel of an operating system. Processes operate exclusively in the user address space (i.e., in unprotected mode) of a processor, where they can access only the memory section allocated to them. An attempt to access memory sections of other processes or the kernel address space leads to an exception, which has to be dealt with by the kernel.

However, when a process wants to access devices or use a functionality of the operating-system kernel, it has to use a system call to do this. A system call causes the processor to change to the protected mode, and access to the kernel address space is a function of the system call. All devices and memory sections can be accessed in protected mode, but only with methods of the kernel.

The work of processes and system calls can be interrupted by other activities. In such a case, their current state (contents of CPU registers, MMU registers, etc.) is saved; then it is restored when the interrupted process or system call resumes its work. Processes and system calls can be stopped voluntarily or involuntarily. In the first case, they cede processing voluntarily? for example, when they wait for a system resource (external device, semaphore, etc.) and go to sleep until that resource becomes available. Involuntary cession of processing is caused by interrupts, which tell the kernel that an important action has taken place, one that the kernel should be dealing with. This could be a notification about availability of a previously busy resource.

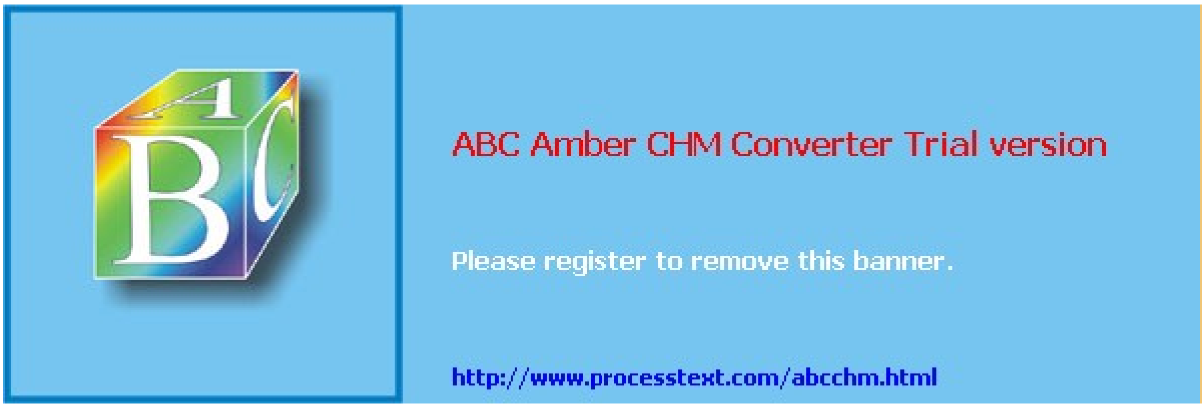
In addition to normal processes and to processes within a system call, we distinguish between further forms of activity in the Linux kernel. These forms of activity are of decisive importance for the Linux network architecture, because the network functionality is handled in the kernel. We will explain the following forms of activity in more detail in the next sections, when we will be discussing mainly their tasks within the Linux network architecture:

- Kernel threads;
- interrupts (hardware IRQs);
- software interrupts (soft IRQs);
- tasklets; and
- bottom halves.

When thinking of the different forms of activity in the kernel (except processes in the system call and kernel threads), an important point will be the parallel execution of the respective form of activity. On the one hand, this concerns the question of whether the instance of a form of activity can be executed concurrently on several processors; on the other hand, of whether two different instances of one form of activity can be executed concurrently on several processors. Table 2-2 shows an overview of these possibilities.

Table 2-2. Concurrent execution of same activities on several processors.

	Same Activity	Different Activities
HW IRQ	?/P>	•
Soft IRQ	•	•



◀ Previous Next ▶

2.3 Locking? Atomic Operations

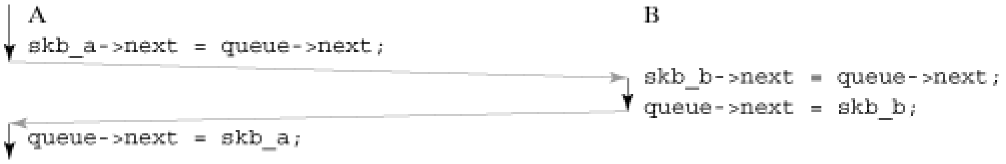
Several different forms of activity can operate and interrupt each other in the Linux kernel. (See Section 2.2.) In multiprocessor systems, different activities even operate in parallel. This is the reason why it is very important for the stability of the system that these operations run in parallel without undesired side effects.

As long as the activities in the Linux kernel operate independently, there will not be any problem. But as soon as several activities access the same data structures, there can be undesired effects, even in single-processor systems.

Figure 2-2 shows an example with two activities, A and B, trying to add the structures `skb_a` and `skb_b` to the list queue. At some point, activity A is interrupted by activity B. After some processing of B, A continues with its operations. Figure 2-3 shows the result of this procedure of the two activities. Structure `skb_b` was added to the list correctly.

Figure 2-2. Activity B interrupts activity A in the critical section.

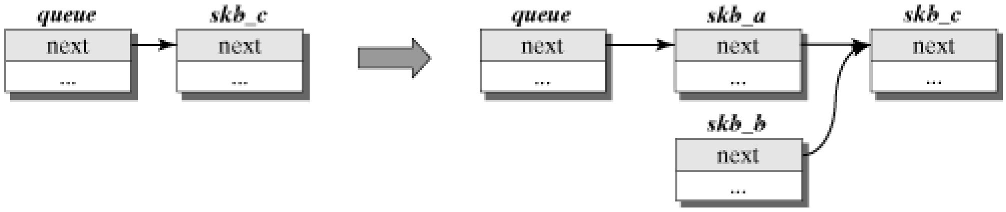
[View full size image]



```
skb_a->next = queue->next;
                                        skb_b->next = queue->next;
                                        queue->next = skb_b;
queue->next = skb_a;
```

Figure 2-3. (Undesired) result of the unprotected operations of activities A and B.

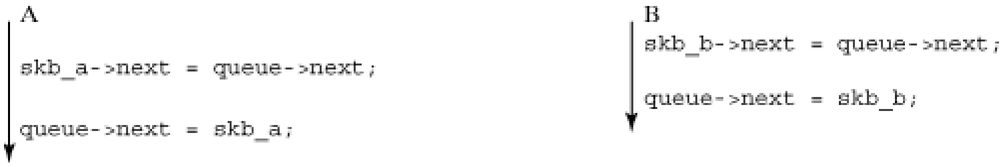
[View full size image]



Undesired results can also occur in multiprocessor systems when the two activities A and B run quasi-in-parallel on different processors, as in the example shown in Figure 2-4.

Figure 2-4. Parallel operations of the activities A and B in the critical section.

[View full size image]



```
                                        skb_b->next = queue->next;
skb_a->next = queue->next;
                                        queue->next = skb_b;
queue->next = skb_a;
```

To avoid these problems when several activities operate on a common data structure (the so-called critical section), then these operations have to be atomic. Atomic means that an operation composed of several steps is executed as an undividable operation. No other instance can operate on the data structure concurrently with the atomic operation (i.e., no other activity can access a critical section that's already busy [Tan95]).

The next four sections introduce mechanisms for atomic execution of operations. These mechanisms differ mainly in the way they wait for entry into a potentially occupied critical section, which implicitly depends on the size of the critical section and the expected waiting time.

2.3.1 Bit Operations

Atomic bit operations form the basis for the locking concepts spinlocks and semaphores, described in the following subsections. Locks are used to protect critical sections, and they are normally

2.4 Kernel Modules

We explained in Section 2.1 that monolithic operating-system kernels, including the Linux kernel, have the drawback that all functionality of the operating system is accommodated in a large kernel, making this kernel big and inflexible. To add a new functionality to the operating-system kernel, you first have to create and install a new kernel. This is a rather cumbersome task and can also be expensive, because running applications have to be interrupted and the system has to be restarted. Moreover, using an operating-system kernel that includes all possible kinds of functions, drivers, and protocols is not recommended either, because the kernel would then become huge and consume an unnecessary amount of memory. In addition, there are always new functionalities we would like to integrate into the kernel, or newer versions of existing functionalities, where errors have been removed. In fact, we can assume that the set of functions of an operating-system kernel will change over time. For this reason, monolithic kernels have to be continually updated? with the problems described above.

Linux is based on the monolithic approach, but it has used a different method to solve the problems noted, since kernel Version 2.0. Note that it does not opt for the microkernel-based approach, which also has drawbacks. The solution are kernel modules. These modules can be easily added to the kernel at runtime and they behave as if they had belonged to the monolithic kernel since the system started. When the functionality of a module is no longer needed, then it can simply be removed and the memory space it used is freed.

We saw in Figure 2-1 in which components of the kernel we can use modules: device drivers, file systems, network protocols, and network drivers. The use of modules is actually not limited to these components. Modules can normally be used on an individual basis. However, adding some functionality means that you need a corresponding kernel interface to inform the rest of the kernel about the new components. The interfaces of the Linux network architecture and the possibilities to expand it by new functionalities are one of the central issues of this book.

When compiled as kernel modules, new functionalities can be added as needed and removed once you don't need them anymore. (See Section 2.4.1.) This means that the principle of modularization is very similar to the flexibility of microkernels, the only difference being that Linux modules run in the k ernel address space, components of microkernel systems in the user address space. More specifically, the Linux module concept combines the benefits of both operating-system variants. On the one hand, it avoids the expensive change of address spaces known from the microkernel-based approach; on the other, it lets you expand the kernel functionality individually at runtime at the same time.

The following sections take a closer look at the structure and management of kernel modules, because modules are the best and most flexible option to enhance the Linux network architecture. Unfortunately, a detailed description of kernel modules would go beyond the scope of this book; we refer mainly to [RuCo01] and [BBDK+01] instead.

2.4.1 Managing Kernel Modules

A kernel module consists of object code, which is loaded into the kernel address space at runtime, where it can be executed. When the system starts, it is not known which modules with what functionalities should be loaded, so the module has to make itself known to the respective components of the kernel. A module should also remove all references to itself when it is removed from the kernel address space. There are two methods available for these tasks, which each kernel module should implement? namely, `init_module()` and `cleanup_module()`. We will have a closer look at these methods in Section 2.4.2; first, however we need some general information about the management of kernel modules outside the kernel.

The following tools are used to manually load a module into the kernel, or remove it from the system:

- `insmod Modulename.o [arguments]`? This command tries to load a kernel module into the kernel address apace. In a successful case, the object code of the module is linked to the kernel; the module can now access the symbols (functions and data structures) of the kernel. Calling `insmod` causes the following system calls to run implicitly:
    - `sys_create_module()` allocates memory space to accommodate the module in the kernel address space.

◀ Previous Next ▶

2.5 Device Drivers

UNIX has its own way of handling physical devices. They are hidden from the user and accessible only over the file system, without limiting their functionality. For example, an application programmer can use the simple file operations `read()` and `write()` to access the driver's hardware, while the `ioctl()` command can be used to configure properties of a device.

Device drivers in the form of modules can be added or removed in Linux at any time. This offers you a comfortable tool to develop and test new functionalities. Figure 2-7 shows an excerpt from the `/dev` directory, where all devices are listed. In Linux, network adapters are not treated as normal devices and so they are not listed in the `/dev` directory. Linux has a separate interface for them. The reasons are described in [RuCo01]. Chapter 5 will discuss network devices in detail.

Figure 2-7. Excerpt from the `/dev` directory.

```
brw-rw----      1 root      disk   3,       0 May 12 19:23 hda
brw-rw----      1 root      disk   3,       1 May 12 19:23 hda1
brw-rw----      1 root      disk   3,       2 May 12 19:23 hda2
brw-rw----      1 root      disk   3,      64 May 12 19:23 hdb
brw-rw----      1 root      disk   3,      65 May 12 19:23 hdb1
crw-rw----      1 root      uucp   4,      64 May 12 19:23 ttyS0
crw-rw----      1 root      uucp   4,      65 May 12 19:23 ttyS1
crw-rw-r--      1 root      root  10,       1 Sep 13 08:45 psaux
```

We can see in Figure 2-7 that the entries for device drivers differ from regular directory entries. Each entry includes two numbers used to identify the device and its driver.

- The major number identifies the driver of a device. For example, Figure 2-7 shows that the PS/2 driver has major number 10 and the hard disk driver (`hdxx`) has major number 3.

  The major number can be specified when you register a device driver, but it has to be unique. For drivers you think you will use less often, it is recommended that you let the kernel assign a major number. This ensures that the numbers are all unique. See details in [RuCo01].

- The minor number is used to distinguish different devices used by the same driver. In Linux, a device driver can control more than one device, if the driver is designed as a reentrant driver. The minor number is then used as an additional number to distinguish the devices that driver controls. For example, the hard disk driver with major number 3 in Figure 2-7 controls three hard disks, distinguished by the minor numbers 1, 2, and 65.

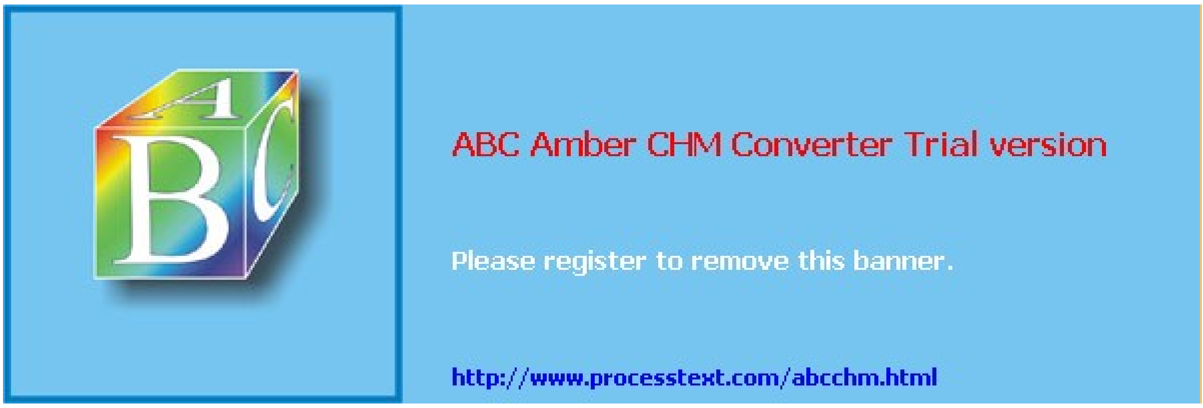
Figure 2-7 also shows that the type of each driver is specified at the beginning of each row. Linux differs between two types of physical devices:

- Block-oriented devices allow you optional access (i.e., an arbitrary set of blocks can be read or written consecutively without paying attention to the order in which you access them). To increase performance, Linux uses a cache memory to access block devices. File system can be accommodated only in block devices (hard disks, CD-ROMs, etc.), because they are required for optional or random access. Block devices are marked with a `b` in the `/dev` directory.

  A block-oriented driver can be registered with the kernel function `register_blkdev()`. If the function was completed successfully, then the driver can be addressed by the returned major number. `Release_blkdev()` is used to release the device.

- Character-oriented devices are normally accessed in sequential order. They can be accessed only outside of a cache. Most devices in a computer are character-oriented (e.g., printer and sound card). Character-oriented devices are marked with a `c` in the `/dev` directory. You can use `register_chrdev()` to register and `release_chrdev()` to release character-oriented devices.

The virtual file `/proc/devices` lists all devices currently known to the kernel. This file is used to find the major number of a driver in the user address space, in case none has been specified during the



2.6 Memory Management in the Kernel

Memory management is one of the main components in the kernel of any operating system. It supplies a virtual memory space to processes, often one much larger than the physical memory. This can be achieved by partitioning memory pages and outsourcing memory pages that are temporarily not needed to the swap memory on the hard disk. Access to an outsourced page by an application is intercepted and handled by the kernel. The page is reloaded into the physical memory and the application can access the memory without even noticing anything about insourcing and outsourcing of things.

The memory residing in the kernel cannot be outsourced because, if the memory management were to move the code to the swap memory, it would not be available later on, and the system would be blocked. For this and, of course, performance reasons, the memory of the kernel cannot be outsourced. Therefore, we will always distinguish between the kernel address space and the user address space in the rest of this book.

Virtual memory management is one of the most important and most complex components of an operating system. [Tan95] offers an overview of the theory of virtual memory management, and detailed information about its implementation in the Linux kernel is described in [BBDK+01] and [BoCe00]. Within the Linux network architecture, the structure of the virtual memory management is less interesting; it is of interest only in regard to whether memory can be reserved and released in an efficient way, as we will see in the following section. We will also introduce methods to exchange data between the kernel address space and the user address space. Section 2.6.2 ends with a brief introduction of the slab cache, representing an efficient management of equalsized memory spaces (for example, similar to those use for socket buffers).

2.6.1 Selected Memory Management Functions

This section introduces the basic functions of memory management a programmer writing kernel components or kernel modules needs. First, we will discuss how memory spaces can be reserved and released in the kernel. Then we will introduce functions used to copy data between the kernel address space and the user address space.

Reserving and Releasing Memory in the Kernel

`kmalloc()` mm/slab.c

`kmalloc(size, priority)` attempts to reserve consecutive memory space with a size of `size` bytes in the kernel's memory. This may mean that some more bytes will be reserved, because the memory is managed in the kernel in so-called slabs. Slabs are caches, each managing memory spaces with a specific size. (See `/proc/slabinfo`.) Letting a slab cache reserve memory space is clearly better performing than many other methods [Tan95].

The parameter `priority` can be used to specify options. We will briefly describe the most important options below and refer our readers to [RuCo01] for a detailed explanation of the large number of options offered by `kmalloc()`. The abbreviation `GFP_` means that the function `get_free_pages()` may be used to reserve memory.

- `GFP_KERNEL` is normally used when the requesting activity can be interrupted during the reservation. It can also be used for processes that want to reserve memory within a system call. For activities that must not be interrupted (e.g., interrupt routines), `GFP_KERNEL` should not be used.
- `GFP_ATOMIC` is the counterpart of `GFP_KERNEL` and shows that the memory request should be atomic (i.e., without interrupting the activity).
- `GFP_DMA` shows that memory in the DMA-enabled area should be reserved.
- `GFP_DMA` can be combined with one of the two previous flags.

◀ Previous Next ▶

2.7 Timing in the Linux Kernel

In the Linux kernel, clocks "tick" slightly different by than they do in the real world. The time does not progress continually, but in increments of 10 ms (milliseconds) each, which is called a tick. This means that the time virtually stands still between any two ticks. The number of ticks since the system started is recorded in a variable called `jiffies` in the kernel. The timer interrupt increments the `jiffies` variable at each interrupt. The terms ticks and jiffies are often used interchangeably.

The resolution frequency of the timer interrupt is initialized to the value of the variable HZ (`include/asm/param.h`), and it increments the `jiffies` variable every $\frac{1}{HZ} s$.[4] This length of time is absolutely sufficient for normal applications, because a higher interrupt frequency would only mean a higher load on the system due to too many unnecessary interruptions [RuCo01]. However, there are certain situations where a high timer resolution is required, especially to measure smaller time increments or for running actions at specific points in time [WeRi00]. In networks, you often find such requirements for protocol instances, for example protocol instances that have to calculate packet run times or traffic shapers that have to measure minimum time intervals in the microsecond range.

Most of these tasks require clocks with a resolution that is at least in the microsecond range. For example, to implement a traffic shaper [Tane97], you have to calculate the number of bytes that could be sent within a specific interval. For example, the `jiffies` time measurement with a resolution of 100 Hz is not suitable. With a rate of 2 Mbits/s, an interval of $\frac{1}{100} s$ already corresponds to a packet with a length of 2500 bytes.

To avoid this problem, most modern processors (Pentium, Alpha, etc.) have appropriate registers. They have been added to those processors mainly to allow system performance measurements and less for traffic shaping in networks. But, while they are present, their use is quite popular. In the Pentium processor and its successors (and most of its clones), this is a 64-bit-wide TSC (Time Stamp Counter) register; its content is incremented by a value of one in each processor clock. The content of this register shows the number of elapsed clock cycles since system start.

The TSC register is actually nothing more than a hardware variant of `jiffies`, except that its resolutions is higher by a factor of between 10^6 and 10^8. This means, for example, that you can measure intervals with an accuracy of 0.001 µs in a Pentium processor with a clock rate of 1 GHz.

Nevertheless, there is a certain inaccuracy when measuring with the TSC register, because it takes a few clocks (approx. ten) to read the register. The reason is the main memory access that occurs after the register value has been read. It can be done only in the bus frequency, which corresponds to a fraction of the CPU frequency. In addition, there could be effects in the first-level and second-level cache accesses that can easily lead to false measurements. However, the error caused by the TSC register is meaningless for normal measurements, because most of them measure only relatively big time cycles (in the 1-µs range). The command `get_cycles()` (defined in `<asm/timex.h>`) can be used to read the content of the TSC register.

2.7.1 Standard Timers

In addition to measuring intervals in the microsecond range, we also need a way to run a function at a specific point in time to implement a traffic shaper [WeRi00], which sends packets at specific points in time. The resolution of such a timer should be at least in the 100-µs range. However, due to the fact that a PC has only one timer component, you can use only this one. As described above, the interrupt is triggered `HZ` times per second. In addition to updating `jiffies`, Linux uses the timer interrupt to run functions at specific points in time (i.e., the timer handler).

A timer queue can be used when a function of the kernel should run at a specific point in time (e.g., switching off the floppy motor). At each occurrence of a timer interrupt, the timer interrupt routine updates the `jiffies` variable and also checks the timer queue for timer handling routines, as may be present. Each `timer_list` structure within the timer queue stands for one function (timer handling routine), which is to run at a specific point in time `(expires)`. The exact process of the timer resolution and of subsequent checking of the timer queue is described in [RuCo01].

[4] HZ depends on the architecture: In Alpha processors, HZ = 1024; HZ = 100 in most other architectures.

◂ Previous Next ▸

2.8 The Proc File System

All files in the `/proc` directory are virtual files. They do not exist on any memory medium, but are generated directly by the kernel upon each read access. A proc file is normally a text file showing information about specific parts of the kernel. For example, the commands `lspci` or `apm` show you information from the proc files `/proc/pci` und `/proc/apm`, respectively, and information about the current devices on the PCI bus or the state of the notebook battery.

The possibilities of the proc file system to display information on the kernel easily in the user mode are used by many system developers. Files and directories in the `/proc` directory can be easily implemented. In addition, you can register and unregister dynamically, so that the proc directory is often used by modules.

The files and directories in the /proc directory are essentially based on the `proc_dir_entry` structure, shown in Figure 2-10. Such a structure represents either a directory or a file. The directory proc is represented by the variable `proc_root`. The attributes and methods of the `proc_dir_entry` structure have the following meaning:

Figure 2-10. Structure of `proc_dir_entry`.

```
struct proc_dir_entry
{
      unsigned short            low_ino;
      unsigned short            namelen;
      const char                *name;
      mode_t                    mode;
      nlink_t                   nlink;
      uid_t                     uid;
      gid_t                     gid;
      unsigned long             size;

      ...
      struct proc_dir_entry     *next, *parent, *subdir;
      void                      *data;
      int                       (*get_info)(buffer, start, off, count);
      int                       (*read_proc) (buffer, start, off, count,
eof, data);
      int                       (*write_proc)(file, buffer, count, data);
      int                       (*readlink_proc)(proc_dir_entry, page);
      unsigned int              count; /* use count */
      int                       deleted; /* delete flag */
};
```

- `low_ino` is the file's Inode number. This value is filled automatically by `proc_register` when the file is initialized.
- `namelen` specifies the length of the file or directory name, `name`.
- `name` is a pointer to the name of the file (or directory).
- `mode` specifies the file's mode; this value is set to `S_DIR` for directories.
- `nlink` specifies the number of links to this file (default = 1).
- `uid` or `gid` specifies the user or group ID of the file.
- `size` specifies the length of the file as shown when the directory is displayed.
- `data` is a pointer that can point to private data.

◂ Previous Next ▸

2.9 Versioning

The Linux kernel is subject to constant improvement and development, and new versions (releases) are published regularly. To prevent users from getting confused and to identify stable versions, we distinguish between so-called hacker and user kernels. The version of a Linux kernel is denoted by a tuple composed of three letters, `x,y,z`:

- A hacker kernel is not a kernel version used by malicious people to break into highly classified computers. The very opposite is the case; in fact, a hacker kernel is the prototype of a Linux kernel under further development. Normally, new concepts and functions have been added to such a prototype and some errors of the previous version have been (hopefully) removed. Hacker kernels are in the testing phase, and faulty behavior or system failure has to be expected at any time. They mainly serve to integrate and test new drivers and functionalities.

  Once a sufficient number of new drivers and technologies have been added to a hacker kernel, Linus Torvalds will proclaim a so-called feature freeze. This means that no new functionality can be integrated, and the only change allowed to that prototype is to remove errors. The objective is a stable user kernel. You can identify a hacker kernel by its odd y version number (e.g.,2.3.z, where z denotes the consecutive number of the kernel version). The next version (e.g., 2.3.51), will then have removed some errors of 2.3.50.

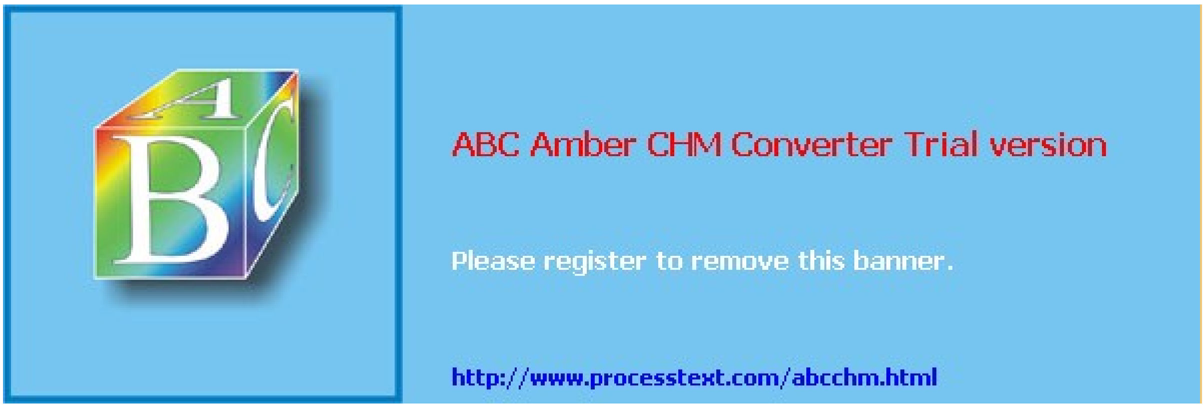
- User kernels are stable kernel versions, where you can assume that they are normally free from errors. A user kernel is denoted by an even version number, e.g., 2.2.z. Such versions are recommended to normal users, because you don't have to fear that the system might c rash. For example, when version 2.3.51 is very stable and the feature freeze has already been proclaimed, then the kernel will be declared user kernel 2.4.1. New drivers and properties will then be added to hacker kernel 2.5.1.

◂ Previous Next ▸

◀ Previous Next ▶

Part II: Architecture of Network Implementation

◀ Previous Next ▶



◂ Previous Next ▸

Chapter 3. The Architecture of Communication Systems

This chapter discusses basic models used to structure communication systems and architectures. The ISO/OSI reference model introduced in Section 3.1.1 failed in practice because of its complexity, especially that of its application-oriented layers. Nevertheless, it still has some fundamental significance for the logical classification of the functionality of telecommunication systems. Though it was less successful in proliferating than expected, this model offers the proposed structure of telecommunication systems in similar form in the field of telematics.

Currently, the technologies and protocols of the Internet (TCP/IP reference model; see Chapter 13) have made inroads and are considered the de facto standards. The architecture of the Internet can easily be paralleled to the ISO/OSI reference model, as far as the four lower layers are concerned. The other layers are application-specific and cannot be compared to the ISO/OSI model.

However, the architecture and protocols of the Internet also represent a platform for open systems (i.e., no proprietary solutions supported by specific manufacturers are used in the network). In addition, the development process for new protocols in the Internet by the Internet Engineering Task Force (IETF) is open for everyone and is designed so that the best and most appropriate technical proposals are accepted.

◂ Previous Next ▸

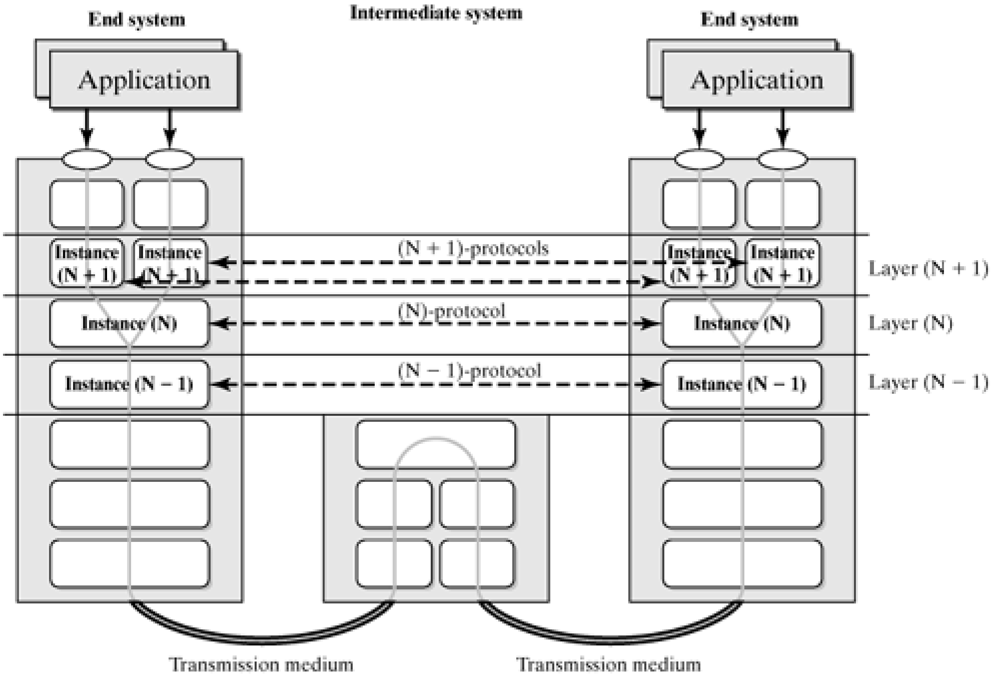
3.1 Layer-Based Communication Models

Telecommunication systems bridge the spatial distance between distributed computers. The implementation of this task is extremely complex for a number of reasons, so it is not recommended to use a monolithic architecture, which could prove very inflexible and difficult to maintain. This is the reason why communication systems are normally developed as layered architectures, where each layer assumes a specific task, offering it in the form of services. The ISO/OSI reference model is probably the best known example of such a layered architecture.

To solve its task, a layer, N, must use only the services provided by the next lower layer (N ?1). More specifically, layer N expands the properties of layer N ?1 and abstracts from its weaknesses. For this purpose, the instance of layer N communicates with the instances of the same layer on other computers. This means that the entire functionality of the communication system is available in the top layer. In contrast to a monolithic structure, layering a communication system means a more expensive implementation, but it offers invaluable benefits, such as the independent development of single partial components, easy exchange of single instances, better maintainability, and higher flexibility. Figure 3-1 shows the principles of communication in a layered system.

Figure 3-1. Communication in layered systems.

[View full size image]



We can deduce two central terms for layer-oriented communication models from the current section, which will be discussed in more detail in Section 3.2:

- Communication between two instances of the same layer on different computers is governed by predefined rules. These rules are called protocols.
- The set of functions offered by a layer, N, to its higher-order layer (N + 1), is called its service. The interface through which this service is offered is called service interface.

This means that an instance is the implementation of a communication protocol and the service provided within one layer on a computer. The theoretical basis of services and protocols are discussed in Section 3.2.

3.1.1 The ISO/OSI Reference Model

At the end of the seventies, experts observed increasingly that the interconnection of several computer networks was difficult (because of vendor-specific properties of these networks), if not impossible, so it was found hard to ensure interoperability between the large number of networks in place. This situation led to the proposal to create a uniform and standardized platform for computer-based

3.2 Services and Protocols

Services and protocols were briefly discussed in Section 3.1; they are basic elements of layered communication systems. This section describes the meaning of these two terms and the functionality of services and protocols. These two terms serve as a theoretical basis for further explanations in this book, where we will focus on services and protocols used in real-world systems.

We know from the models described in the previous sections that modern telecommunication systems consist of several layers. Each layer has different purposes (depending on the reference model) and offers services to the next higher layer. For example, the IP layer in the TCP/IP reference model offers the following services: forwarding data units (without guarantees) from a local computer to another computer, specified by its IP address. This service is used by the transport layer (e.g., by TCP) and expanded so that a byte stream can be transmitted free from errors and in the correct order.

We can say that a service describes the set of functions offered to the next higher layer. In addition, a service defines single service elements, used to access the entire range of services. In other words, the service definition defines the extent and type of service and the interface used to call that service. The definition of a service refers only to the interaction between two neighboring layers and the interfaces concerned. The literature describes this often as vertical communication. Exactly how a layer provides its service is not part of the service definition; it only deals with what an implementation has to offer the service user at the interface.

To be able to use the services of a layer, the participating systems have to overcome the spatial separation and coordinate their communication. This is achieved by use of communication protocols, which run by instances of a layer in the communicating systems. A protocol regulates the behavior of the distributed instances and defines rules for their coordination. For example, it defines messages to be exchanged between the instances to regulate distributed handling between these instances. More specifically, a layer, N, provides its service by distributed algorithms in the respective instances of layer N and by exchanging protocol messages about their coordination. (See Figure 3-1.) Coordination between the instances by protocol messages is also called horizontal communication. The service of the lower layer (N ?1) is used to exchange protocol messages.

The specification of a service describes the behavior of a layer versus the next higher layer (vertical communication), but says nothing about how a service is implemented. It merely defines the format and dynamics at the interfaces to the layer that uses the service. A service is rendered by instances of a layer, which use protocols to coordinate themselves (horizontal communication). The protocol specification describes the syntactic and dynamic aspects of a protocol. The protocol syntax describes the format of the protocol data units (PDUs) to be exchanged and the protocol dynamics describe the behavior of the protocol. The goal of this book is to explain how all of these elements can be designed and implemented in a communication system. Using Linux as our example operating system, we will see what the interfaces between the different layers can look like and what design decisions play a role, mainly from the perspective of efficiency and correctness of the protocols. In addition, we will see how different protocols use their instances, to show the technologies used to implement network protocols.

3.2.1 Interplay of Layers, Instances, and Protocols

After our brief introduction to services and protocols in the previous sections, this section describes the horizontal and vertical processes involved when protocol instances provide a service. The description of these processes forms the basis for understanding how network protocols work, mainly the principles of horizontal and vertical communication. The terms introduced earlier will help us better classify and distinguish structures and parameters involved in the interaction of different layers at the interfaces.

Instances are the components offering services within a layer. To offer a service, the instances of a layer communicate (horizontally). This communication is realized by exchanging protocol data units (PDUs) of layer N. However, data is not exchanged directly between the two instances, but indirectly, over the next lower layer. This means that the instance of layer N uses the service of layer (N ?1) to exchange a PDU with its partner instance. Figure 3-4 shows the interplay of layers and the elements involved.

Figure 3-4. Data units for vertical and horizontal communication.

◀ Previous Next ▶

Chapter 4. Managing Network Packets in the Kernel

One of the most important tasks of the network subsystem of an operating system is to process data packets according to the protocols used. In the designing of such a system, the multitude and flexibility of available methods play an important role, in addition to the performance and correctness of these protocols. Many network protocols differ a lot externally, but, when you implement them within an operating system, you can see quickly that the algorithms and operations on data packets are similar, and most of them can be reused. This chapter uses a Linux system as an example to show how data packets can be realized and what general methods are available to manipulate them.

One main reason for the flexibility and efficiency of the Linux network implementation is the architecture of the buffers that manage network packets? the so-called socket buffers, or skb for short. This central structure of the network implementation represents a packet during its entire processing lifetime in the kernel, representing one of the two basic elements of this network implementation, in addition to network devices. This means that a socket buffer corresponds to a sending or received packet.

This chapter introduces buffer management (i.e., the structure of socket buffers) and the operations used to manage or manipulate them. Beginning with an introduction to the `sk_buff` structure, we will use an example to show how an IP packet is represented in this structure and how it changes along its way across different protocols and layers. In addition, this chapter introduces functions used to manage and change the structure.

◀ Previous Next ▶

◂ Previous Next ▸

4.1 Socket Buffers

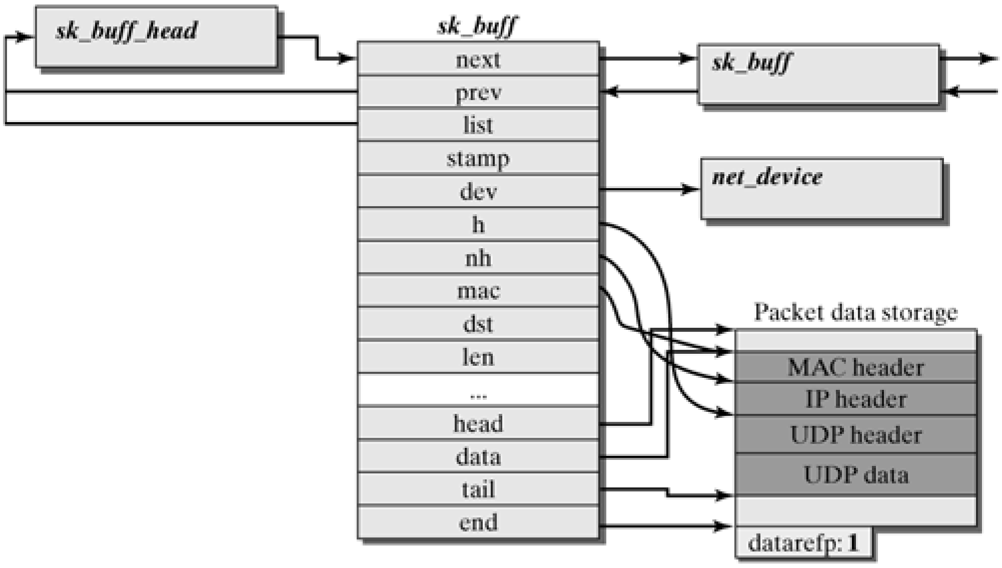
The network implementation of Linux is designed to be independent of a specific protocol. This applies both to the network and transport layer protocols (TCIP/IP, IPX/SPX, etc.) and to network adapter protocols (Ethernet, token ring, etc.). Other protocols can be added to any network layer without a need for major changes. As mentioned before, socket buffers are data structures used to represent and manage packets in the Linux kernel.

A socket buffer consists of two parts (shown in Figure 4-1):

- Packet data: This storage location stores data actually transmitted over a network. In the terminology introduced in Section 3.2.1, this storage location corresponds to the protocol data unit.
- Management data (`struct sk_buff`): While a packet is being processed in the Linux kernel, the kernel requires additional data that are not necessarily stored in the actual packet. These mainly implementation-specific data (pointers, timers, etc.). They form part of the interface control information (ICI) exchanged between protocol instances, in addition to the parameters passed in function calls.

Figure 4-1. Structure of socket buffers (`struct sk_buff`) with packet storage locations.

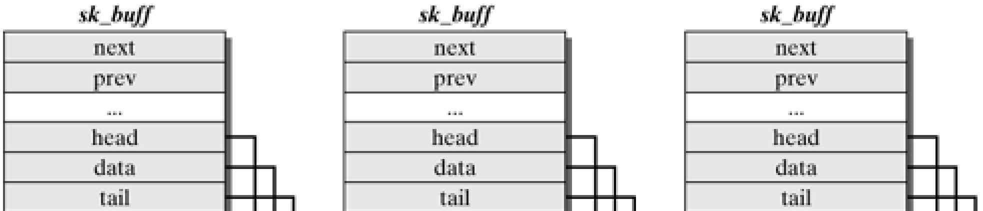
[View full size image]

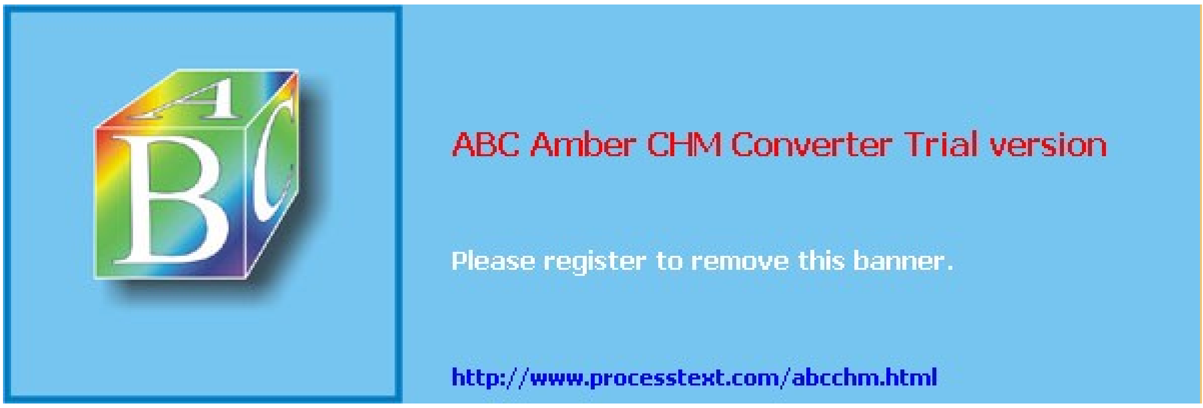


The socket buffer is the structure used to address and manage a packet over the entire time this packet is being processed in the kernel. When an application passes data to a socket, then the socket creates an appropriate socket buffer structure and stores the payload data address in the variables of this structure. During its travel across the layers (see Figure 4-2), packet headers of each layer are inserted in front of the payload. Sufficient space is reserved for packet headers that multiple copying of the payload behind the packet headers is avoided (in contrast to other operating systems). The payload is copied only twice: once when it transits from the user address space to the kernel address space, and a second time when the packet data is passed to the network adapter. The free storage space in front of the currently valid packet data is called headroom, and the storage space behind the current packet data is called tailroom in Linux.

Figure 4-2. Changes to the packet buffers across the protocol hierarchy.

[View full size image]





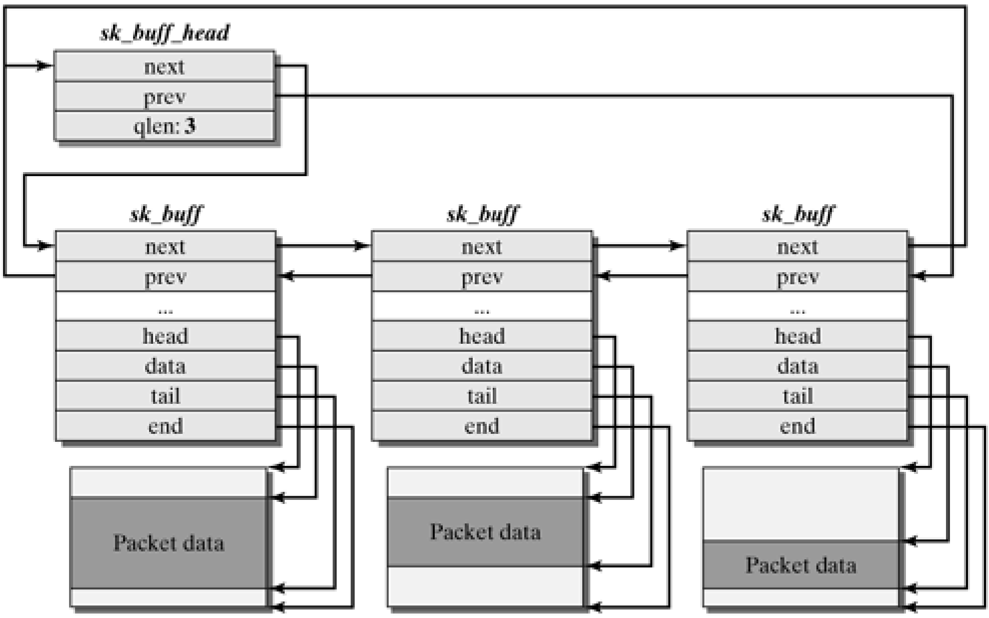
4.2 Socket-Buffer Queues

When a packet is currently not handled by a protocol instance, it is normally managed in queues. Linux supports the management of packets in a queue structure (`struct sk_buff_head`) and in a number of operations on this structure. The programmer can use these functions to abstract from the actual implementation of a socket buffer and queues to easily change the underlying implementation of the queue management.

Figure 4-6 shows that the socket buffers stored in a queue are dual-concatenated in a ring structure. This dual concatenation allows quick navigation in either of the two directions. The ring structure facilitates concatenation and prevents the occurrence of `NULL` pointers.

Figure 4-6. Packet queues in the Linux kernel.

[View full size image]



A queue header consists of the following `skb_queue_head` structure:

```
struct sk_buff_head
{
        struct sk_buff *next;
        struct sk_buff *prev;
        __u32 qlen;
        spinlock_t lock;
};
```

- `next` and `prev` are used to concatenate socket buffers; `next` points to the `first` and `prev` to the last packet in the queue.
- `qlen` specifies the current length of the queue in packets.
- `lock` is a spinlock (see Section 2.3.2) and can be used for atomic execution of operations on the queue. When a critical access occurs, if the spinlock is not free, the access will have to wait until it is released.

4.2.1 Operations on Socket-Buffer Queues

Socket-buffer queues are a powerful tool to arrange packets in Linux. The power of this functionality is complemented by a large number of methods to manage socket buffers in queues.

Most operations on socket buffers are executed during critical phases, or they can be interrupted by

Chapter 5. Network Devices

Each (tele)communication over a network normally requires a physical medium, which is accessed over a network adapter (network interface). Together, the network adapter and the medium eventually allow bridging of the spatial distance, so that data can be exchanged between two or more communication systems. If we use the ISO/OSI reference model introduced in Section 3.1.1, then the tasks of a network adapter extend over layers 1 and 2a: They include all tasks dealing with data? signal? data conversion (and media access in the case of shared media). All higher-order protocol functions are handled by the protocol instances of the respective operating system.[1] This interface is characterized by the following properties:

[1] This view is limited to software-based communication systems on PC basis. More instances are normally implemented in hardware for dedicated systems.

- interfacing between specialized hardware in the network adapters and software-based protocols;
- asynchronous input and output point of the protocol stack in the operating system kernel.

In the network architecture of the Linux operating systems, this interface between software-based protocols and network adapters is implemented by the concept of network devices. A network-device interface primarily should meet the following requirements:

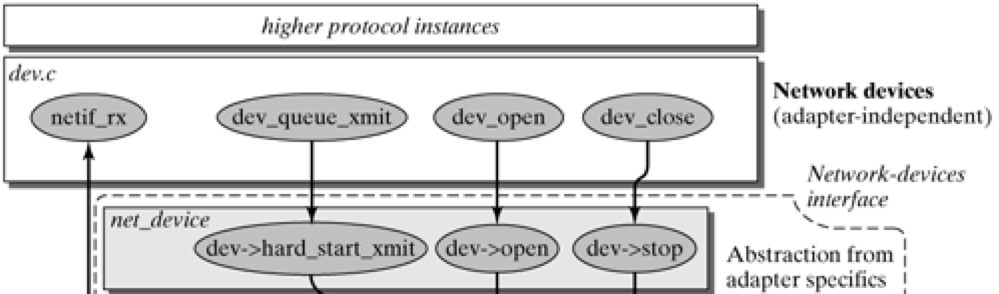
- Abstract from the technical properties of a network adapter: Network adapters might implement different layer-1 and layer-2 protocols and are manufactured by different vendors. This means that their configurations are individual and specific to each network adapter. For this reason, we need a piece of software for each adapter to communicate with the hardware: the driver of a network adapter (which is, by the way, also a protocol).
- Provide a uniform interface for access by protocol instances: In a system like Linux, there are several protocol instances using the services of network adapters. To be consistent with the principle of layered communication systems (see Section 3.1), these instances should be implemented independently of a specific type of adapter. This means that network adapters should have a uniform interface to the higher layers.

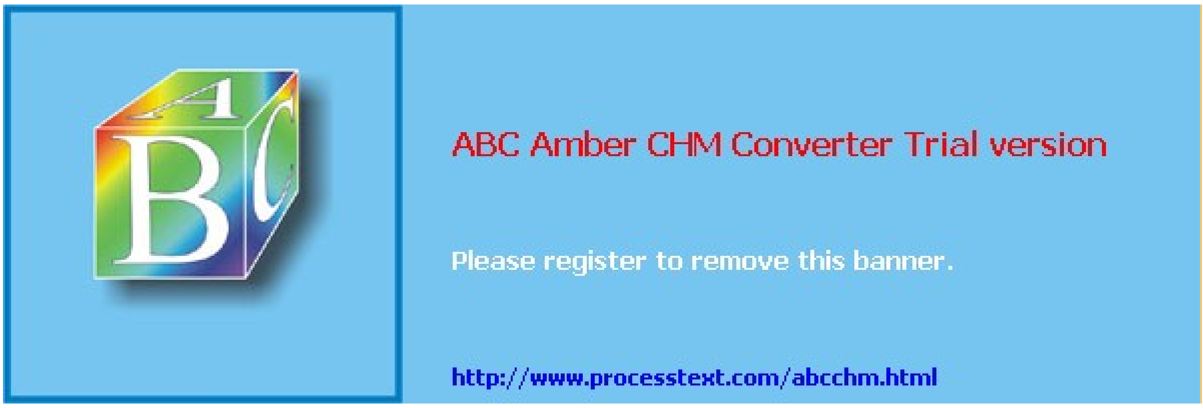
In the Linux kernel, these two tasks are handled by the concept of network devices and are often seen as one single unit. However, it makes sense to distinguish between the two views of network devices and discuss them separately. For this reason, the following section introduces the network-device interface visible from the "top," which offers a uniform interface to the higher protocol instances for physical transmission of data. Later on, Section 5.3 will discuss the "lower" half: the adapter-specific functions that are the actual network driver. Subsequently, Chapter 6 will introduce an example describing how a packet is sent and received on the level of network devices interfacing to the higher protocols.

Not every network device in the Linux kernel represents a physical network adapter. There are network devices, such as the `loopback` network device, that offer a logical network functionality. The interface of network devices is also often used to bind protocols, such as the point-to-point protocol (PPP).

Figure 5-1. The structure of a network device interface.

[View full size image]





◂ Previous Next ▸

5.1 The net_device Interface

In addition to character and block devices, network devices represent the third category of adapters in the Linux kernel [RuCo01]. This section describes the concept of network devices from the perspective of higher-layer protocols and their data structures and management.

Network adapters differ significantly from the character and block devices introduced in Section 2.5. One of their main characteristics is that they have no representation in the device file system `/dev/`, which means that they cannot be addressed by simple `read-write` operations. In addition, this is not possible because network devices work on a packet basis; a behavior comparable to character-oriented devices can be achieved only by use of complex protocols (e.g., TCP). For example, there are no such network devices as `/dev/eth0` or `/dev/atm1`. Network devices are configured separately by the `ifconfig` tool on the application level. More recently, another tool available is `ip`, which can be used for extensive configuration of most network functions.

One of the reasons why network devices are so special is that the actions of a network adapter cannot be bound to a unique process; instead, they run in the kernel and independently of user processes [RuCo01]. For example, a hard disk is requested to pass a block to the kernel: The action is triggered by the adapter (in the case of network adapters), and the adapter has to explicitly request the kernel to pass the packet.

5.1.1 The net_device Structure

`struct net_device` include/linux/netdevice.h

```
struct net_device
{
   char                 name[IFNAMSIZ];
   unsigned long        rmem_end, rmem_start, mem_end, mem_start,
base_addr;
   unsigned int         irq;
   unsigned char        if_port, dma;
   unsigned long        state;
   struct net_device    *next, *next_sched;
   int                  ifindex, iflink;

   unsigned long        trans_start, last_rx;
   unsigned short       flags, gflags, mtu, type, hard_header_len;
   void                 *priv;
   struct net_device    *master;
   unsigned char        broadcast[MAX_ADDR_LEN], pad;
   unsigned char        dev_addr[MAX_ADDR_LEN], addr_len;
   struct dev_mc_list   *mc_list;
   int                  mc_count, promiscuity, allmulti;

   int                  watchdog_timeo;
   struct timer_list    watchdog_timer;

   void                 *atalk_ptr, *ip_ptr, *dn_ptr, *ip6_ptr, *ec_ptr;
   struct Qdisc         *qdisc, *qdisc_sleeping, *qdisc_list,
*qdisc_ingress;
   unsigned long        tx_queue_len;

   spinlock_t           xmit_lock;
   int                  xmit_lock_owner;
   spinlock_t           queue_lock;
   atomic_t             refcnt;

   int                  features;
```

◂ Previous Next ▸

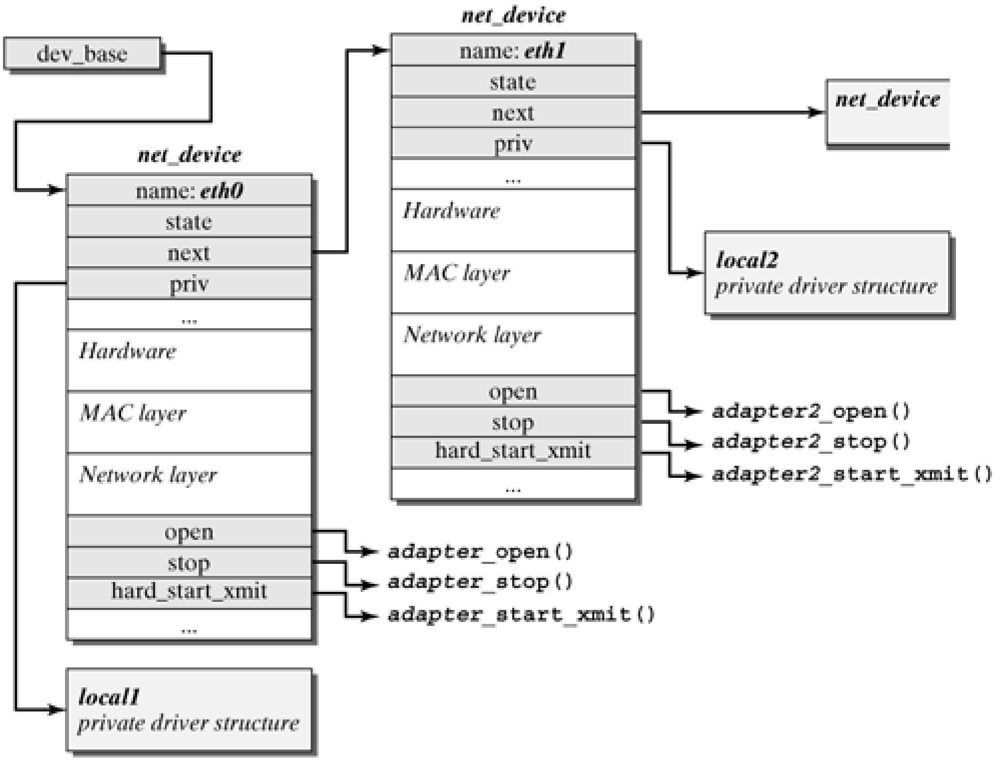
5.2 Managing Network Devices

Now that we know how a network device can be represented by the `net_device` structure in the Linux kernel, this section discusses the management of network devices. First, we will describe how network devices can be linked, then we will introduce methods that can be used to manage and manipulate network devices. As was mentioned earlier, this section will look at network devices only from the "top"? their uniform interface for protocol instances of the higher-order layers.

All network devices in the Linux kernel are connected in a linear list (Figure 5-3). The kernel variable `dev_base` represents the entry point to the list of registered network devices, pointing to the first element in the list, which, in turn, uses `next` to point to the next element. Each `net_device` structure represents one network device.

Figure 5-3. Linking net_device structures.

[View full size image]



The proc directory (`/proc/net/dev`) or the (easier to read) command `ifconfig -a` can be used to call the list of currently registered devices. (See Appendix C.1.)

5.2.1 Registering and Unregistering Network Devices

We know from the previous section that network devices are managed in the list `dev_base`. This list stores all registered network devices, regardless of whether they are activated. When `register_netdevice()` is used to add a new device to this list, then we first have to create and initialize a `net_device` structure for it. This process can be done in two different ways:

- If we specified in the kernel configuration that the driver of a network device should be integ rated permanently into the kernel, then there is already a `net_device` structure. A clever mechanism with preprocessor definitions creates different instances of the `net_device` structure during the translation, depending on the kernel configuration, and these instances are used for the existing network adapters when booting.

  For example, to integrate the driver of an Ethernet card into the kernel, eight `net_device` structures are created for Ethernet network devices, and these structures are initially not all ocated to any card.

- If the driver was translated as a kernel module, then the driver itself has to create a

◀ Previous Next ▶

5.3 Network Drivers

The large number of different protocols in the Linux network architecture leads to considerable differences in the implementations of drivers for different physical network adapters. As was mentioned in the section that described the `net_device` structure, the properties of different network adapters are hidden at the interface of network devices, which means that they offer a uniform view upwards.

Hiding specific functions (i.e., abstracting from the driver used) is achieved by using function pointers in the `net_device` structure. For example, a higher-layer protocol instance uses the method `hard_start_xmit()` to send an IP packet over a network device. Notice, however, that this is merely a function pointer, hiding the method `el3_start_xmit()` in the case of a 3c509 network adapter. This method takes the steps required to pass a socket buffer to the 3c509 adapter. The upper layers of the Linux network architecture don't know which driver or network adapter is actually used. The function pointer can be used to abstract from the hardware actually used and its particularities.

The following sections provide an overview of the typical structuring and implementation characteristics of the functions of a network driver, without discussing adapter-specific properties, such as manipulating the hardware registers or describing the transmit buffers. In general, these tasks depend on the hardware, so we will skip them here. Readers interested in these details can use the large number of network drivers included in the `drivers/net` directory as examples. We use the `skeleton` driver to explain how driver methods work. This is a sample driver used to show usual processes in driver methods rather than a real driver for a network adapter. For this reason, it is particularly useful for explaining the implementation characteristics of network drivers.[2]

[2] At this point, we would like to thank Donald Becker, who implemented most of the network drivers for Linux, greatly contributing to the success of Linux. Donald Becker is also the author of the `skeleton` driver used here.

Some of the methods listed below are not implemented by some drivers (e.g., `example_set_config()` to change system resources at runtime); others are essential, such as `example_hard_start_xmit()` to start a transmission process.

5.3.1 Initializing Network Adapters

Before a network device can be activated, we first have to find the appropriate network adapter; otherwise, it won't be added to the list of registered network devices. The `init()` function of the network driver is responsible for searching for an adapter and initializing its `net_device` structure with patching driver information. Because we search for a network adapter, this function is often called search function.

The argument of the `init()` method is a pointer to the initializing device `dev`. The return value of `init()` is usually `0`, but a negative error code (e.g., `-ENODEV`) when no adapter was found.

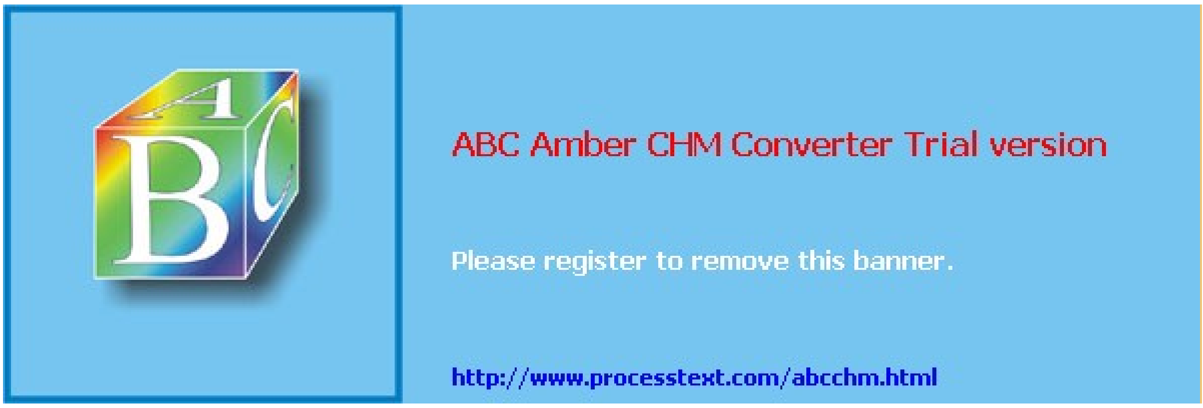
`net_init()/net_probe()`	net/core/dev.c

The tasks of the method `dev->init(dev)` are explained in the source text of our example driver, `isa_skeleton`. There is an example driver in `drivers/net/pci_skeleton.c` for PCI network adapters, but we will not describe it here.

As was mentioned earlier, the main task of the `init()` method is to search for a matching network adapter (i.e., it has to discover the I/O port, especially of the basic address stored in `dev->base_addr`).

We distinguish between two different cases of searching for a network adapter:

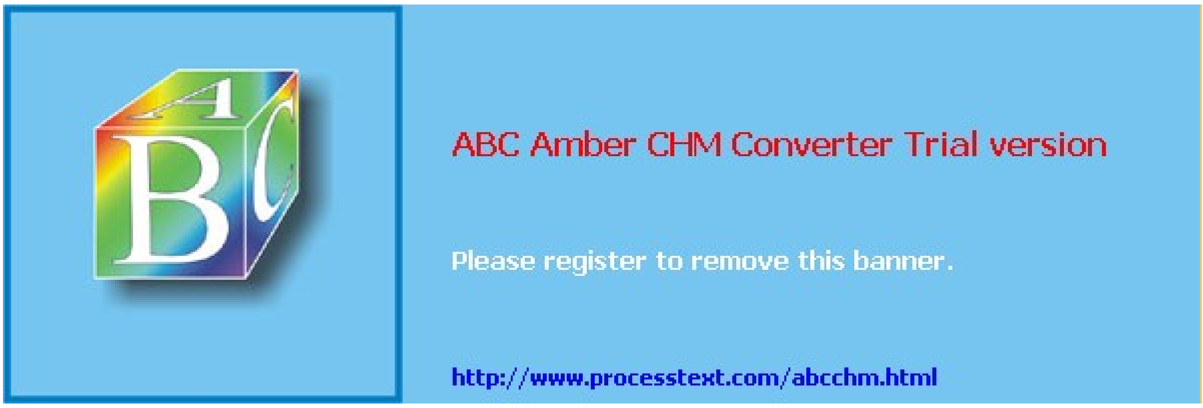
- Specifying the basic address: In this case, the previously created `net_device` structure of the network device is passed as parameter to the `init()` method. The caller can use this structure to specify a basic address for I/O ports in advance. When no matching adapter is



◂ Previous Next ▸

Part III: Layer I + II? Medium Access and Logical Link Layer

◂ Previous Next ▸



◀ Previous Next ▶

Chapter 6. Introduction to the Data-Link Layer

In the following chapters, we will leave the hardware area and move on to the world of network protocols. Chapters 7 through 24 discuss the structure and implementation of network protocols in the Linux kernel.

The previous chapters introduced the most important basics of the Linux network architecture, including the general structure of communication systems and protocol instances (Chapter 3), representation of network packets in the Linux kernel (socket buffers, Chapter 4), and the abstraction of physical and logical network adapters (network devices, Chapter 5). Before we continue discussing the structure and implementation of network protocols in detail, this chapter gives a brief introduction to the structuring of the data-link layer, which represents the connecting layer between network devices and higher network protocols. Of primary interest is the background where network protocols run. Another important topic of this chapter is the interplay of different activities (hardware and software interrupts, tasklets) of the Linux network architecture.

The transition between the different activities in the data-link layer (layers 1 and 2 of the OSI model) occurs when packets are sent and received; these processes are described in detail in Sections 6.2.1 and 6.2.2. First, we will describe the path a packet takes from its arrival in a network adapter until it is handled by a protocol instance in the network layer; then we will describe how a packet is sent from the network layer until it is forwarded to the network adapter.

◀ Previous Next ▶

◂ Previous Next ▸

6.1 Structure of the Data-Link Layer

Chapter 3 introduced two reference models where the lower layers up to the network layer were structured in a different way. In the Internet reference model (TCP/IP model) there is only the data-link layer with the network adapter, and no other instance underneath the Internet protocol (network layer). In the ISO/OSI basic reference model, there are two different layers (physical layer and data-link layer), where the data-link layer is expanded by the media-access layer (Layer 2a) when using local area networks.

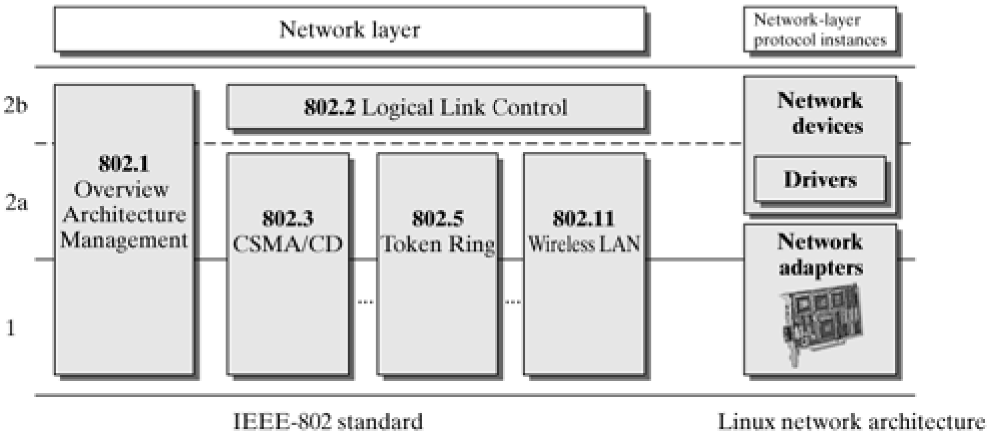
This book deals mainly with the protocols of the Internet world, and one assumes that the Internet reference model would best describe the structure of the Linux network architecture. Interestingly, the classification of the ISO/OSI reference model matches the structure of communication systems in local area networks much better. When taking a closer look at the IEEE 802 standards for local area networks, which are actually always used in the Internet, and their implementation in the Linux kernel, we can clearly recognize the structuring of the ISO/OSI model.

For this reason, the following discussion assumes a structuring as shown in Figure 6-1:

- The OSI layers 1 (physical layer) and 2a (media-access control layer ?MAC) are implemented in network adapters.
- The logical-link control (LLC) layer is implemented in the operating system kernel; network adapters are connected to the operating system kernel by the network devices described in Chapter 5.

Figure 6-1. Standardization of layers 1 and 2 in IEEE 802 and their implementation in the Linux network architecture.

[View full size image]



6.1.1 IEEE Standard for Local Area Networks (LANs)

With its IEEE 802.x standards, the IEEE (Institute of Electrical and Electronics Engineers) found a very extensive proliferation for local area networks (LANs). The best known LAN technologies are 802.3 (CSMA/CD), 802.5 (Token Ring), and 802.11 (wireless LANs). Figure 6-1 gives a rough overview of the 802.x standards and classifies them within the ISO/OSI layer model. As mentioned above, the data-link layer is divided into a logical-link control (LLC) and a media-access control (MAC) layer for networks with jointly used media. The LLC layer hides all media-specific differences and should provide a uniform interface for protocols to the higher layers; the MAC layer reflects the differences between different transmission technologies.

To hide the characteristics of the underlying transmission technology, the LLC layer should offer three services, regardless of this technology:

- Unreliable datagram service (LLC type 1): This very simple service offers no flow control or error control, so it doesn't even guarantee that data is transmitted. The removal of errors is left to the protocols of the higher layers.
- Connection-oriented service (LLC type 2): This service establishes a logical connection

◀ Previous Next ▶

6.2 Processes on the Data-Link Layer

As was mentioned in the beginning of this chapter, the data-link layer forms the connecting layer between drivers or network devices and the higher world of protocol instances. This section gives an overview of the processes on the data-link layer. We will explain what activity forms play an important role on this layer and how the transition between them occurs. Section 6.2.1 describes the process involved when a packet arrives, and Section 6.2.2 discusses how a packet is sent. First, however, we introduce the activity forms and their tasks in the Linux network architecture.

Figure 6-2 gives an overview of the activity forms in the Linux network architecture. As compared with earlier kernel versions, Version 2.4 and up introduced significant performance improvements. Mainly, the use of software interrupts, as compared with the low-performing bottom halves, means a clear performance increase in multiprocessor systems. As shown in Figure 6-2, we can distinguish between the following activities:

- Hardware interrupts accept incoming data packets from the network adapters and introduce them to the Linux network architecture (per Chapter 5). To ensure that the interrupt can terminate as quickly as possible (see Section 2.2.2), incoming data packets are put immediately into the incoming queue of the processing CPU, and the hardware interrupt is terminated. The software interrupt `NET_RX_SOFTIRQ` is marked for execution to handle these packets further.

- The software interrupt `NET_RX_SOFTIRQ` (for short, `NET_RX soft-IRQ`) assumes subsequent (not time-critical) handling of incoming data packets. This includes mainly the entire handling of protocol instances on layers 2 through 4 (for packets to be delivered locally) or on layers 2 and 3 (for packets to be forwarded). This means that most of the protocol instances introduced in Chapters 7 through 25 run in the context of `NET_RX` soft-IRQ.

  Packets incoming for an application are handled by `NET_RX` soft-IRQ upto the kernel boundary and then forwarded to the waiting process. At this point, the packet leaves the kernel domain.

  Packets to be forwarded are put into the outgoing queue of a network device over the layer-3 protocol used (or by the bridge implementation). If the `NET_RX` soft-IRQ has not yet used more than one tick ($1/H_z$) to handle network protocols, then it tries immediately to send the next packet. If the soft-IRQ was able to send a packet successfully, it will handle it to the point where it is passed to the network adapter. (See Chapter 5 and Section 6.2.2.)

- The software interrupt `NET_TX_SOFTIRQ` (for short, `NET_TX` soft-IRQ) also sends data packets, but only provided that it was marked explicitly for this task. This case, among others, occurs when a packet cannot be sent immediately after it was put in the output queue? for example, because it has to be delayed for traffic shaping. In such a case, a timer is responsible for marking the `NET_RX` soft-IRQ for execution at the target transmission time (see Section 6.2.2) and transmitting the packet.

  This means that the `NET_TX` soft-IRQ can transmit packets in parallel with other activities in the kernel. It primarily assumes the transmission of packets that had to be delayed.

- Data packets to be sent by application processes are handled by system calls in the kernel. In the context of a system call, a packet is handled by the corresponding protocol instances until it is put into one of the output queues of the sending network device. As with `NET_RX` soft-IRQ, this activity tries to pass the next packet to the network adapter immediately after the previous one.

- Other activities of the kernel (tasklets, timer handling routines, etc.) do various tasks in the Linux network architecture. However, unlike the tasks of the activities described so far, they cannot be clearly classified, because they are activated by other activities upon demand. In general, these activity forms run tasks at a specific time (timer handling routines) or at a less specified, later time (tasklets).

- Application processes are not activities in the operating-system kernel. Nevertheless, we mentioned them here within the interplay of activities of the kernel, because some are started

6.3 Managing Layer-3 Protocols

The previous section of this chapter described the path of a packet between a network adapter and the interface to higher protocol instances. This section discusses this interface in more detail. First, we will explain how new protocols can be added. Because only protocols of the network layer (IP, ARP, IPv6, IPX) are added to the Linux network architecture over this interface, it is also referred to as the interface to the network layer or layer-3 protocols in the following discussion.

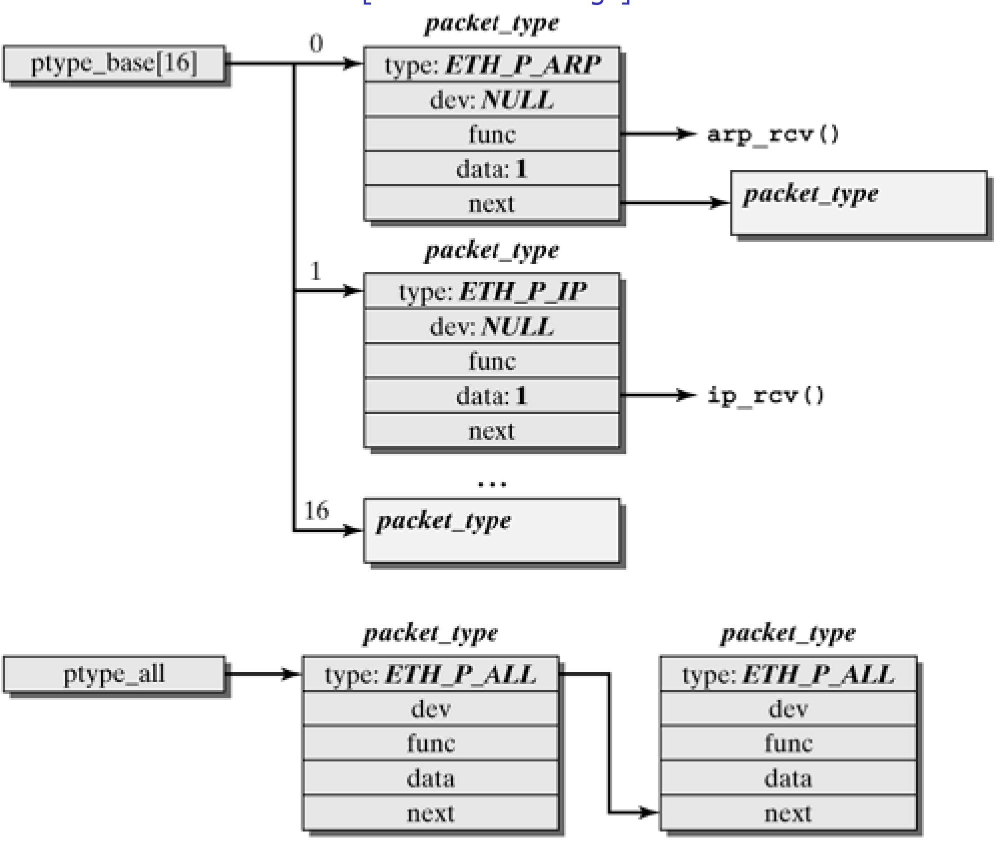
In the Linux kernel, we distinguish between two types of layer-3 protocols, where the first type is used mostly for analysis purposes:

- A protocol receives all packets arriving at the interface to the layer-3 protocols.
- A protocol receives only packets with the correct protocol identifier (e.g., 0x0800 for the Internet Protocol).

Figure 6-5 shows that these two types of protocols are managed in two different data structures. We can see in this figure that the two types of layer-3 protocols do not differ much. Both types are managed in a structure of the type `packet_type` and linked in different lists, depending on the above-mentioned type. The simple linked list, `ptype_all`, stores the protocols that should receive all incoming socket buffers. The hash table, `ptype_base`, manages all normal layer-3 protocols.

Figure 6-5. Managing protocols above network devices.

[View full size image]



A `packet_type` structure is created and placed into the corresponding data structure for each protocol. The following parameters are required in the `packet_type` structure to define a protocol:

- `type:` This field specifies the protocol identifier (i.e., the constants listed in Figure 6-6). If `ETH_P_ALL` is stated in this field, then the protocol is added to the list `ptype_all` when it is registered, and it receives all packets. Otherwise, it receives only packets with protocol identifier type.

  The identifier of a protocol has to be extracted from the packet data in the receive routine of

◀ Previous Next ▶

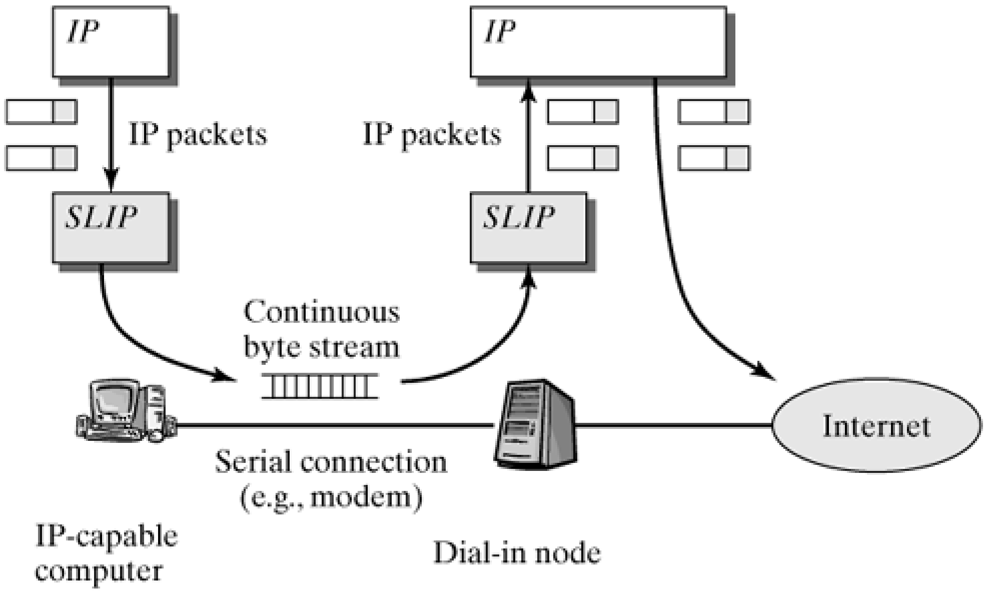
Chapter 7. The Serial-Line Internet Protocol (SLIP)

◀ Previous Next ▶

7.1 Introduction

The packet-oriented IP protocol is used to communicate over the Internet. However, a modem can transmit only a continuous byte stream. For this reason, to establish a connection from your local PC over an analog telephone line to the worldwide Internet, we need a protocol that encapsulates network packets so that they can be transmitted over a modem connection between a local computer and a point of presence (PoP). The two endpoints of the modem connection can then communicate over IP. The point of presence itself is directly connected to the Internet and routs IP packets between the local PC and the Internet. (See Figure 7-1.)

Figure 7-1. Scenario for the use of SLIP.



Another possible use of such a protocol is for the IP communication of two computers over the serial V.24 interface, which is available in most PCs. This use lets you build an IP network at little cost (and very low speed) without the need to install additional interfaces, such as Ethernet cards.

RFC 1055 [Romk88] specifies the SLIP (Serial Line IP) for the V.24 task. SLIP represents an intermediate layer within the network architecture: At its upward face, packets are taken from or forwarded to the IP layer; at its downward face, data are sent to or received from a serial interface driver.

As compared with the more recent PPP protocol (see Chapter 8), SLIP is very simple, but offers a rather limited functionality:

- SLIP includes no mechanisms for establishment of a controlled connection: As soon as SLIP has been started on both ends, the connection is implicitly established. For this reason, no parameters, such as IP address, DNS information, or the SLIP operating mode used, can be negotiated. These parameters have to be set manually or by use of a script before SLIP is started.
- SLIP serves exclusively for the transmission of Version-4 IP packets. Other network protocols (e.g., IP version 6 or X.25) are not supported.
- SLIP has no mechanisms to detect or correct errors; these functions have to be handled by higher network layers.
- In contrast to PPP, the payload in transmitted IP packets cannot be compressed. The CSLIP operating mode (described in the next bullet) allows you to compress the IP packet headers only.

In addition to the standard operating mode, SLIP supports the following modes:

◂ Previous Next ▸

7.2 Slip Implementation in the Linux Kernel

Before we describe the SLIP implementation in the Linux kernel, we will first discuss the concept of TTY devices and TTY line disciplines, the better to illustrate how SLIP is implemented in Linux. Subsequently, this section will give an overview of the most important functions of the SLIP implementation before we describe the steps involved in implementing SLIP in detail.

7.2.1 TTY Devices and Line Disciplines

In Linux, all devices that can act as terminals are grouped under the collective term TTY (Teletype or Terminal Type). Table 7-2 shows several examples. A TTY device is a character device offering special functions to control a terminal. This includes, for example, the flag for whether the terminal should produce an echo and commands to position the cursor and change color.

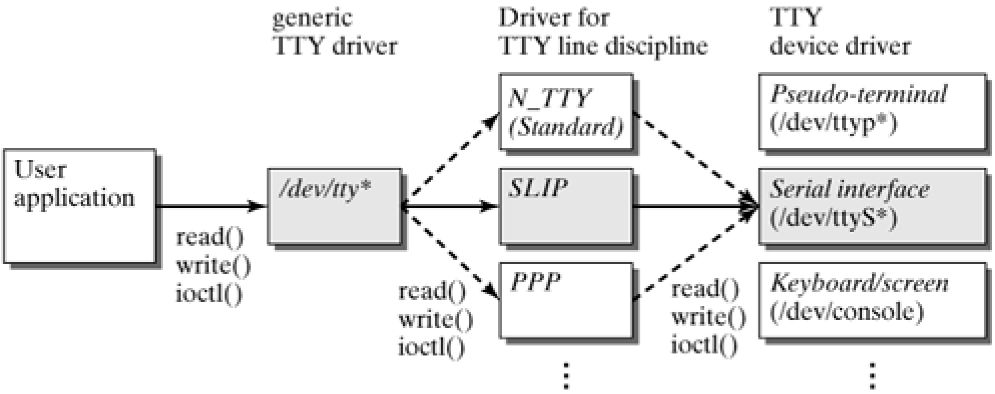
Table 7-2. Examples of TTY devices.

/dev Entry	Meaning
tty0? tty7	virtual consoles
pty#	pseudo-terminals? e.g., xterm window
ttyS#	serial interfaces

A TTY device can generally switch between different TTY line disciplines. This means that, in the Linux kernel, each system call to read (`read()`), write (`write()`), or control (`ioctl()`) invokes a routine specific to this line discipline. More specifically, the implementation of a TTY line discipline is inserted between the TTY device driver (low-level driver), which is in charge for the actual input and output, and the user process that wants to access the TTY device. (See Figure 7-3.)

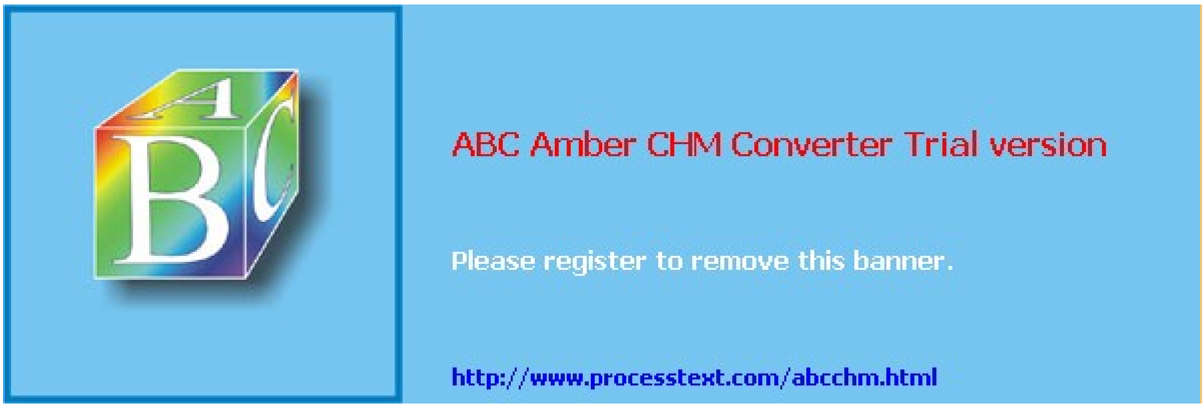
Figure 7-3. Interplay between TTY line disciplines and TTY device drivers.

[View full size image]



One possible use for a TTY line discipline is the automatic conversion of all line ends between UNIX and Windows computers (LF versus CR/LF). In addition, TTY line disciplines offer an elegant means whereas serial interfaces can intercept and change all data transmitted over a serial interface without the need for the TTY line discipline driver to open and close the serial interface or to establish a modem dialup connection.

To register a new TTY line discipline with the Linux kernel, the driver has to first create a `tty_ldisc` structure (declared in `<include/linux/tty_ldisc.h>`) and set the function pointers contained in it. Subsequently, this data structure is registered by `tty_register_ldisc()` with the kernel to make the new TTY line discipline available for user programs.



◀ Previous Next ▶

Chapter 8. The Point-to-Point Protocol (PPP)

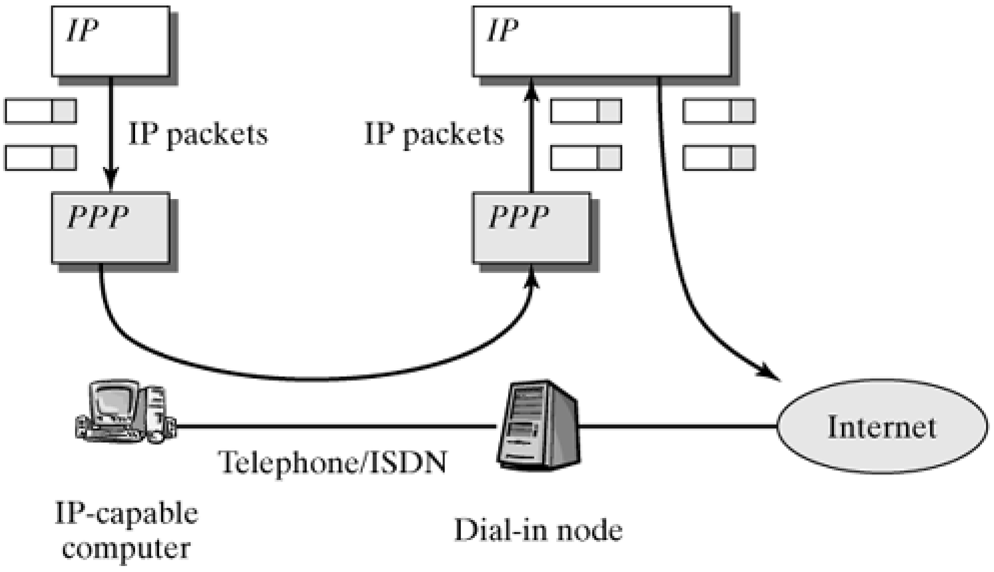
◀ Previous Next ▶

◀ Previous Next ▶

8.1 Introduction

The Point-to-Point protocol (PPP) can be used by two computers connected directly (i.e., not over a local area network) to communicate. PPP is defined in RFC 1661 [Simp94a]. A typical application for PPP is dialing into the Internet over a modem; see Figure 8-1. In this case, it increasingly replaces the older SLIP protocol (see Chapter 7), which has proven to be not as flexible as modern applications demand.

Figure 8-1. Scenario for the use of PPP.



In contrast to SLIP, PPP is multiprotocol enabled. In addition to IPv4, IPv6, and a large number of other network protocols, PPP also supports several subprotocols, which handle authentication and configuration tasks (e.g., negotiating important connection parameters and allocating dynamic IP addresses).

The architecture of PPP is basically designed for peer-to-peer communication. Nevertheless, in the case of a dialup connection to the Internet, the point of presence is often called server and the dialing computer is called client. Though the protocol allows both ends of a connection to expect that the peer authenticate itself and allocates it a dynamic IP address, this would naturally not make much sense when dialing into the Internet.

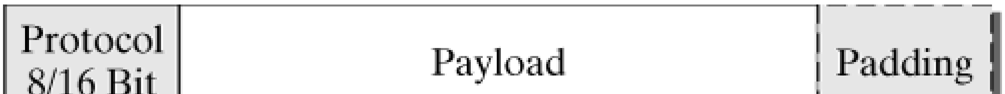
Linux distinguishes between synchronous and asynchronous PPP, depending on whether the underlying TTY device supports packet-oriented data transmission (synchronous? for example, in ISDN with HDLC as the layer-2 protocol) or it works with a continuous byte stream (asynchronous? e.g., in a modem connection).

We will discuss the asynchronous transmission over a serial interface in more detail later, because it requires more protocol functionality than synchronous PPP. The ISDN subsystem of Linux has its own, independent PPP implementation, which is not discussed here.

8.1.1 Subprotocols

Figure 8-2 shows the structure of a (synchronous) PPP packet. Synchronous PPP always processes entire frames of the lower-layer protocol, which is the reason why it is not necessary to specify the length. Asynchronous PPP additionally requires a frame detection, similar to SLIP. (See Section 7.1.1 .) Section 8.3.5 describes how this frame detection is implemented in PPP.

Figure 8-2. Structure of a PPP packet.



8.2 PPP Configuration in Linux

PPP drivers in the Linux kernel have comparatively few configuration options. The reason is that most settings were moved to `pppd`, which means that they can be changed at runtime or even set separately for different devices.

It is important to note that PPP over ISDN requires different settings, which have nothing to do with the settings discussed in this section, because the ISDN subsystem includes its full PPP implementation. This applies particularly to the kernel options, but also to the `pppd` configuration. To be able to use PPP over ISDN, for example, it is not necessary to activate the "normal" PPP in the configuration of the Linux kernel.

8.2.1 Kernel Options

The Linux kernel version 2.2 included only one option that could be used to enable or disable the full PPP support. Version 2.3 introduced three additional setting options (shown in Table 8-2).

Table 8-2. PPP driver options in the Linux kernel.

Option	Meaning
`CONFIG_PPP`	Activates the generic PPP.
`CONFIG_PPP_ASYNC`	Activates the asynchronous PPP.
`CONFIG_PPP_DEFLATE`	Supports payload compression.
`CONFIG_PPP_BSDCOMP`	Supports alternative payload compression.

The payload compression by the deflate option is preferred over the BSD compression algorithm, because it is free from patents and more effective. By the way, the deflate algorithm is also used in `gzip`.

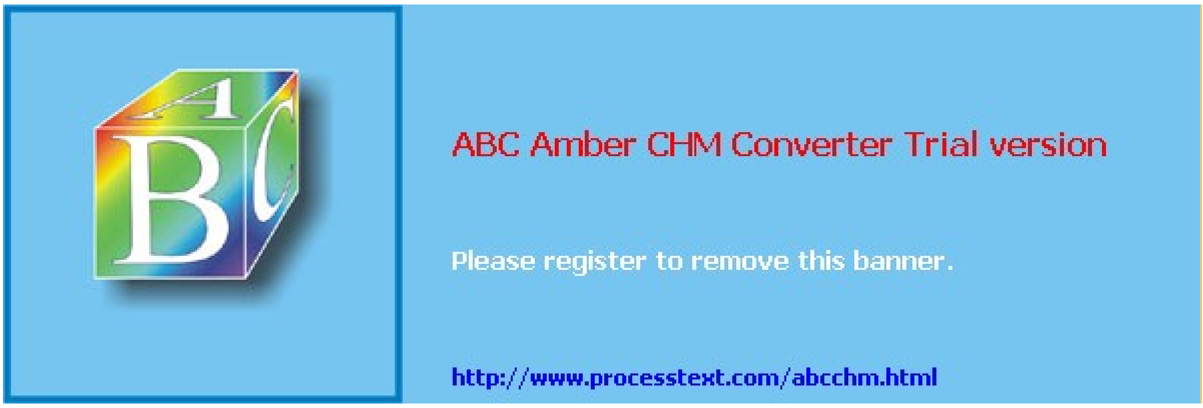
8.2.2 pppd? the PPP Daemon

As was mentioned before, most settings are effected by `pppd`. The configuration files required for these settings are normally stored in the directory `/etc/ppp/`. See also the manpage of `pppd`, Files section, for details.

Upon startup, `pppd` reads first the general configuration file `options` and then a device-specific configuration file (e.g., `options.ppp0`). In addition, there is a possibility of adding user-specific settings in `$HOME/.ppprc`. These files include information about the serial interface to be used, about whether configuration requests of the peer should be accepted, and about which user name will be used to log into the peer. The following represent some important entries in the configuration file; however, they do not represent a full configuration:

```
# Options for pppd over a serial line
# /etc/ppp/options
modem                      # use the modem control lines
crtscts                    # use hardware flow control
lock                       # create lockfile to ensure exclusive access
defaultroute               # set default route to this interface
debug                      # enable connection debugging facilities
user egon
```

The user name in the last line serves as key for the entry in the pap-secrets and chap-secrets files, which include the passport of each user in a PAP or CHAP authentication. Both files have the same



8.3 PPP Implementation in the Linux Kernel

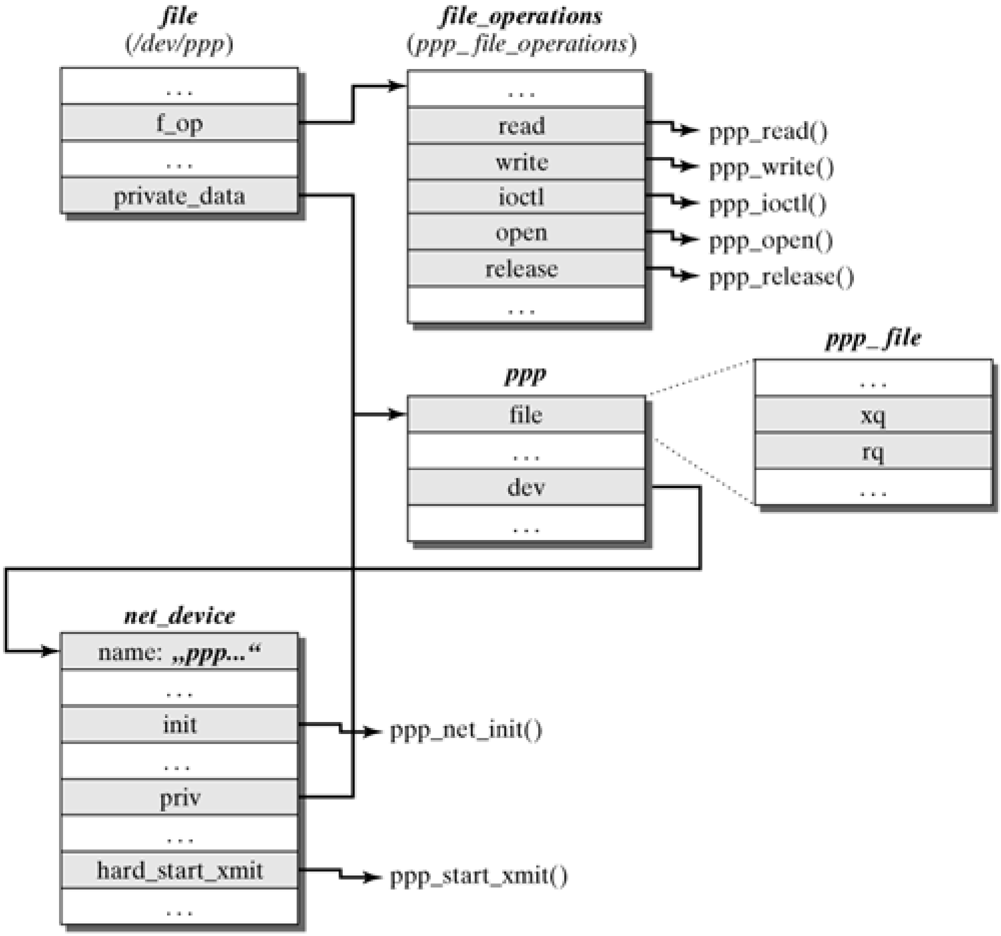
As mentioned before, the PPP implementation in Linux is divided into four different tasks: three kernel modules and the `pppd` user space daemon. During design of this division, care was taken to move as little functionality as possible into the Linux kernel. For this reason, the kernel modules are rather simple. `pppd` includes 13,000 lines of code (2,100 lines alone in `main.c`), which means that it is four times the size of the three kernel modules (`ppp_generic.c, ppp_synctty.c,` and `ppp_async.c`) together. In the following sections, we will first discuss the generic PPP driver and then the driver for the asynchronous PPP TTY line discipline. The driver for the synchronous PPP line discipline is relatively simple, so we will not discuss it here.

8.3.1 Functions and Data Structures of the Generic PPP Driver

Figure 8-4 shows the most important data structures of the generic PPP driver. There is a separate `ppp` structure with general management information for each PPP device. Some important entries, particularly the transmit and receive queues, `xq` and `rq`, are in a substructure of the type `ppp_file`. This substructure is also found in the channel structure, which is used to manage single channels in multilink PPP, which will not be discussed here, for the sake of simplicity.

Figure 8-4. Important data structures of the generic PPP driver.

[View full size image]



There is a PPP device for each network device, the `net_device` structure of which refers to the related `ppp` structure in the field `priv`. In addition, the PPP daemon can send and receive control packets of subprotocols (see Section 8.1.1) over the device `/dev/ppp`. For this purpose, it must first bind the device `/dev/ppp` to a specific PPP device by use of an `ioctl()` call. This binding means that a pointer to the ppp structure is entered into the field `private_data` of the relevant file structure.

`ppp_init()` drivers/net/ppp_generic.c

Previous Next

8.4 Implementing the PPP Daemon

As was mentioned repeatedly in previous sections, the largest part of the implementation effort takes place in the PPP daemon, pppd. One of the reasons is that it processes all subprotocols to control the PPP connection. To maintain expandability, utmost care was taken to keep the implementation highly modular, and it has a clearly defined interface for subprotocol implementations.

8.4.1 Managing Subprotocols

```
struct protent  pppd/pppd.h
```

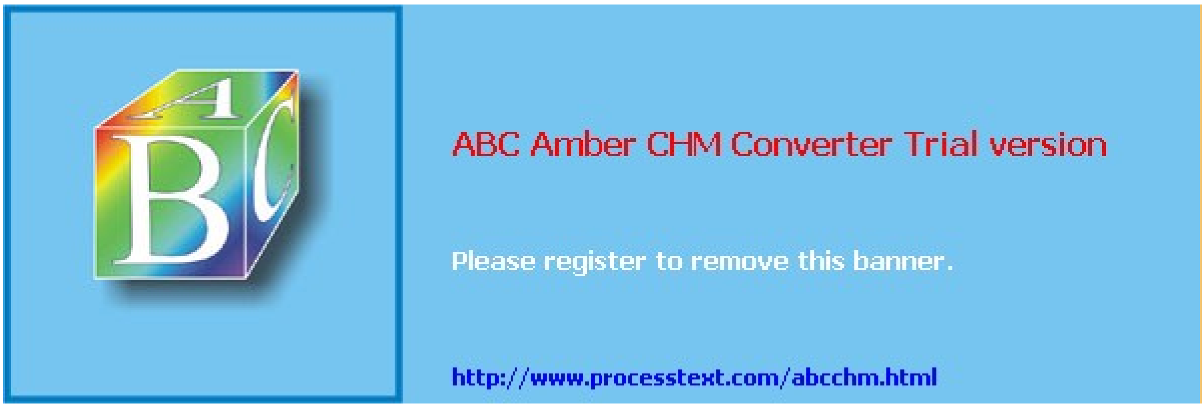
The core of the pppd interface for subprotocols is the protent structure, which is defined in the file pppd/pppd.h. It includes mainly entries for callback functions, which are always called whenever pppd receives a packet that it allocates to this subprotocol, given the protocol ID:

```
struct protent {
    u_short protocol;           /* PPP protocol number */
    /* Initialization procedure */
    void (*init) ___P((int unit));
    /* Process a received packet */
    void (*input) ___P((int unit, u_char *pkt, int len));
    /* Process a received protocol-reject */
    void (*protrej) ___P((int unit));
    /* Lower layer has come up  */
    void (*lowerup) ___P((int unit));
    /* Lower layer has gone down */
    void (*lowerdown) ___P((int unit));
    /* Open the protocol */
    void (*open) ___P((int unit));
    /* Close the protocol */
    void (*close) ___P((int unit, char *reason));
    /* Print a packet in readable form */
    int (*printpkt) ___P((u_char *pkt, int len,
                          void (*printer) ___P((void *, char *, ...)),
                          void *arg));
    /* Process a received data packet */
    void (*datainput) ___P((int unit, u_char *pkt, int len));
    bool enabled_flag;    /* 0 iff protocol is disabled */
    char *name;           /* Text name of protocol */
    char *data_name;      /* Text name of corresponding data protocol */
    option_t *options;    /* List of command-line options */
    /* Check requested options, assign defaults */
    void (*check_options) ___P((void));
    /* Configure interface for demand-dial */
    int (*demand_conf) ___P((int unit));
    /* Say whether to bring up link for this pkt */
    int (*active_pkt) ___P((u_char *pkt, int len));
};
```

Each of the protocols known to pppd has exactly one entry in the global list struct protent protocols[].

Figure 8-6 shows a flow diagram representing a simplified procedure of how a connection is established. The function init() is executed immediately after pppd has started. Shortly after that, the function check_options() is run to handle settings, if applicable, using command-line arguments or options in /etc/ppp/options.

Figure 8-6. Procedure involved when pppd establishes a connection.



◂ Previous Next ▸

Chapter 9. PPP over Ethernet

◂ Previous Next ▸

9.1 Introduction

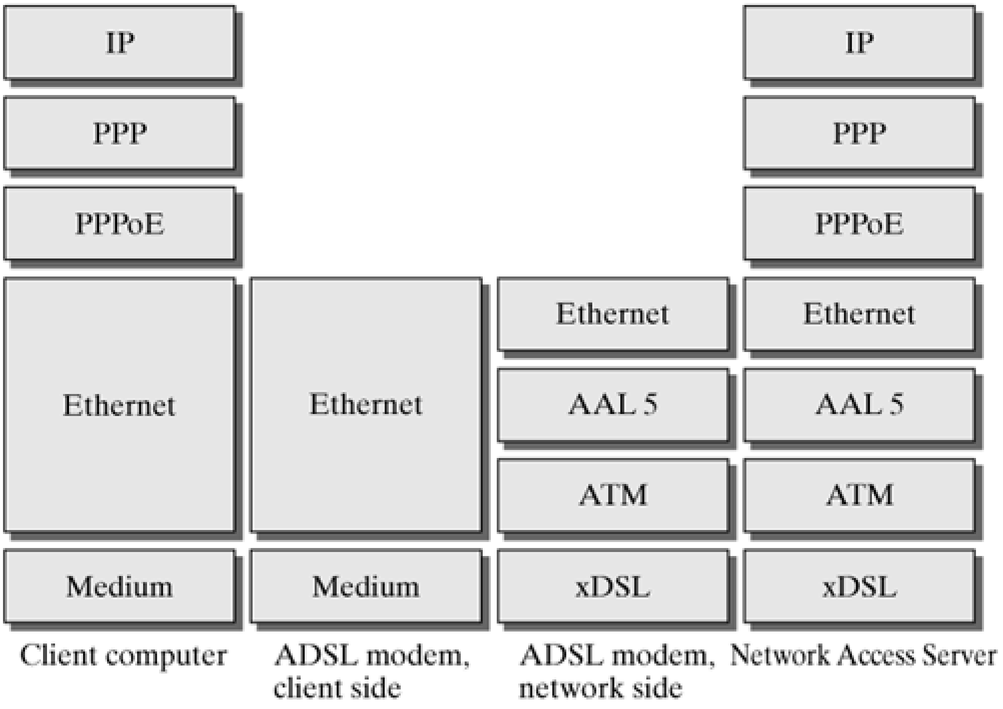
Chapter 8 introduced the Point-to-Point protocol (PPP). Today, it is most frequently used in access networks that use ADSL as the access technology.

The ADSL (Asymmetric Digital Subscriber Line) access technology offers high-speed Internet access for private or commercial customers. From the technical viewpoint, this is a dedicated line (i.e., a permanent connection). Dedicated lines are normally billed on the basis of transmission volumes. In contrast, private Internet links are billed on a time basis. To enable ADSL to support time-specific billing as well, a new protocol, PPPoE, was developed. PPPoE is based on two accepted standards? PPP and Ethernet.

More specifically, an ADSL modem (NTBBA? Network Termination Point Broad-Band Access), installed behind a so-called splitter, is connected to the computer over Ethernet. This means that the computer has to be equipped with an Ethernet network card. This dedicated Ethernet line between the PC of the home user and the dialup computer of the access network operator is used to establish a PPP connection, which allows the access network operator to identify the user and bill for the usage time between the PPP dialup and the termination of that PPP session. This PPP connection can be used to exchange IP packets.

Figure 9-1 shows the resulting protocol stack. This chapter first introduces the PPPoE (PPP over Ethernet) protocol described in [MLEC+99]. Then, it introduces the implementation in the user space, which is used in kernel Versions 2.2 and 2.3. Finally, this chapter discusses the implementation in the kernel from kernel version 2.4 and up.

Figure 9-1. Protocol stack for the use of PPP over Ethernet.



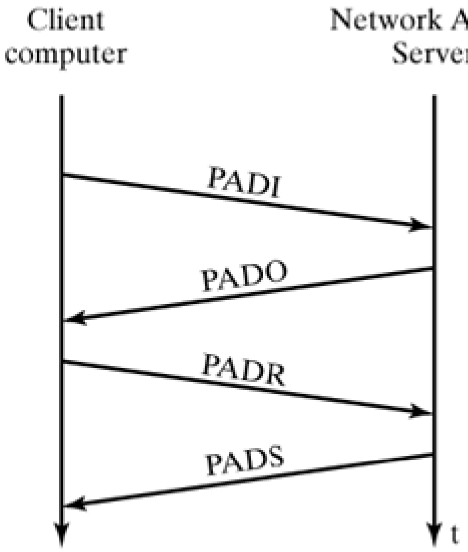
◀ Previous Next ▶

9.2 PPPOE Specification in RFC 2516

To be able to transport PPP protocol units over Ethernet, they are inserted as payload in Ethernet frames. For this purpose, two new ethertype values were defined, which show the receiver that the Ethernet frame contains PPP payload.

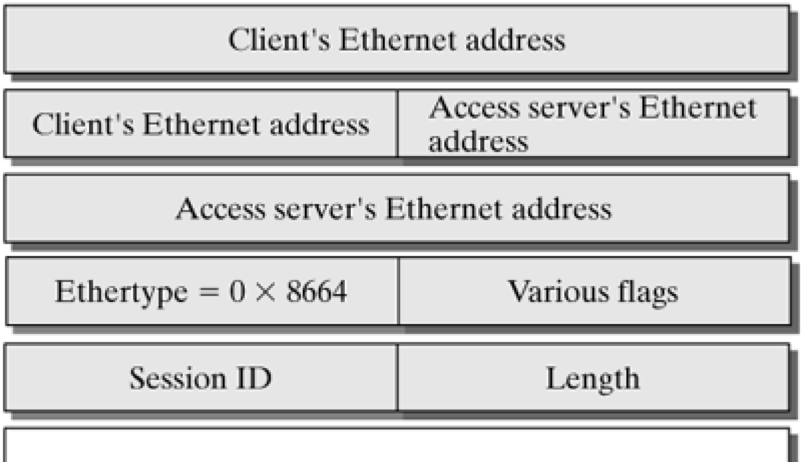
The two different types serve to distinguish between two phases within PPoE: the discovery stage, and the session stage. A typical discovery stage consists of four steps, which appear as follows (in Figure 9-2): The host sends a PADI (PPPoE Active Discovery Initiation) packet to the Ethernet broadcast address to find out which access concentrators are available in the Ethernet. One (or several) of these access concentrators replies by sending a PADO (PPPoE Active Discovery Offer) packet, informing the host about the Ethernet address where an access concentrator is available, which may specify additional services. The host selects one from the available access concentrators and requests that this concentrator establish a connection by sending a PADR (PPPoE Active Discovery Request) packet. The access concentrator replies by sending a PADS (PPPoE Active Discovery Session Confirmation) packet.

Figure 9-2. Typical sequence for PPPoE Active Discovery.



Subsequently, the discovery stage is left behind and the session stage begins, where PPP payload is packed transparently in Ethernet frames having ethertype value `0x8864` (in contrast to packets in the discovery stage, which have ethertype value `0x8863`). Figure 9-3 shows what a PPPoE packet looks like in the session stage. The underlying Ethernet already forms frames, so PPPoE does not require character stuffing, in contrast to the asynchronous case described in Chapter 8.

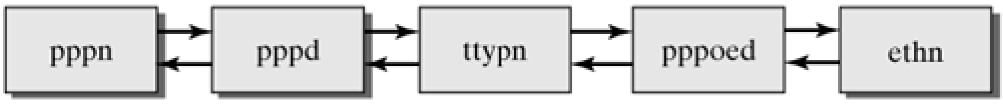
Figure 9-3. Protocol data unit of the PPPoE session stage.


Client's Ethernet address
Client's Ethernet address
Access server's Ethernet address
Access server's Ethernet address
Ethertype = 0 × 8664
Various flags
Session ID
Length

9.3 Implementation in the User Space

The kernels of Versions 2.2 and 2.3 do not support PPPoE. Instead, another daemon is started in the user space, in addition to `pppd`. This daemon is called `pppoed`; it processes PPP packets of the Ethernet card and forwards them to `pppd.pppd`, and `pppoed` communicate over a pseudo-terminal, as shown in Figure 9-4.

Figure 9-4. `pppd` and `pppoed` communicate in the user space.



There are various implementations in the user space, including the Roaring Penguin implementation [Roar01], which appears to be the most elaborate. The major drawback of this approach and similar approaches is that the intermediate pseudo-terminal requires an additional transition between the kernel and the user space, which reduces the performance considerably. For this reason, we will consider only the kernel implementation available from kernel Version 2.4 in the following discussion.

◂ Previous Next ▸

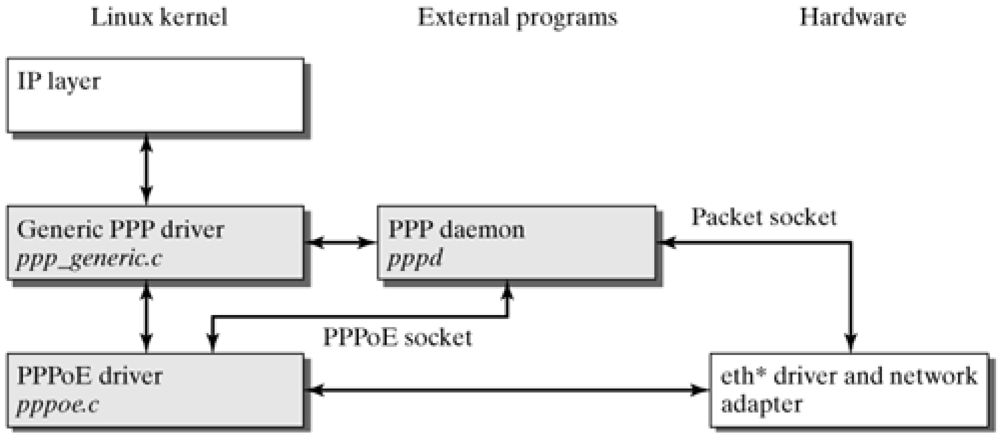
9.4 Implementation in the Linux Kernel

Together with kernel Version 2.4, PPPoE support was integrated in the pppd daemon, and the kernel was expanded by a connection between the generic PPP driver and the Ethernet network card.

Figure 9-5 shows the interaction between these components. The PPPoE driver assumes several functions within the kernel. To the lower layer (i.e., the Ethernet card and the driver software), the PPPoE driver plays the role of a layer-3 protocol. As we will see later in more detail, incoming Ethernet packets are allocated to a protocol matching the type identifier in the Ethernet frame (e.g., the IP protocol or the PPPoE protocol for the ethertype values 0x8863 and 0x8864 mentioned earlier). Towards the higher-layer generic PPP driver, which was described in the previous chapter, the PPPoE driver behaves much as does the asynchronous PPP driver. In contrast to that driver, however, the PPPoE driver does not implement a tty operating mode.

Figure 9-5. Communication between pppd and the PPP and PPPoE drivers.

[View full size image]



To initiate the PPPoE discovery stage of pppd in the user space, it is additionally necessary to have the PPPoE driver and pppd communicate directly. Section 9.4.2 discusses this communication in detail.

9.4.1 Changes to the Kernel

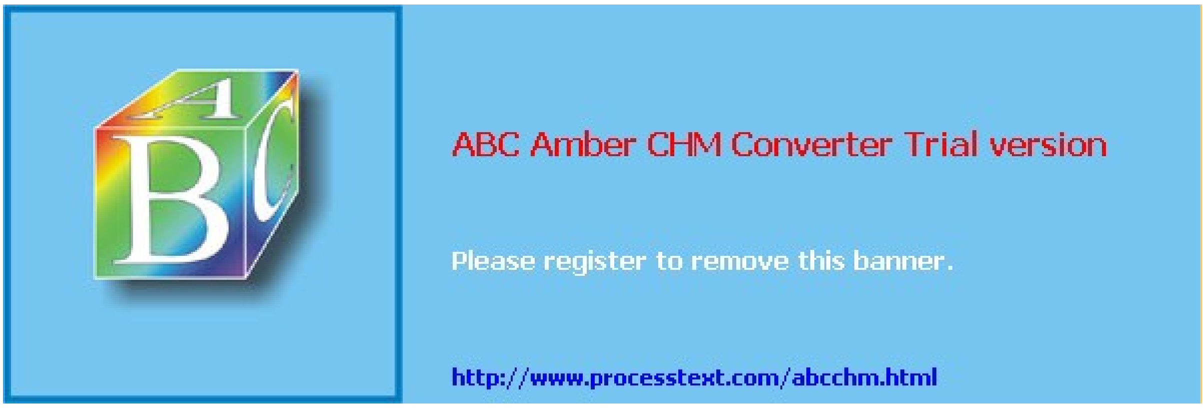
The PPPoE driver, which is included in kernel Version 2.4 and higher in experimental form, consists of the file drivers/net/pppoe.c. In addition, there is a file called drivers/net/pppox.c, which is intended to harmonize present and future PPP implementations in the kernel. General functions that previously were used only by the PPPoE implementation were moved to the file pppox.c, and other PPP implementations over other networks should be available in the future.

Functions and Data Structures of the PPPoE Driver

In the first step, the PPPoE driver registers the PPPoE protocol with the kernel. This can be seen in the following piece of source text:

pppoe_init() drivers/net/pppoe.c

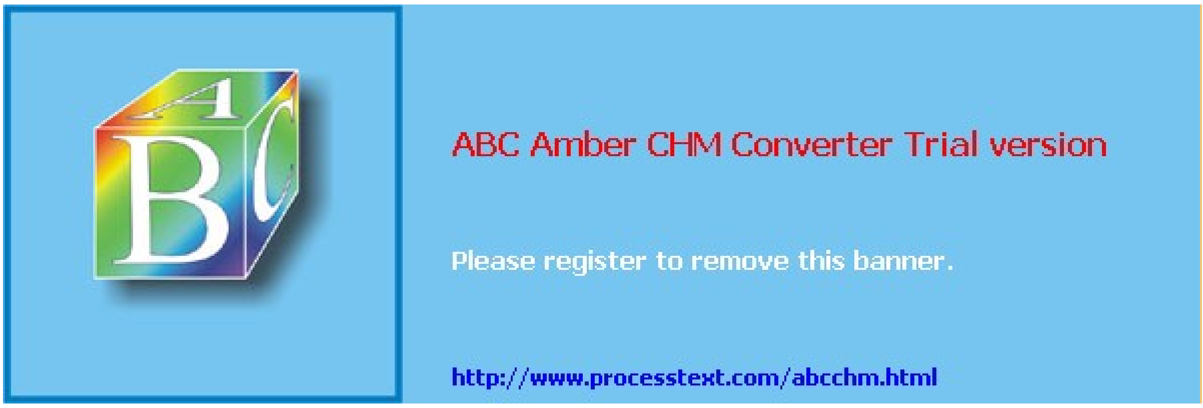
```
{
Use a function from drivers/net/pppox.c to register the PPPoE protocol:
int err = register_pppox_proto(PX_PROTO_OE, &pppoe_proto);
if (err==0) {
        dev_add_pack(&pppoes_ptype);
        /*Add a packet handler
        for incoming packets of type ETH_P_PPP_SES
        (PPPoE session packets), which points to pppoe_rcv */
        dev_add_pack(&pppoed_ptype);
        /*Add a packet handler for incoming packets of type
```



◂ Previous Next ▸

Chapter 10. Asynchronous Transfer Mode? ATM

◂ Previous Next ▸



10.1 Introduction

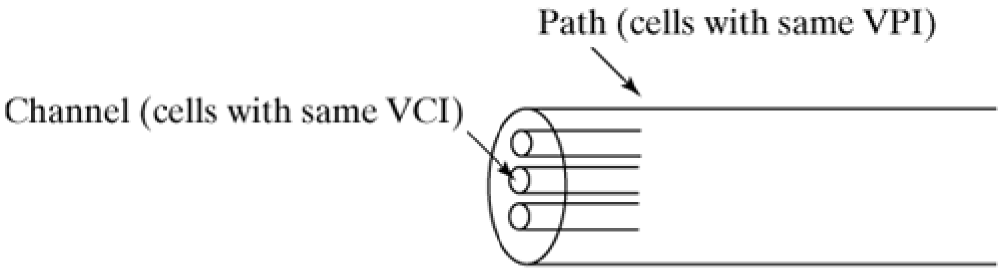
Initially, the Asynchronous Transfer Mode (ATM) was introduced to provide a uniform protocol for the transmission of voice and data, offering guarantees for the required QoS (Quality of Service) p arameters (such as data rate and delay) [McSp95]).

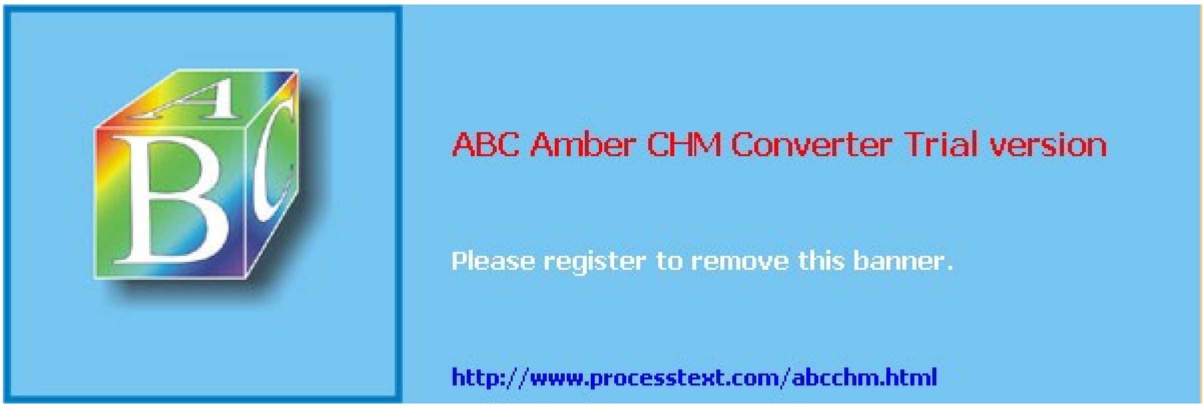
In contrast to initial expectations and forecasts, the ATM network technology has not established itself in end systems, but it is widely used in core networks. First of all, ATM offers a uniform concept to support QoS (Quality of Service) in networks; QoS was attempted much later in IP-based networks.

The ATM network technology is connection-oriented, which means that a connection has to be established before data can be transmitted. There are two types of connections: In a Permanent Virtual Connection (PVC), the connection throughout the network is established by the network management; a network management station extends the forwarding tables within the forwarding nodes between two endpoints of an ATM connection so that the ATM cells created by the endpoints are forwarded to the other endpoint. The second type of ATM connection is a Signaled Virtual Connection (SVC); in this connection type, the connection is established by the communicating end systems, which send connection requests and respond to such requests.

In ATM jargon, packets are called cells. In contrast to IP protocol data units, an ATM cell has a fixed size, 53 bytes: 5 bytes for the packet header, 48 bytes for the payload. The 5-byte packet header includes forwarding information, as for IP frames, which allocates a cell to a connection. The ATM network technology uses a hierarchical connection concept, which distinguishes between paths and channels. Each cell is allocated to exactly one virtual path, and to exactly one virtual channel within that path, as shown in Figure 10-1. This allocation to a path and a channel is specified in two bit fields in the cell header: an 8-bit field for the Virtual Path Identifier (VPI), and a 16-bit field for the Virtual Channel Identifier (VCI).

Figure 10-1. Virtual paths and channels in the ATM network technology.





Previous Next

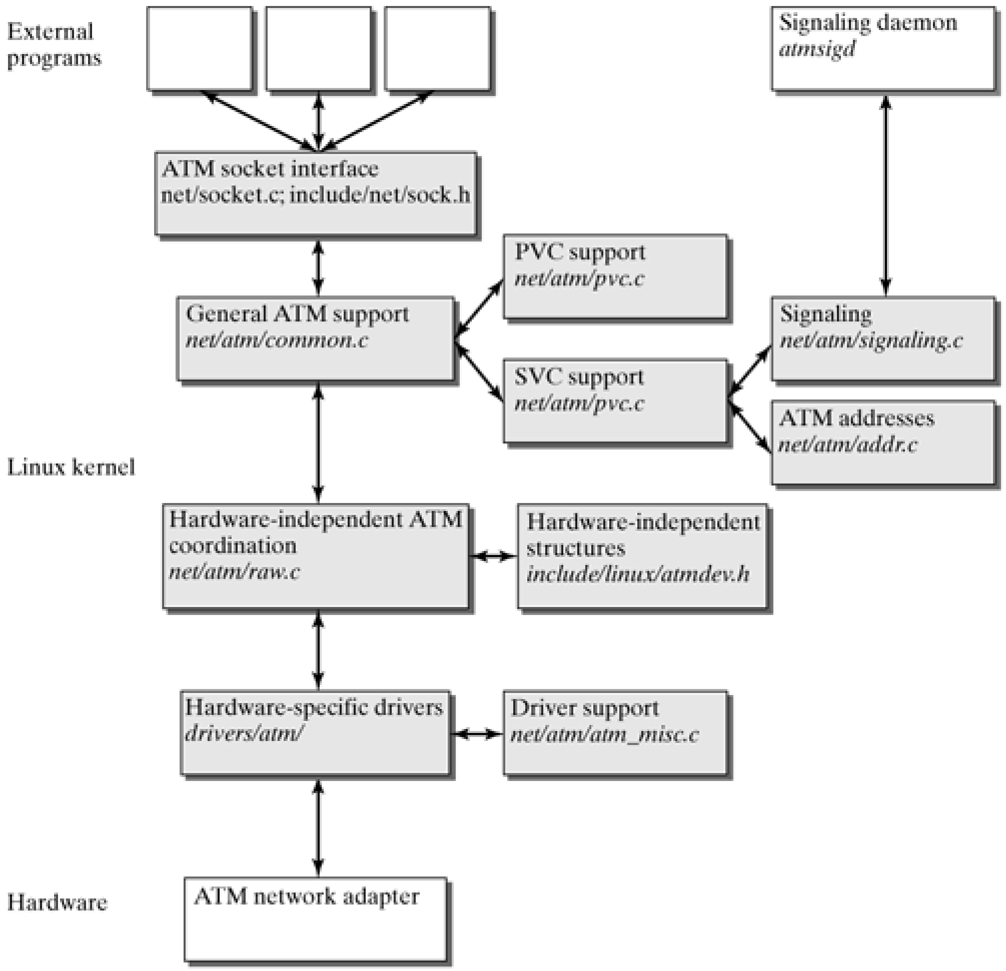
10.2 Implementing ATM in Linux

Figure 10-2 shows how the ATM support is structured in the Linux kernel. This implementation comprises two major parts:

- Extension of the socket interface to support the ATM protocol. We will not further discuss this part in this chapter, because the socket interface will be described in detail in Chapters 26 and 27.
- General ATM support within the operating system kernel. Various additional functions are available, depending on whether a connection is a permanent or a signaled virtual connection. This part will be described in detail below.
- Support of various ATM network cards. Again, this is divided into a general part, which is independent of the type of hardware used, and a part that includes the driver for the respective network card and some support functions.

Figure 10-2. Structure of the ATM support in the Linux kernel.

[View full size image]



The following sections begin with a description of the data transmission over a permanent virtual channel (PVC). Subsequently, we describe how the signaled virtual channel (SVC) is supported in the Linux kernel.

10.2.1 Permanent Virtual Channels

An application accesses a permanent virtual channel (PVC) over a socket. A PVC socket can take any of four states; `closed`, `created`, `connected`, and `connecting` (as shown in Figure 10-3).

Figure 10-3. State transition diagram when opening a socket for a permanent virtual channel

◀ Previous Next ▶

10.3 Configuration

ATM support has been part of the Linux kernel since kernel Version 2.4 and requires no additional patches; however, configuring of the kernel requires the entry `"Prompt for development and/or incomplete driver"` to be activated to provide selection of the desired ATM support.

The signaling daemon described above is not part of the Linux kernel; it has to be installed additionally in the user space. It can be downloaded from [BlAl01], where you will also find other utilities that complete the ATM support in Linux. The current development of the ATM support for Linux can be followed up from the mailing list available at [BlAl01].

◀ Previous Next ▶

◀ Previous Next ▶

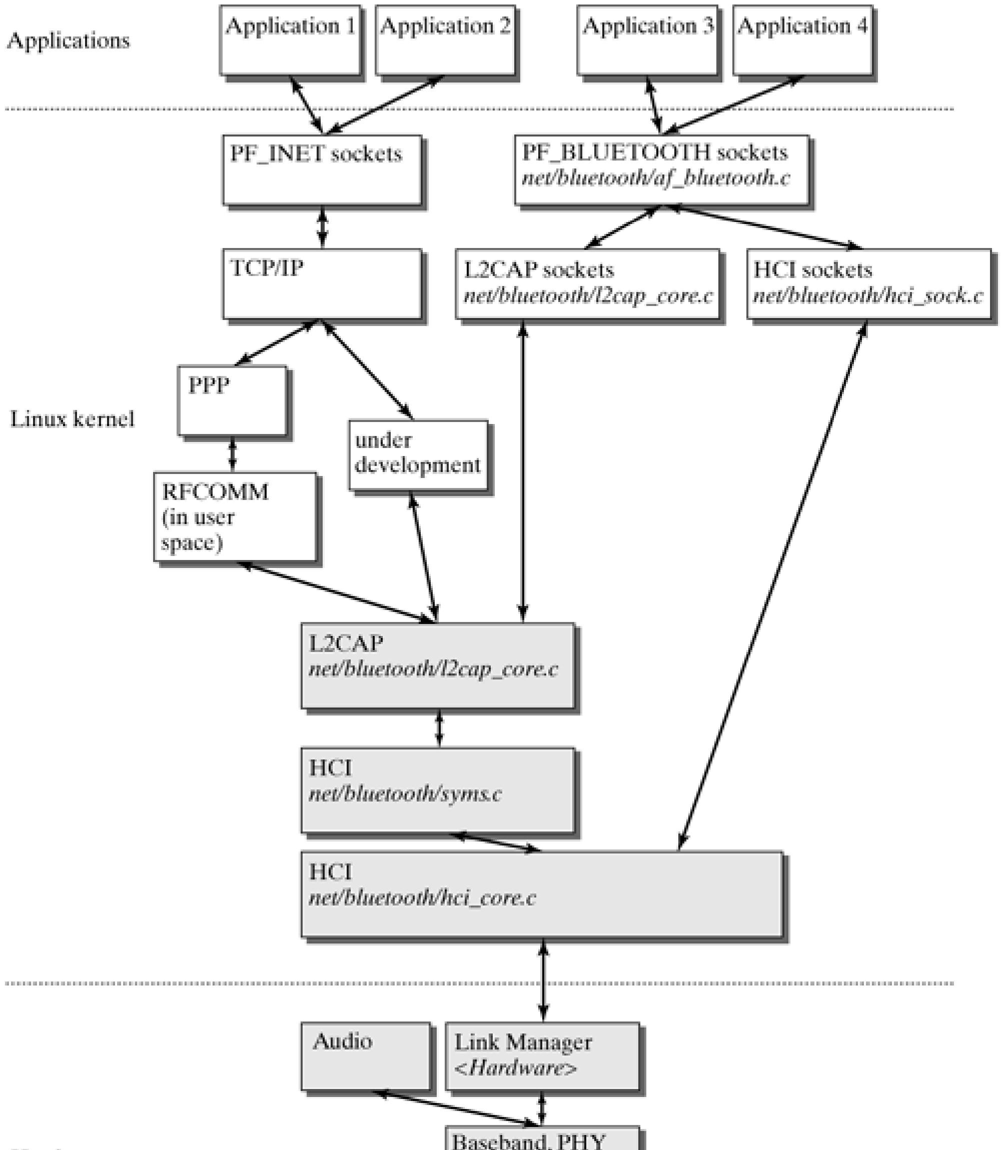
Chapter 11. Bluetooth in Linux

In connection with the enormous proliferation of portable devices such as laptops, PDAs (Personal Digital Assistants), and mobile phones, it becomes increasingly important to find ways to network these devices. Wireless technologies appear to be an ideal solution to this problem, because they don't need a permanently installed infrastructure and facilitate fast establishment and tear-down of networks (so-called ad-hoc networks). In 1998, a number of manufacturers, including Ericsson, Nokia, IBM, Toshiba, and Intel, cooperated in the development of a standard for wireless communication over short distances for consumer electronics. The result of this joint effort is the Bluetooth technology, which operates in the 2.4 GHz frequency range. The Bluetooth consortium specified the radio interface and the higher protocol layers (Bluetooth core), plus so-called profiles (Bluetooth profiles), each of which defines procedures and parameters of the protocol stack for a specific application field (e.g., telephones, headsets, and file transfer). This standardization effort was intended to ensure interoperability of all Bluetooth devices. The Bluetooth specifications are available at [Group01].

The core specifications include the elements shown in Figure 11-1. The three bottom layers are implemented in the Bluetooth hardware (firmware). The radio interface deals with frequency bands, signal outputs, transmission channel parameters, and other mobile properties. The baseband processing includes both additional transmission-specific aspects and media-access aspects (e.g., detection of devices in the neighborhood and initialization of synchronous or asynchronous communication channels).

Figure 11-1. The Bluetooth protocol stack in the Linux kernel.

[View full size image]



◀ Previous Next ▶

11.1 Host Controller Interface (HCI)

The Host Controller Interface (HCI) forms the interface between the software protocol stack and the Link Manager underneath it, which is implemented in the firmware of a Bluetooth device. Notice that this is a packet-oriented communication between HCI and the Link Manager rather than a device driver. The difference is that HCI does not access the register and the memory locations of a Bluetooth device directly. Instead, it sends command and data packets to the device and receives data packets and event-message packets from this device. This means that the Host Controller Interface offers a uniform interface for accessing the hardware.

11.1.1 Command Packets

There is a uniform packet format for command packets sent by HCI to the Link Manager. All packets are ordered in groups. In the command group (opcode group), we distinguish between individual commands (opcode commands). Each command packet consists of a 10-bit OCF (Opcode Command Field) and a 6-bit OGF (Opcode Group Field). There are the following command groups:

- Link control commands serve to establish a connection to other Bluetooth devices and to control the connection.
- Link policy commands serve to change parameters, which are used by the Link Manager to manage connections. For example, such commands can cause connections to switch into the hold mode.
- Host controller and baseband commands allow you to specify additional parameters for the behavior of the Link Manager (e.g., to filter event messages or to activate the flow control discussed further below).
- Information parameters offer a pure read access to values of a Bluetooth device, such as the size of the transmit buffer, the version number, and the 48-bit Bluetooth device address.

In addition to these groups, there are the following groups: status parameters, testing commands, Bluetooth logo testing, and vendor-specific debug commands.

The following sections describe how command packets can be sent within the Bluetooth implementation in the Linux kernel.

`hci_send_cnd()` net/bluetooth/hci_core.c

This function is used to compose a command packet in the form of an `sk_buff` out of the passed data, the OCF and OGF values, the length, and a pointer to parameters. The function `skb_queue_tail` then appends this packet to the end of the command queue of the `struct hci_dev` of the Bluetooth device.

The structure `hci_dev` is defined in `include/net/bluetooth/hci_core.h`. In addition to the command queue, it contains queues for transmit data. In addition to a number of other parameters, it includes four function pointers, to the functions `open(), close(), flush()`, and `send()` made available by the Bluetooth device.

Finally, the function `hci_send_cmd()` invokes the function `hci_sched_cmd()`, which marks the `hdev->cmd_task` as ready to be executed. This tasklet was assigned to the device by the function `tasklet_init()` within the function call `hci_register_init()` (`net/bluetooth/hci_core.c`) when the HCI support was initialized. In addition, there is a `hdev->rx_task` to receive data and another `hdev->tx_task` to send data. When the tasklet `hdev->cmd_task` runs, then the function `hci_cmd_task()` is invoked. This function invokes `hci_send_frame()`.

`hci_send_frame()` net/bluetooth/hci_core.c

11.2 L2CAP

The Logical Link Control and Adaptation Protocol (L2CAP) handles tasks on the data-link layer in the Bluetooth protocol stack. It establishes ACL connections for the next lower layer, but does not transport pure audio data, which primarily is transported over SCO connections. In particular, the L2CAP protocol is responsible for multiplexing data streams from higher layers to an ACL connection, because there must always be at most one ACL connection at a time between two Bluetooth devices. Other important tasks include the segmenting and reassembling of data packets to be able to send and receive the much larger packets of the higher-layer protocols despite the small packet sizes of the baseband layer. L2CAP supports packet sizes of up to 64 Kbytes.

To multiplex several data streams, L2CAP uses the abstraction of the channel. Each channel is allocated to one specific protocol. There are connection-oriented channels for point-to-point communication and connectionless channels used for group communication. A simple signaling method is used to establish an L2CAP connection. The L2CAP protocol can also be accessed directly from the user space over a socket. This is normally a socket from the `PF_BLUETOOTH` socket family with protocol identifier `BTPROTO_L2CAP`.

Now, when HCI receives an ACL packet, then it is passed to the receive function `l2cap_recv_frame()`. If the channel identifier in the packet header is 0x0001, then it is a signaling packet. Subsequently, the function `l2cap_sig_channel()` is invoked; otherwise, the function `l2cap_data_channel()` is invoked.

`12cap_sig_channel()` net/bluetooth/12cap_core.c

The type of signaling packet is recognized within this function and, depending on the type, an appropriate handling function is invoked:

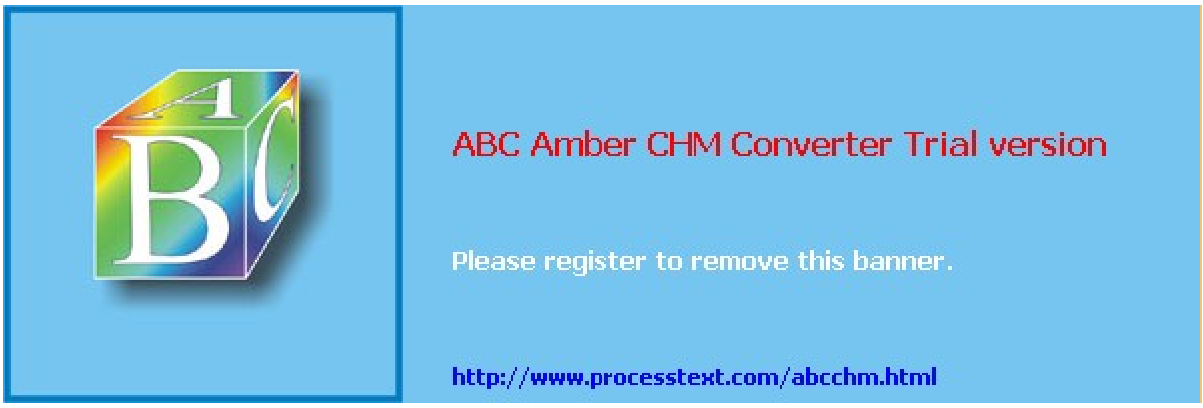
```
switch (cmd.code) {
       case L2CAP_CONN_REQ:
             err = l2cap_connect_req(conn, &cmd, data);
             break;
       case L2CAP_CONN_RSP:
             err = l2cap_connect_rsp(conn, &cmd, data);
             break;
       case L2CAP_CONF_REQ:
             err = l2cap_config_req(conn, &cmd, data);
             break;
       case L2CAP_CONF_RSP:
             err = l2cap_config_rsp(conn, &cmd, data);
             break;
       case L2CAP_DISCONN_REQ:
             err = l2cap_disconnect_req(conn, &cmd, data);
             break;
       case L2CAP_DISCONN_RSP:
             err = l2cap_disconnect_rsp(conn, &cmd, data);
             break;
```

The following section uses the example of an incoming connection request from a remote communication partner in the Bluetooth network to describe how the L2CAP protocol implementation works.

11.2.1 Connection Establishment Phase

When a request to establish a connection arrives from a remote communication partner, then the signaling code `L2CAP_CONN_REQ` is detected, and the function `l2cap_connect_req()` is invoked.

`12cap_connect_req()` net/bluetooth/12cap_core.c

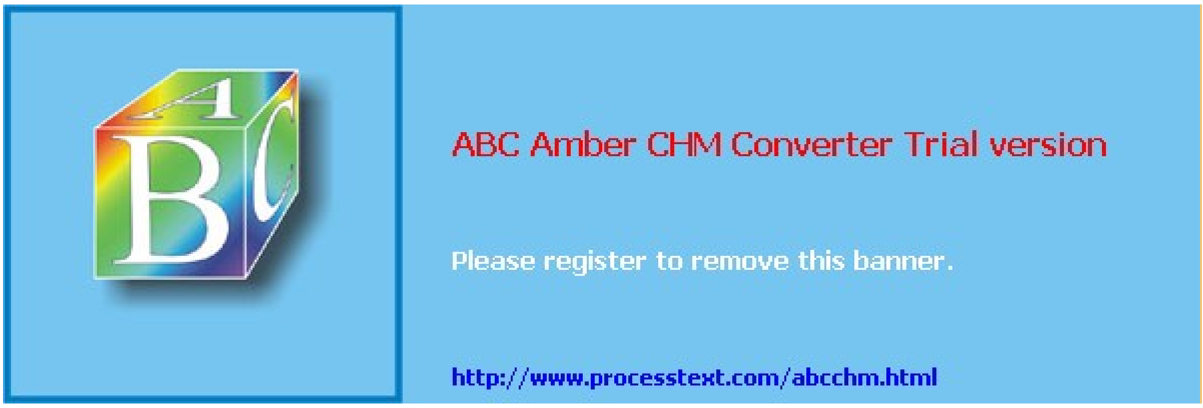


◄ Previous Next ►

11.3 Other Protocols

The L2CAP functionality is currently available for `BTPROTO_L2CAP` sockets only. An interface to higher protocol layers, such as RFCOMM, or for future developments that will allow you to run TCP/IP directly over L2CAP, was not available in the Linux kernel implementation at the time of writing. However, the L2CAP sockets allow you to install these protocols in the user space. The SDP (Service Discovery Protocol) protocol is not integrated in the Linux kernel either. RFCOMM might be implemented in the kernel in the future, but this is currently not intended for SDP.

◄ Previous Next ►



◀ Previous Next ▶

Chapter 12. Transparent Bridges

◀ Previous Next ▶

◀ Previous Next ▶

12.1 Introduction

Local area networks (LANs) are limited both in their reach and in the number of stations that can be connected. For example, only a maximum of 30 stations per segment can be connected to Ethernet based on the 10Base2 standard; and even if you connect fewer than the maximum number of stations, but use an extremely traffic-intensive application, it can happen that the traffic in a LAN is so high that the throughput of the entire network drops rapidly.

This degradation is due mainly to the fact that local area networks are broadcast networks? when station A sends a data frame to station B, then the data packet is concurrently transported to all other stations. The bandwidth in a local area network is used only by the sending station at that time (asynchronous time division multiplexing? TDM). The more stations there are in a local area network, the smaller is the share of each single station. Depending on the network technology, a lot of additional time might be used to decide which station may send next (Medium Access Control).

For the above reasons, it is meaningful to divide a heavily loaded or very large local area network into several subnetworks. Similarly, several local area networks can be linked by single coupling elements to form one large internetwork. In this regard, the parts of the original local area network should not be split into different subnetworks (as is possible in IP), but should always represent themselves as one single (sub)network to the network layer. The two networks are connected transparently, for the network layer.

One coupling element that can link different local area networks to form one single logical LAN is called a bridge. A bridge connects several local area networks on the data-link layer (layer 2 in the OSI r eference model) and distributes the traffic over the subnetworks. Stations that communicate often are generally grouped into one subnetwork. Grouping frequently communicating stations within the same subnetwork means that the entire network has less load to carry, because these stations can exchange traffic within their subnetwork regardless of the traffic in other subnetworks.

◀ Previous Next ▶

12.2 Basics

As was mentioned above, a bridge is a coupling element that links several local area networks on the data link layer [BaHK94]. For this purpose, a bridge has two or more network adapters (ports), which are used to connect to a local area network. In contrast to a repeater, which can merely extend the distance of a LAN, and which simply forwards packets as it received them, a bridge can evaluate certain information in a packet and decide whether that packet should be forwarded.

Bridges come in different variants and with various properties, which will be briefly introduced below:

- Local or remote bridges: Local bridges connect two or more neighboring local area networks? see Figure 12-1. These local area networks are normally linked on the MAC layer.

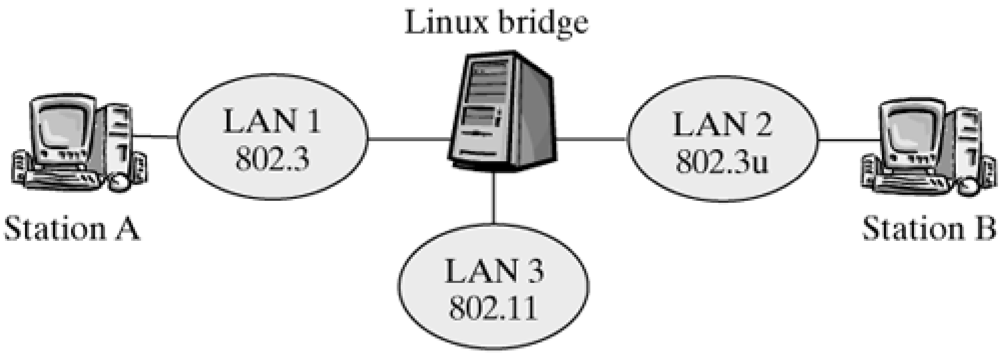
  Remote bridges connect two local area networks physically separated by another network, normally a Wide Area Network (WAN). This bridge type interconnects local area networks on the LLC layer. [BaHK94] includes a detailed description of local and remote bridges. This chapter considers only local bridges.

- Translation or nontranslation bridges: A translation bridge is capable of connecting several local area networks over different media-access protocols (e.g., Ethernet and token ring). Linux is limited in supporting this property, because there could be problems during the transition from one standard to another one. For example, 802.3 supports a limited maximum frame size of 1,500 bytes, but 802.5 supports a much bigger size. For this reason, we cannot feed large 802.5 packets into an 802.3 network.

- Source-routing or transparent bridges: Source-routing bridges represent an extension of the token-ring standard and must be used in token-ring networks only. We will not consider them any further.

  In contrast, transparent bridges can be used in all 802.x networks. They mainly handle the transparent interconnection of different 802.x LANs, where the participating stations do not know that there is a bridge in the LAN. In other words, the bridge is not visible to the stations in the interconnected LANs? it is transparent. The bridge functionality under Linux corresponds exactly to the type of a transparent bridge.

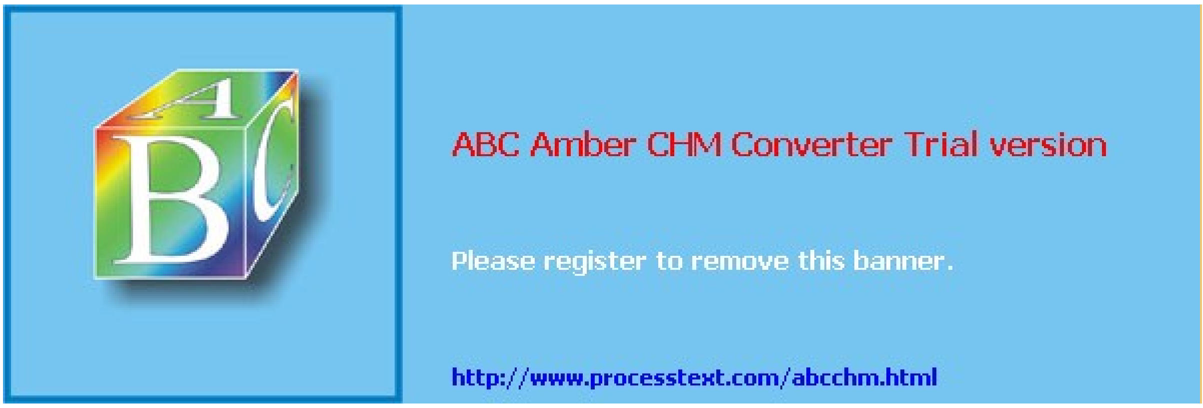
Figure 12-1. A Linux computer acts as a transparent bridge, connecting several local area networks.



12.2.1 Properties of Transparent Bridges

bridges: In accord with the definition in Section 12.2, a Linux system can be used to implement a local transparent translation[1] bridge, which can interconnect different 802.x LANs. Figure 12-1 shows an example in which the Linux computer acts as a bridge, connecting three LANs of different types: one Ethernet (IEEE 802.3), one Fast Ethernet (IEEE 802.3u), and one wireless LAN (IEEE 802.11).

[1] ... however, with the limitation that the 802.x networks be compatible, mainly with regard to their maximum frame lengths. For example, 802.3 and 802.11 can easily be combined, but problems could arise when you use 802.5 LANs.



◂ Previous Next ▸

12.3 Configuring a Bridge in Linux

A bridge interconnects several local area networks on the data-link layer, simulating the behavior of one large single network to the outside. To connect several local area networks in a Linux system, we need only install several network adapters in the computer. Linux also allows you to manage several bridges within one system, which can operate independently of one another. Each bridge instance has a logical name. One network adapter can always belong to exactly one bridge instance. This allows the system administrator to build virtual local networks (VLANs), which previously required expensive VLAN switches.

The following sections introduce options to configure and control Linux bridges.

12.3.1 Configuring the Kernel and the Network Adapter

To be able to use a Linux system as a bridge, the Linux kernel has to contain the bridge functionality. This is normally not the case, so we have to create a new kernel. When configuring the kernel, you should select the `BRIDGING` option from the `Networking Options`. You can integrate it into the kernel either as a module or permanently.

Once you have booted your new kernel (and loaded the module, if applicable), you can use the bridge functionality. Sometimes, you might incur problems when trying to activate several network adapters. If this happens, you can specify the boot parameters `linux ether=0,0,ethx` for each card when you start the system. If you use the LILO boot loader, you can also have the boot parameter passed automatically.

If the bridge functionality resides in the loaded kernel and all network adapters are activated, you can use the `brctl` tool to create and configure the desired bridge instances. `brctl` will be introduced in the next section.

12.3.2 Using the `brctl` Tool to Configure Linux Bridges

You can use the `brctl` (Bridge Control) tool to configure a bridge in Linux. This tool is part of the `bridge-utils` package and can be obtained from [Buyt01].

This tool can be used by the administrator to pass control commands to the bridge implementation in the kernel by using `ioctl()` commands. This section gives an overview of how you can use this program. [BoBu01] includes a detailed description of these commands and several examples.

The `brctl` tool lets you use the following commands to activate and deactivate a bridge. The commands are passed as parameters when `brctl` is called:

- `addbr bridge`: This command creates a new instance of a bridge with the identifier `bridge`.
- `addif bridge device`: This command adds the network adapter `device` to `bridge`. A network adapter can always belong to one bridge only.
- `delbr bridge`: This command deletes the instance of the specified bridge.
- `delif bridge device`: This command deletes the adapter `device` from `bridge`.

The following commands are available in the `brctl` tool to change the default parameters of a bridge:

- `setaging bridge time`: This command sets the max age parameter to the specified value. The topology of the LAN internetwork is recalculated when a BPDU with a larger aging time arrives.
- `setbridgeprio bridge prio`: This command sets the bridge priority, not to be confused with the port priority.
- `setfd bridge time`: This command sets the bridge forward delay parameter. This value is added to the aging time parameter of a BPDU in each bridge.

◂ Previous Next ▸

12.4 Implementation

The implementation of the bridge functionality discussed here is relatively new. It has been integrated into the Linux kernel since Version 2.2.14 and 2.3.x and replaces the former and in many ways less flexible implementation. This version includes several new functions (e.g., the capability of managing several bridges in one system, and better options to configure the bridge functionality).

In addition, several details of the implementation have changed to provide more efficient handling. Among other things, the forwarding table is no longer stored in the form of an AVL tree, but in a hash table. Though AVL trees are data structures with a relatively low search cost, O(log n), hash tables are generally faster when the collision domain remains as low as possible. This means that a well-distributed hash table has the cost O(1). We can assume that a Linux bridge has to store several hundred reachable systems at most, so a hash table is probably the better choice, especially considering that it is much easier to configure.

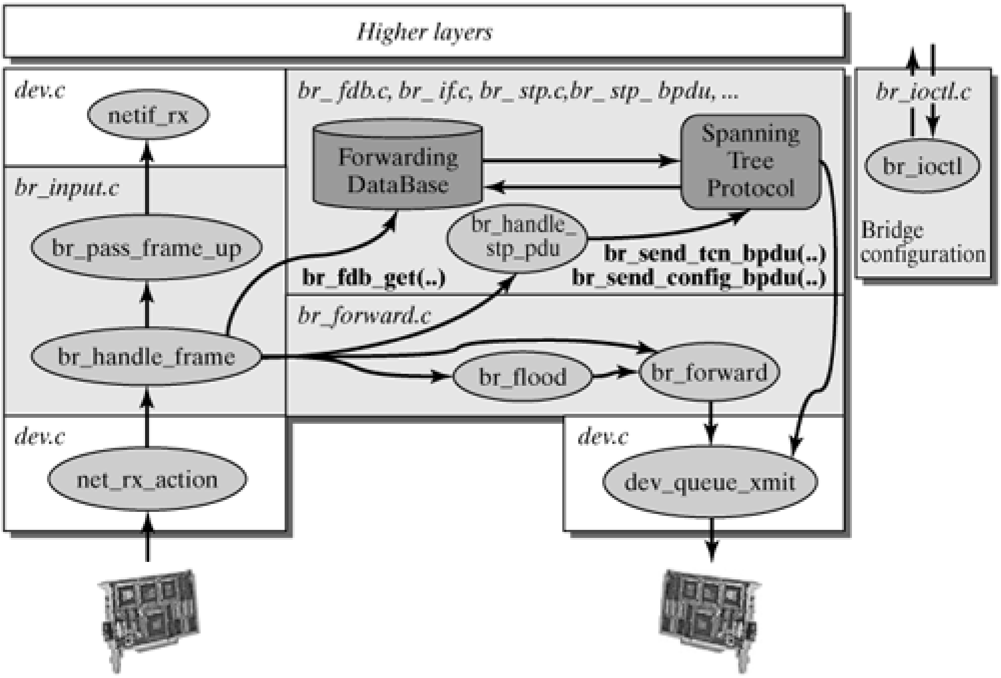
The following sections describe in more detail how you can implement the bridge functionality in Linux. We will first introduce the most important data structures and how they are linked, then discuss the algorithms and functions.

12.4.1 Architecture of the Bridge Implementation

Figure 12-12 shows the architecture of the bridge implementation in the Linux kernel. The individual components are divided, by their tasks and over several files. This makes the program text easier to understand and forces the programmer to define the interfaces between the individual components well.

Figure 12-12. Integrating the bridge implementation into the Linux network architecture.

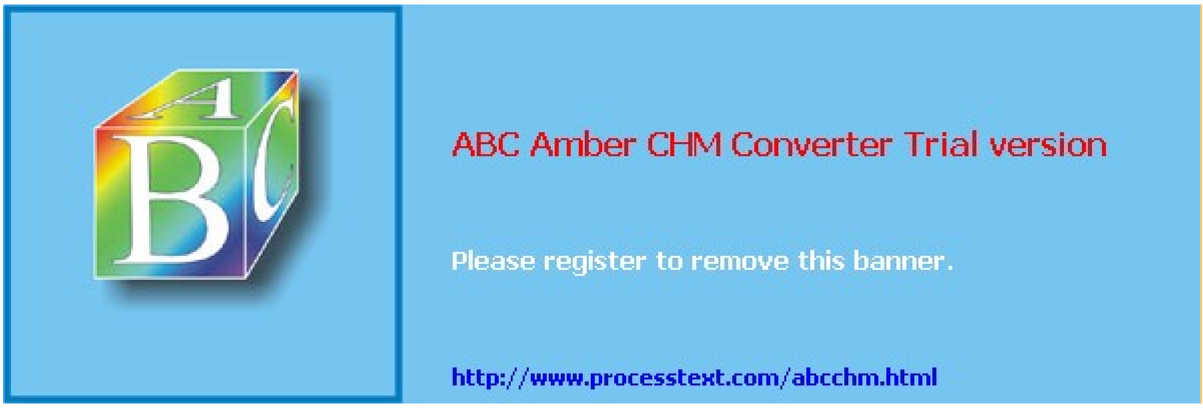
[View full size image]



12.4.2 Building and Linking Important Data Structures

The most important data structures of a Linux bridge include information about the bridges themselves and information about the network adapters (ports) allocated to them. We want to repeat here that you can use the new bridge implementation to construct several logically separated bridges in a Linux system. For example, this allows you to easily configure virtual local area networks (VLANs) that are not mutually accessible. In addition to information about the bridge and its ports, you need to store the forwarding table (filter table) for each bridge.

The forwarding table stores the IDs of each reachable station and the port used to reach that station.



◂ Previous Next ▸

Part IV: Network Layer

◂ Previous Next ▸

◂ Previous Next ▸

Chapter 13. The TCP/IP Protocols

This chapter introduces the TCP/IP protocol suite, which represents the basis of the popular Internet. Chapter 3 introduced the TCP/IP reference model. The sections in this chapter and the following chapters begin with an introduction of the tasks of each of these protocols and then describe how they operate and how they are implemented in Linux.

The history of the Internet and its protocols began in 1961, when Leonard Kleinrock developed packet-switching theory at MIT. His work was based on the idea of splitting data into many small packets and sending them to the destination separately, without specifying the exact path. After initial skepticism, the principle was eventually used in a research project of ARPA (Advanced Research Projects Agency), a division of the United States Department of Defense. In 1968, ARPA granted a budget of more than half a million dollars for a heterogeneous network, which was called ARPANET.

In 1969, this experimental network connected the four universities of Los Angeles (UCLA), Santa Barbara (UCSB), Utah, and the Stanford Research Institute (SRI) and expanded very quickly. Later, satellite and cellular links were successfully connected to the ARPANET. In one impressive demonstration, a truck in California was connected with the next university over a radio link and used the satellite network to access a computer based in London, UK.

This system was used intensively in the years following. On the basis of the knowledge gained from this system, a second generation of protocols was developed. By 1982, a protocol suite with the two important protocols, TCP and IP, had been specified. Today, the name TCP/IP is used for the entire protocol suite. In 1983, TCP/IP became the standard protocol for the ARPANET. The TCP/IP protocols proved particularly suitable for providing a reliable connection of networks within the continually growing ARPANET. ARPA was very interested in establishing the new protocols and convinced the University of California at Berkeley to integrate the TCP/IP protocols into its widely used Berkeley UNIX operating system. They used the principle of sockets to design applications with network functionality. This helped the TCP/IP protocols to soon become very popular for the exchange of data between applications.

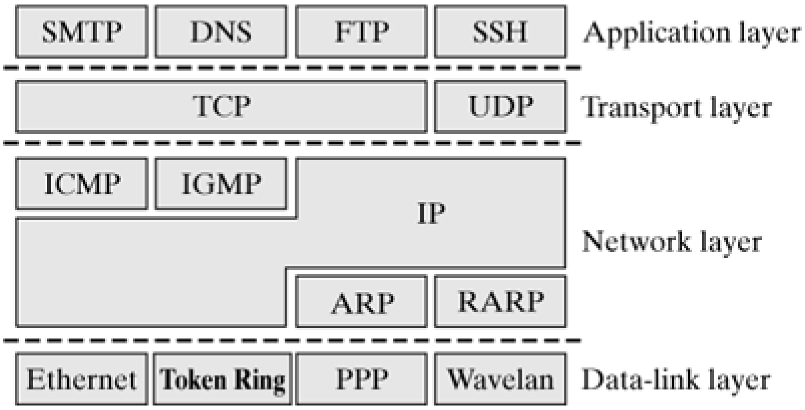
In the following years, the ARPANET had grown to a size that made the management of all computers IP addresses in one single file too expensive. As a consequence, the Domain Name Service (DNS) was developed and is used to hide IP addresses behind easy-to-remember computer and domain names. Today, the Internet protocol Version 4 is the most frequently used network-layer protocol. However, it was not designed for such an enormous proliferation and has already hit its capacity limits, so a new version had to be developed. The new Internet Protocol Version 6 is also called IPv6 or IPng.

◂ Previous Next ▸

13.1 The Internet Protocol Suite

Each protocol of the TCP/IP protocol suite handles certain tasks within the TCP/IP protocol stack. Figure 13-1 gives an overview of the TCP/IP protocol stack and its protocols.

Figure 13-1. The protocols of the TCP/IP protocol stack.



- On the data-link layer in the Internet model, you find network adapters and their drivers. They allow you to exchange data packets having a specific maximum length within the connected LAN (Ethernet, token ring,...) or within a WAN (PPP over ISDN, ATM). The previous chapters introduced some protocols that also belong to the data-link layer (SLIP, PPP, ATM, Bluetooth, etc.). All adapters and protocols on this layer have the common property that they represent only one communication link between two IP routers (i.e., they don't support Internet routing).

- The Address Resolution Protocol (ARP) also resides on the data-link layer. Notice that there are contradictory opinions in the literature. ARP is used to map globally valid IP addresses to locally valid MAC addresses. ARP is actually not limited to IP addresses or specific physical addresses; it was designed for general use. ARP uses the broadcast capability of local area networks to find addresses. Chapter 15 describes this protocol in detail.

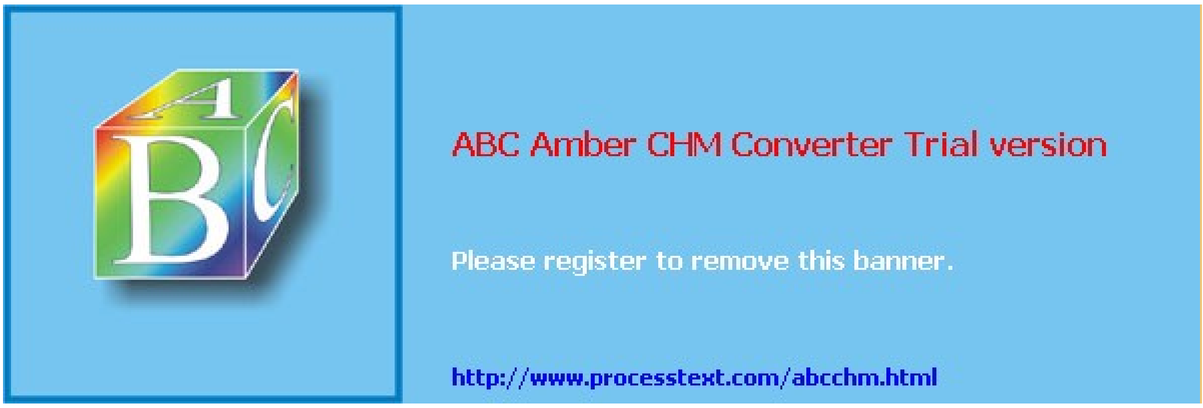
- The Internet Protocol (IP) forms the core of the entire architecture, because it allows all IP-enabled computers in the interconnected networks to communicate. Each computer in the Internet has to support the Internet Protocol. IP offers unreliable transport of data packets. IP uses information from routing protocols (OSPF, BGP, etc.) to forward packets to their receivers.

- The Internet Control Message Protocol (ICMP) has to be present in each IP-enabled computer; it handles the transport of error messages of the Internet Protocol. For example, ICMP sends a message back to the sender of a packet if the packet cannot be forwarded because routing information is missing or faulty. Section 14.4 deals with ICMP and its implementation in Linux.

- The Internet Group Management Protocol (IGMP) is responsible for managing multicast groups in local area networks. Multicast provides for efficient sending of data to a specific group of computers. IGMP allows the computers of a LAN to inform its router that they want to receive data for a certain group in the future. Chapter 17 discusses multicast in the Internet.

- The Transmission Control Protocol (TCP) is a reliable, connection-oriented and byte-stream-oriented transport-layer protocol. TCP is primarily responsible for providing a secured data transport between two applications over the unreliable service of the IP protocol. TCP is the most frequently used transport protocol in the Internet. It has a large functionality, and so its implementation is extensive. Chapter 24 discusses the TCP.

- The User Datagram Protocol (UDP) is a very simple transport protocol, providing connectionless and unreliable transport of data packets between applications in the Internet. In this context, unreliable does not mean that the data could arrive corrupted at the destination computer. It means that UDP does not offer any protocol mechanisms to guarantee that the data will arrive at the destination at all. When data arrives at the destination



◀ Previous Next ▶

Chapter 14. The Internet Protocol V4

The Internet Protocol (IP) is the central element in the TCP/IP protocol stack. It provides the basic service for all the data traffic in the Internet and other IP-based networks and was specified in RFC 791. The primary task of the Internet Protocol is to hide differences between data transmission layers and to offer a uniform presentation of different network technologies. For example, the Internet protocol can run on top of LAN technologies and SLIP (Serial Line IP) or PPP (Point-to-Point Protocol) over modem or ISDN connections. The uniform presentation of the underlying technology includes an introduction of the uniform addressing scheme (IP address family) and a mechanism to fragment large data packets, so that smaller maximum packet sizes can be transported across networks.

In general, each network technology defines a maximum size for data packets? the Maximum Transmission Unit (MTU). The MTU depends on the hardware used and the transmission technology and varies between 276 bytes and 9000 bytes. The Internet layer fragments IP datagrams, which are bigger than the MTU of the network technology used, into smaller packets (fragments). These fragments of a datagram are then put together into the original IP datagram in the destination computer. Section 14.2.3 explains how data packets are fragmented and reassembled.

In summary, the Internet Protocol handles the following functions:

- provides an unsecured connectionless datagram service;
- defines IP datagrams as basic units for data transmission;
- defines the IP addressing scheme;
- routes and forwards IP datagrams across interconnected networks;
- verifes the lifetime of packets;
- fragments and reassembles packets; and
- uses ICMP to output errors.

◀ Previous Next ▶

◂ Previous Next ▸

14.1 Properties of the Internet Protocol

The Internet Protocol was developed with the idea of maintaining communication between two systems even when some transmission sections fail. For this reason, the Internet Protocol was developed on the basis of the principle of datagram switching, to transport IP data units, rather than on that of circuit-switching, like conventional telephone network.

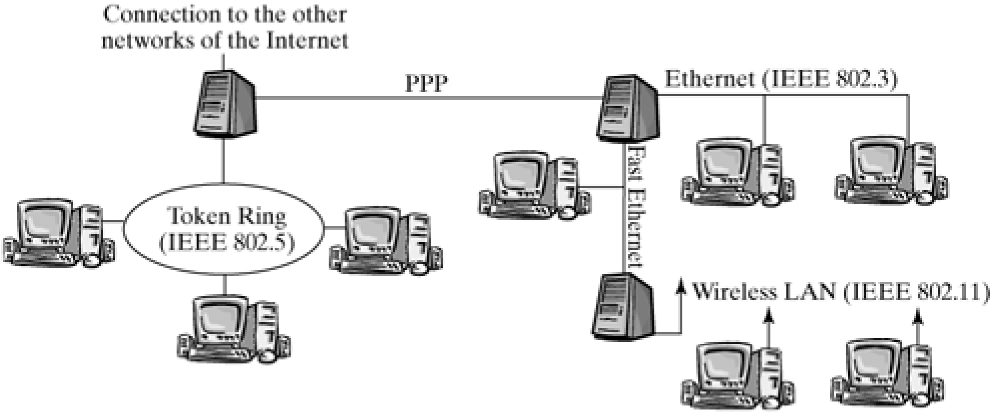
The following sections describe the protocol mechanisms of the Internet Protocol. Section 14.2 will then explain how IP is implemented in the Linux kernel.

14.1.1 Routing IP Packets Across Routers

Figure 14-1 shows how the Internet is structured. Rather than being one single network, the Internet is composed of many smaller local area networks, which are connected by routers. This is the reason why it is often called the network of networks or global network. Each network connected to the Internet can be different both in size and in technology. Within one network (e.g., the network of a university), it is often meaningful to build several subnetworks. These? often independent? networks and subnetworks are connected by routers and point-to-point lines.

Figure 14-1. The structure of the global Internet.

[View full size image]


Connection to the other networks of the Internet
PPP
Ethernet (IEEE 802.3)
Fast Ethernet
Token Ring (IEEE 802.5)
Wireless LAN (IEEE 802.11)

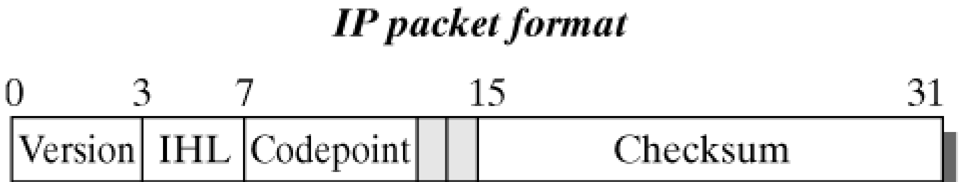
The interconnection of single local area networks offers a way to send data from an arbitrary computer to any other computer within the internetwork. Before it sends a packet, an Internet computer checks for whether the destination computer is in the same local area network. If this is not the case, then the data packet is forwarded to the next router. If both the sender and the receiver are in the same local area network, then the packet is delivered to the receiver directly over the physical medium. In either case, the IP layer uses the service of the data-link layer to physically transport the packet (horizontal communication? see Section 3.2).

Let's assume that, in the first case, the packet has not yet arrived in the destination computer. The router checks the destination address in the IP packet header and the information in the routing table to determine how the packet should be forwarded. Next, the packet travels from one router to the next until it eventually arrives in the destination computer. Chapter 16 discusses routing in IP networks.

14.1.2 The IP Packet Header

Figure 14-2 shows the format of an IP packet. The fields of the IP packet header have the properties described below.

Figure 14-2. Packet-header format of the Internet Protocol.



◂ Previous Next ▸

14.2 Implementing the Internet Protocol

This section explains the architecture of the IP instance in the Linux kernel. We will use the path a packet takes across the IP layer to introduce the basic properties of the Internet Protocol. We assume that this is a normal IP packet without special properties, to ensure that our explanations will be clear and easy to understand. All special functions of the Internet Protocol, such as fragmenting and reassembling, source routing, multicasting, and so on, will be described in the next chapters.

The objective of this section is to introduce the fundamental operation of the IP implementation in Linux, to be able to better understand more complex parts later on. This section also serves as an entry point into the other chapters of this book, because each packet passes the IP layer, where it can take a particular path (e.g., across a firewall or a tunnel). It is necessary to understand how the Internet Protocol is implemented in the Linux kernel to understand later chapters.

An IP packet can enter the IP instance in three different places:

- Packets arriving in a computer over a network adapter are stored in the input queue of the respective CPU, as described in Chapter 6. Once the layer-3 protocol in the data-link layer has been determined (which is `ETH_PROTO_IP` in this case), the packets are passed to the `ip_rcv()` function. The path these packets take will be described in Section 14.2.1.
- The second entry point for IP packets is at the interface to the transport protocols. These are packets used by TCP, UDP, and other protocols that use the IP protocol. They use the `ip_queue_xmit()` function to pack a transport-layer PDU into an IP packet and send it. Other functions are available to generate IP packets at the boundary with the transport layer. These functions and the operation of `ip_queue_xmit()` will be described in Section 14.2.2.
- With the third option, the IP layer generates IP packets itself, on the Internet Protocol's initiative. These are mainly new multicast packets, new fragments of a large packet, and ICMP or IGMP packets that don't include a special payload. Such packets are created by specific methods (e.g., `icmp_send()`). (See Section 14.4.)

Once a packet (or socket buffer) has entered the IP layer, there are several options for how it can exit. We generally distinguish two different roles a computer can assume with regard to the Internet Protocol, where the first case is a special case of the second:

- End system: A Linux computer is normally configured as an end system? it is used as a workstation or server, assuming primarily the task of running user applications or providing application services. Also, a Web server and a network printer are nothing but end systems (with regard to the IP layer). The basic property of end systems is that they do not forward IP packets. This means that you can recognize an end system easily by the fact that it has only one network adapter. Even a system that has several network accesses can be configured as a host, if packet forwarding is disabled.
- Router: A router passes IP packets arriving in a network adapter to a second network adapter. This means that a router has several network adapters that forward packets between these interfaces. When packets arrive in a router, there are generally two options: they can deliver packets locally (i.e., deliver them to the transport layer) or they can forward them. The first case is identical with the procedure of packets arriving in an end system, where packets are always delivered locally. Consequently, a router can be thought of as a generalization of an end system, with the additional capability of forwarding packets. In contrast to end systems, generally no applications are started in routers, to ensure that packets can be forwarded as fast as possible.

Linux lets you enable and disable the packet-forwarding mechanism at runtime, provided that the forwarding support was integrated when the kernel was created. The directory `/proc/sys/net/ipv4/` includes a virtual file, `ip_forward`. You will see in Appendix B.3 that there is a way to change system settings from within the `proc` directory. If a `0` is written to this file, then packet forwarding is disabled. To activate IP packet forwarding, you can use the command `echo '1' > /proc/sys/net/ipv4/ip_forward`.

◀ Previous Next ▶

14.3 IP Options

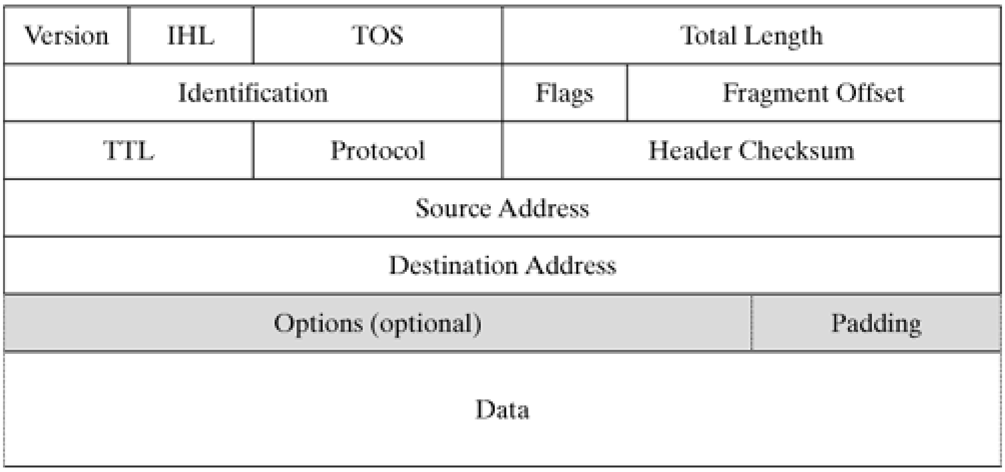
When a packet is sent to the IP layer, then it normally includes all required information in the packet's protocol header. However, there could be times when packets require additional information in the protocol header? for example, for diagnostics purposes, or if a packet's path across the Internet is specified before it is sent. For these purposes, an Option field with variable length can be added to each IP packet header. All guidelines for these IP options are described in [Post81c].

14.3.1 Standardized IP Packet Options

Figure 14-9 shows that the IP packet options are appended to the end of an IP header. The length of the Option field is variable, and the end of a packet header has to be aligned to a 32-bit boundary, so a n additional padding field of the appropriate length is added (and set to 0 by default). In this case, "variable" also means that the packet options can be left out, if they are not required. The Option field can take one or several packet options, where an option can be given in either of two formats:

- One single byte describes only the option type. The length of these options is always exactly one byte.
- The first byte includes the option type, and the second byte contains the length of this packet option. The following bytes include the actual data of that option.

Figure 14-9. The IP packet header.



The byte stating the length of the packet option in the second case includes merely the number of data bytes. The first two bytes are not counted. The option type in the first byte is composed as follows:

Copy Flag	Option Class	Option Number

The (1-bit) copy flag is required for packet fragmentation. If a packet has to be fragmented, then this bit states whether this packet option has to appear in all fragments or may be set in the first fragment only.

The option class is represented by 2 bits. The (5-bit) option number shows the length of a packet option implicitly (i.e., we can see whether the next byte also belongs to this packet option or already belongs to the next packet option). Table 14-1 lists all IP packet options defined in RFC 791, including their lengths and their defined option numbers and option classes. There are four option classes in total, but only two are currently used. Option class 0 includes packet options for control and management; option class 2 includes debugging and measurement options. The option classes 1 and 3 are reserved for future IP packet-option classes.

Table 14-1. Defined IP packet options.

◂ Previous Next ▸

14.4 Internet Control Message Protocol (ICMP)

The Internet Control Message Protocol (ICMP) is the error-report mechanism for the IP layer, which also resides in the network layer. Though ICMP is based on IP, it doesn't make IP more reliable. Packets can be lost despite the use of ICMP, and IP or ICMP won't notice that packets are lost. The only purposes of this error-report mechanism are to report errors to other computers and to respond to such reports. It is mendatory for each IP implementation to implement ICMP. The ICMP implementation is defined in the following RFC documents:

- RFC 792 [Post81b]: This is the basic definition, describing the packet types and their uses.
- RFC 1122 [Brad89]: Definition of the requirements on terminal equipment (hosts) connected to the Internet.
- RFC 1812 [Bake95]: This document describes the requirements for switching computers (routers) in the Internet.

However, RFC specifications often leave much room for flexible implementation. For some functions, it is even optional whether you implement them. For this reason, ICMP implementations and even configurations of the same implementation can differ considerably.

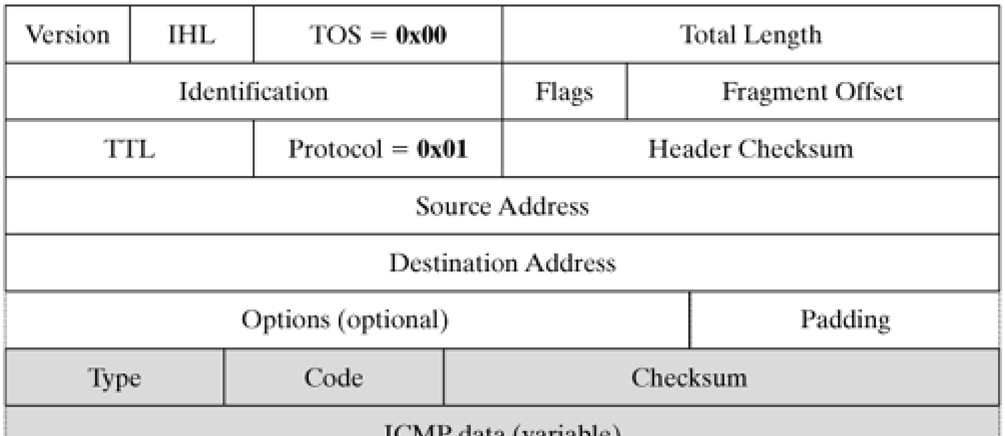
The most popular application of ICMP is error detection or error diagnostics. In more than ninety percent of all cases, the first information transmitted by a newly installed network adapter over an IP network will probably be that of the `ping` command, which is fully based on ICMP. This allows you to check the reachability of other computers easily and without noticeable load on the network. This procedure is often done in automated form (e.g., to monitor servers). Beyond simply checking the reachability of computers, the set of different error messages allow a network administrator (or a network-analysis tool) to obtain a detailed overview of the internal state of an IP network. For example, poorly selected local routing tables or wrongly set transmit options in individual computers can be detected. And finally, it is possible to use ICMP to synchronize computer clocks within a network, in addition to other? partly outdated? functions, which will be briefly discussed in this section.

14.4.1 Functional Principle of ICMP

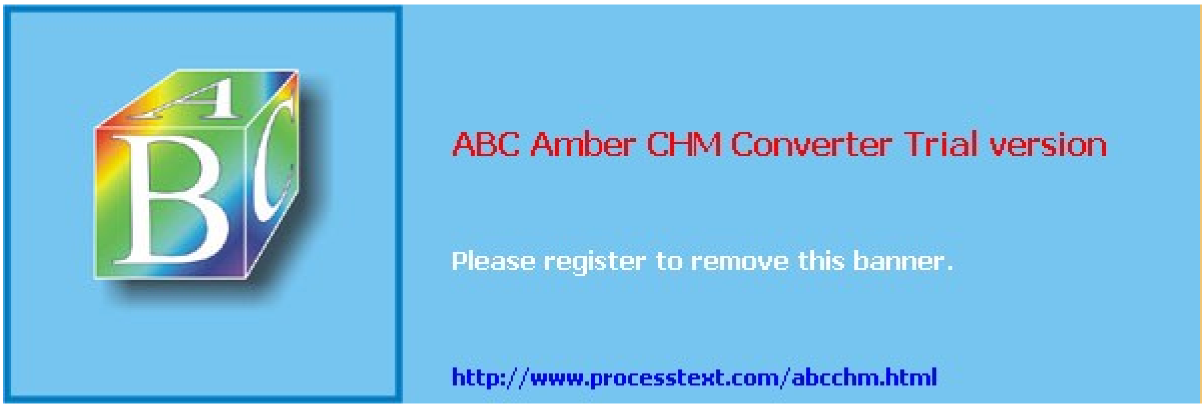
ICMP sends and receives special IP packets representing error or information messages. Error messages occur whenever IP packets have not reached their destinations. All other cases create information messages, which can additionally include a request for reply. Notice that the ICMP functionality becomes active within the network implementation of the Linux kernel only provided that a problem situation occurs during another data traffic or when ICMP packets arrive from another computer. As mentioned earlier, ICMP transmits messages in IP packets. Figure 14-11 shows the general structure of ICMP messages (gray fields), which are transported in the payload of an IP packet. It is typical for the IP header of a packet containing an ICMP message that the Type-of-Service field is set to `0x00`, which means that the packet is treated like a regular IP packet without priority. The protocol type in the IP header for ICMP messages is set to `0x01`, as specified in RFC 790 [Post81a].

Figure 14-11. Structure of an IP packet containing an ICMP message.

[View full size image]



Version	IHL	TOS = **0x00**	Total Length
Identification		Flags	Fragment Offset
TTL	Protocol = **0x01**	Header Checksum	
Source Address			
Destination Address			
Options (optional)			Padding
Type	Code	Checksum	
ICMP data (variable)			



◂ Previous Next ▸

Chapter 15. Address Resolution Protocol (ARP)

The conversion of addresses between different protocol layers represents an important task for unique identification of resources in a computer network. Such a conversion is required at the transition between two neighbouring layers within a reference model, because each layer uses its own address types, depending on its functionality (IP, MAC, ATM addresses, etc.). For example, the destination computer is specified in the form of an IP address if a packet is sent over the Internet Protocol. This address is valid only within the IP layer. In the data-link layer, both the service used by the Internet Protocol to transport its data and different LAN technologies (e.g., Ethernet, token ring, ATM), each with its own address formats, can be used. The network adapters of a LAN are generally identified by 48-bit addresses, so-called MAC addresses. A MAC address identifies a unique network adapter within a local area network.

To be able to send a packet to the IP instance in the destination computer or to the next router, the MAC address of the destination station has to be determined in the sending protocol instance. The problem is now to do a unique resolution of the mapping between a MAC address and an IP address. What we need is a mapping of network-layer addresses to MAC addresses, because the sending IP instance has to pass the MAC address of the next station in the form of interface control information (ICI) to the lower MAC instance. (See Section 3.2.1.) At the advent of the Internet, this mapping was implemented by static tables that maintained the mapping of IP addresses to MAC addresses in each computer. However, this method turned out to be inflexible as the ARPANET grew, and it meant an extremely high cost when changes were necessary. For this reason, RFC 826 introduced the Address Resolution Protocol (ARP) to convert address formats.

Though the TCP/IP protocol suite has become the leading standard for almost all computer networks, it is interesting to note that ARP was not designed specifically for mapping between IP and MAC addresses. ARP is a generic protocol that finds a mapping between ordered pairs (P,A) and arbitrary physical addresses, where P is a network-layer protocol and A is an address of this protocol P. At the time at which ARP was developed, different protocols, such as CHAOS and Decnet, had been used in the network layer. The ARP instance of a system can be extended so that the required addresses can be resolved for each of the above combinations, which means that no new protocol is necessary. The most common method to allocate addresses between different layers maps the tuple (Internet Protocol, IP address) to 48-bit MAC addresses.

◂ Previous Next ▸

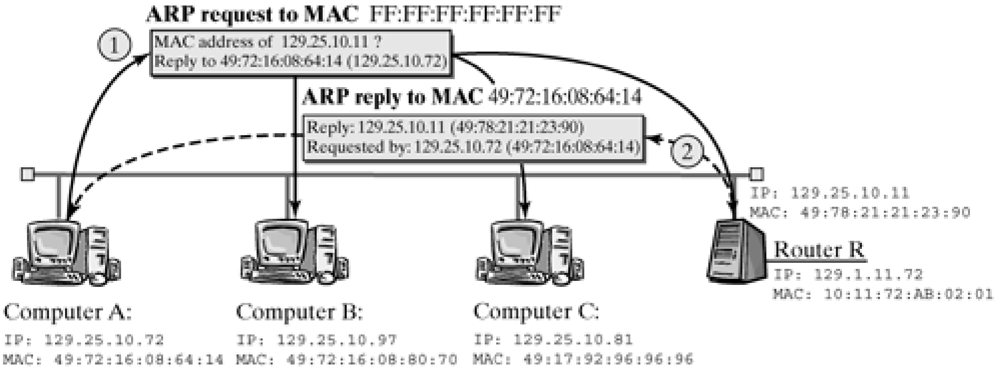
◀ Previous Next ▶

15.1 Using the Address Resolution Protocol

As mentioned above, the Address Resolution Protocol (ARP) is a decentralized protocol to resolve address mappings between layer-3 addresses and layer-2 addresses in local area networks. Figure 15-1 shows how ARP works. When computer A wants to send a packet to router R in the same LAN, then it needs the layer-2 address, in addition to the IP address, to be able to tell the data link layer which computer is supposed to get this packet. For this purpose, computer A sends an ARP Request to all computers connected to the LAN. This request is generally sent in a MAC broadcast message by using the MAC broadcast address (`FF:FF:FF:FF:FF:FF`). The intended computer can see from the destination IP address in the ARP PDU that this request is for itself, so this computer returns a reply to the requesting computer, A, including its MAC address. Computer A now learns the MAC address of R and can instruct its data-link layer to deliver the packet.

Figure 15-1. Example showing how ARP resolves addresses.

[View full size image]



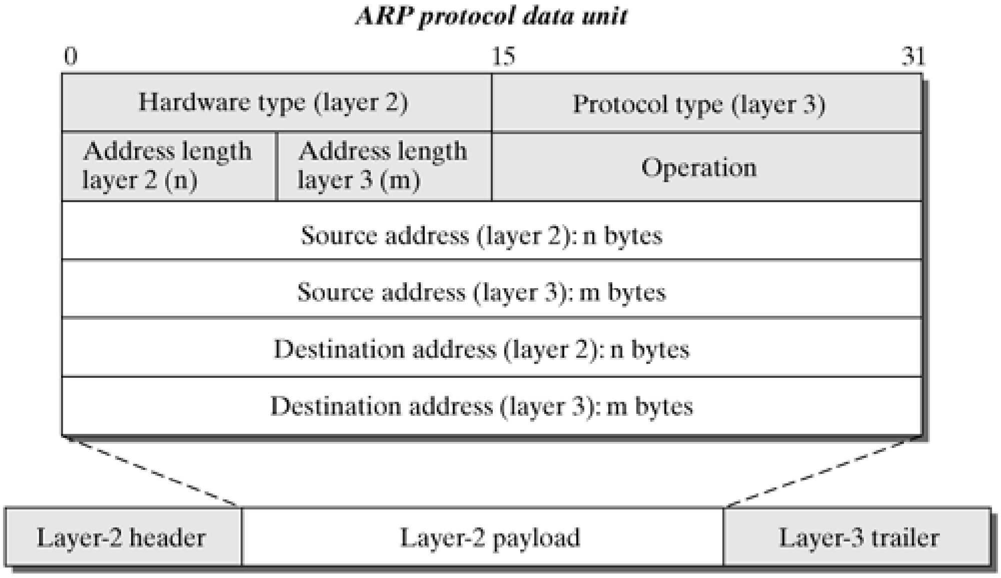
To avoid having to request the MAC address again for subsequent packets, A stores the MAC address of R in a local table? the ARP cache. (See Section 15.3.) Computer R can also extract the MAC address of A from A's request and store that in its own ARP cache. It can be seen from A's request that A and R will communicate soon, which means that the MAC address of A will be needed. In this case, we avoid one ARP request, because the mapping will have been previously stored.

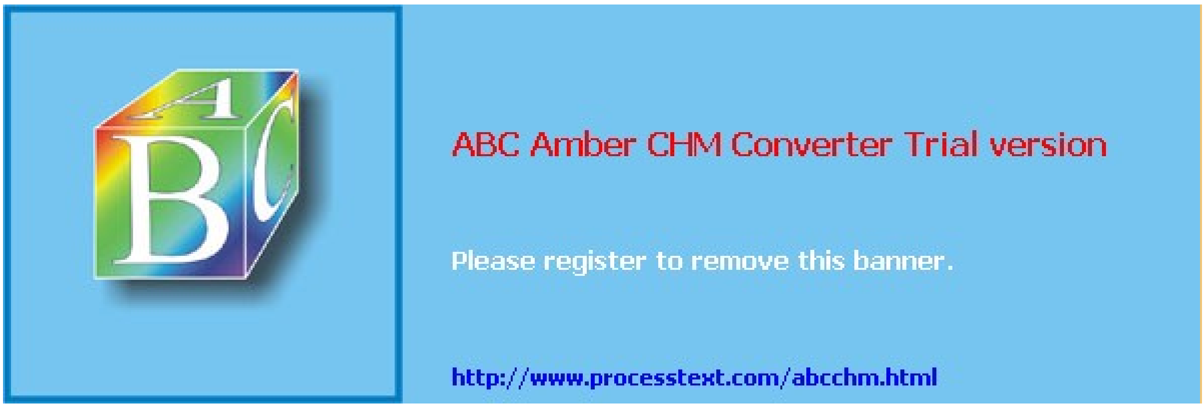
15.1.1 The Structure of ARP Protocol Data Units

Figure 15-2 shows how an ARP PDU is structured; this PDU is used for the two protocol data units defined in the ARP protocol, ARP Request and ARP Reply. The only difference between these two types is in the Operation field.

Figure 15-2. Format of the ARP Request and ARP Reply PDUs.

[View full size image]


ARP protocol data unit
0
15
31
Hardware type (layer 2)
Protocol type (layer 3)
Address length layer 2 (n)
Address length layer 3 (m)
Operation
Source address (layer 2): n bytes
Source address (layer 3): m bytes
Destination address (layer 2): n bytes
Destination address (layer 3): m bytes
Layer-2 header
Layer-2 payload
Layer-3 trailer



◂ Previous Next ▸

15.2 The ARP Command

The `arp` command can be used to output the ARP table (ARP cache) of a computer. It can also be used to manipulate the ARP table (e.g., to create permanent entries or delete entries).

The following options are available for the `arp` command:

- Display the ARP table: you can use option -a when running the `arp` command to view the ARP table of a computer:

  ```
  root@tux # arp -a
  IP address HW type HW address
  129.25.10.97 10Mbit/s Ethernet 49:72:16:08:80:70
  129.25.10.72 10Mbit/s Ethernet 49:72:16:08:64:14
  129.25. 10.81 10Mbit/s Ethernet 49:17:92:96:96:96
  ```

  The first column shows the IP address of the destination computer; the second column shows the LAN category (e.g., 10-mbps Ethernet); the last column shows the layer-2 address of the network adapter.

  If the word `incomplete` appears in an entry in the last column upon repeated calls, then this means that the network device specified by the entry has failed or is defective.

- Address format: In addition to Ethernet, ARP is also used in other broadcast-enabled LAN technologies (e.g., AX.25 amateur radio networks and token ring) for address resolution. These network technologies may use different address formats. `arp` shows the address format used in the second column. Notice that `arp` shows only the entries for Ethernet addresses, by default. To view a list of AX.25 addresses, you have to use the `-t` option with the command: `arp -a -t ax25`.

- Deleting ARP entries: You can use `arp` with the option `-d computer` to remove the entry of that computer. This forces a new ARP request upon the next request for the layer-2 address of the specified computer. Deleting an ARP address mapping can be useful when a computer's configuration is wrong or when the layer-2 address has changed? for example, when a network adapter has been replaced.

  To avoid this case, ARP entries are automatically declared invalid after a certain period of time. This period is in the range of a few minutes, so that the replacement of a network adapter should actually not cause any problem.

- Setting ARP entries: It can sometimes be useful to add an entry manually to the ARP table. The option `-s computer layer-2-address` is available for such cases. It can also be used when ARP requests to a specific computer are not answered, because of faulty or missing ARP instances. The option -s can also be useful when a second computer in the same LAN identifies itself erroneously with the same IP address and replies sooner to the ARP request. The following command adds the computer `tux` having layer-2 address `49:72:16:08:64:14` to the ARP table: arp `-s` tux `49:72:16:08:64:14`.

  In contrast to entries determined automatically in the ARP cache, entries created with the option `-s` are not removed after a certain period; they remain in the ARP cache until the computer restarts (static entry).

◂ Previous Next ▸

◄ Previous Next ►

15.3 Implementing the ARP Instance in the Linux Kernel

In theory, ARP would have to run an address resolution for each outgoing IP packet before transmitting it. However, this would significantly increase the required bandwidth. For this reason, address mappings are stored in a table? the so-called ARP cache? as the protocol learns them. We have mentioned the ARP cache several times before. This section describes how the ARP cache and the ARP instance are implemented in the Linux kernel.

Though the Address Resolution Protocol was designed for relatively generic use, to map addresses for different layers, it is not used by all layer-3 protocols. For example, the new Internet Protocol (IPv6) uses the Neighbor Discovery (ND) address resolution to map IPv6 address to layer-2 addresses. Though the operation of the two protocols (ARP and ND) is similar, they are actually two separate protocol instances. The Linux kernel designers wanted to utilize the similarity between the two protocols and implemented a generic support for address resolution protocols in LANs, the socalled `neighbour` management.

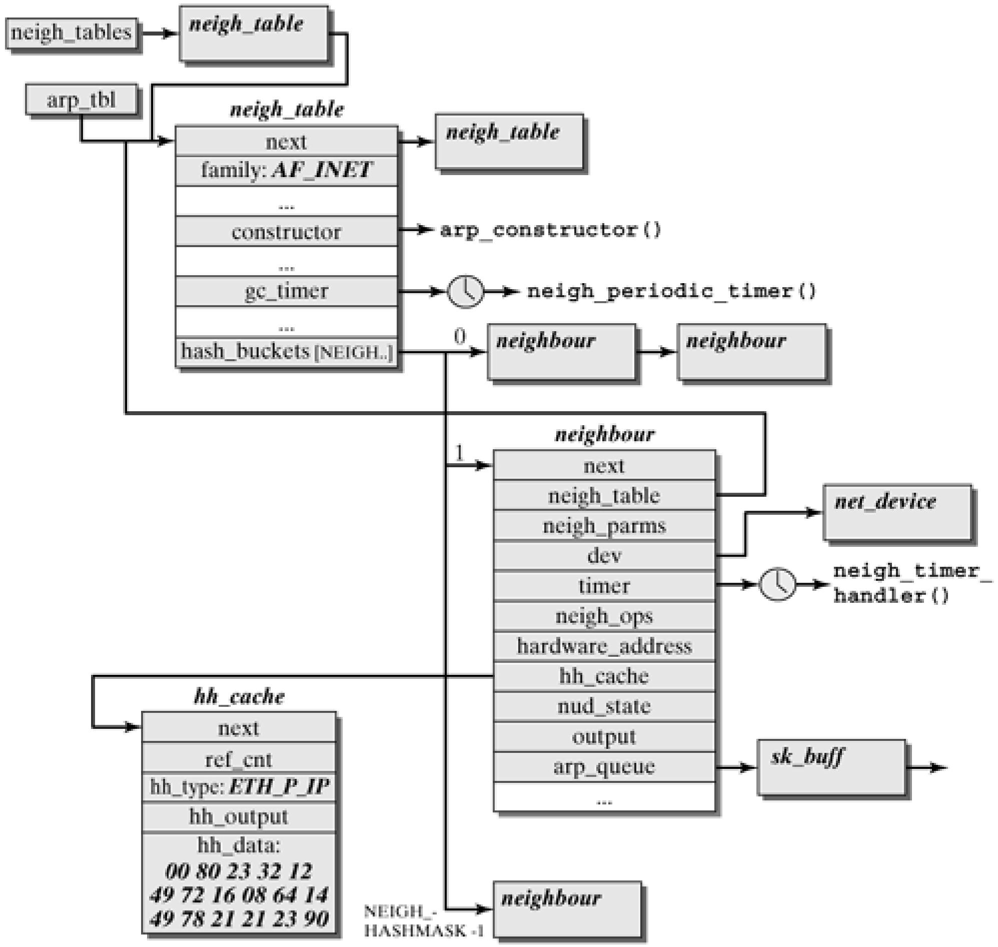
A `neighbour` represents a computer that is reachable over layer-2 services (i.e., directly over the LAN). Using the `neighbour` interface and the available functions, you can implement special properties of either of the two protocols (ARP and Neighbour Discovery). The following sections introduce the `neighbour` interface and discuss the ARP functions. Chapter 23 describes how Neighbor Discovery is implemented.

15.3.1 Managing Reachable Computers in the ARP Cache

As was mentioned earlier, computers that can be reached directly (over layer 2) are called neighbor stations in Linux. Figure 15-4 shows that they are represented by instances of the neighbour structure.

Figure 15-4. Structure of the ARP cache and its neighbor elements.

[View full size image]

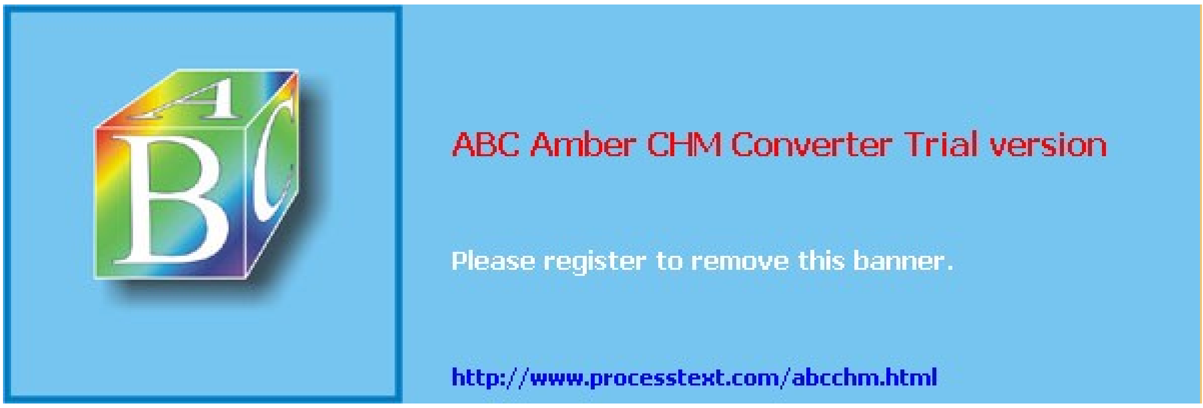


The set of reachable computers is managed in the ARP cache, which is organized in a hash table. The

◀ Previous Next ▶

Chapter 16. IP Routing

◀ Previous Next ▶

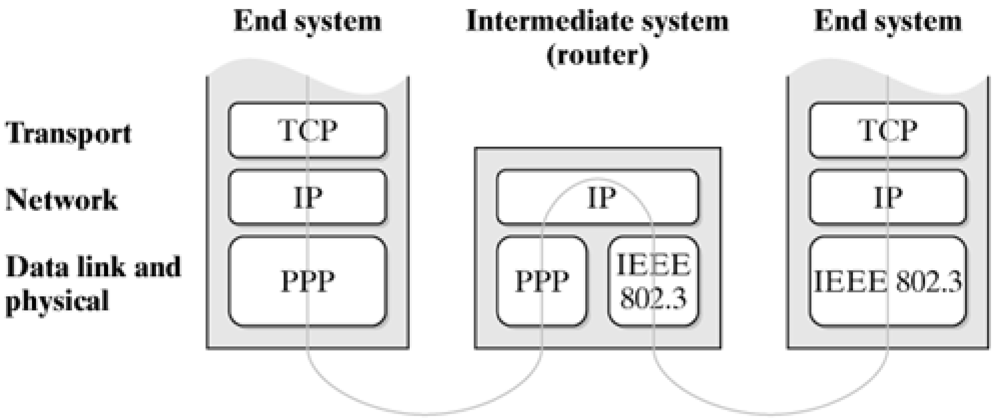


◀ Previous Next ▶

16.1 Introduction

One of the most important functions of the IP layer (the network layer of the TCP/IP protocol architecture) is to forward packets between communicating end systems across a number of intermediate systems. (See Figure 16-1.) The determination of the route that packets will take across the Internet and the forwarding of packets towards their destination is called routing.

Figure 16-1. Routing within the IP layer in the TCP/IP protocol architecture (protocols in the other layers are examples).



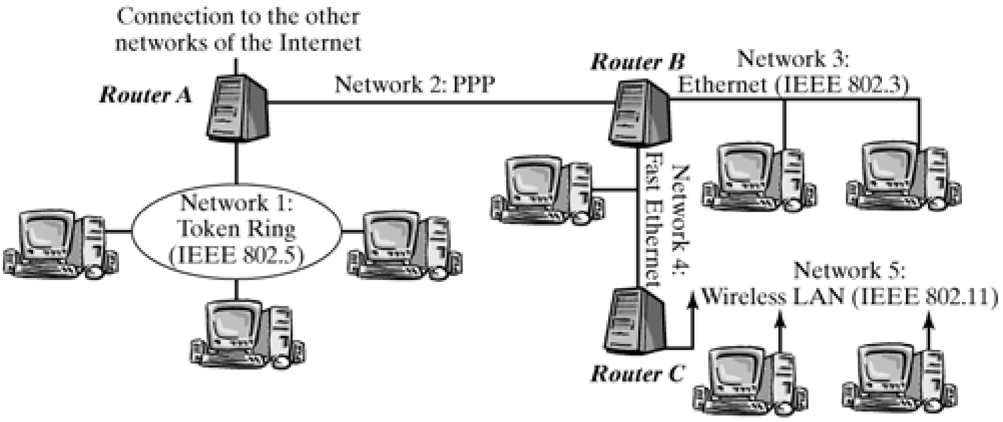
16.1.1 Networks and Routers

As was mentioned in Chapter 14, the Internet represents a network of networks. The physical subnetworks built by use of different layer-2 transmission technologies, such as Ethernet, can include a different number of nodes each? for example just two nodes connected over a point-to-point link. The IP layer interconnects these subnetworks to form a global network having millions of nodes.

Special nodes, which are integrated in all subnetworks that are connected in one place, are used to link these subnetworks; these nodes are called routers. Figure 16-2 shows an example with five local area networks, connected through three routers. Router A also connects the network to the rest of the Internet. The network layer abstracts from lower layers, so it is irrelevant for the communication implemented over IP that the end systems are connected to different LAN types.

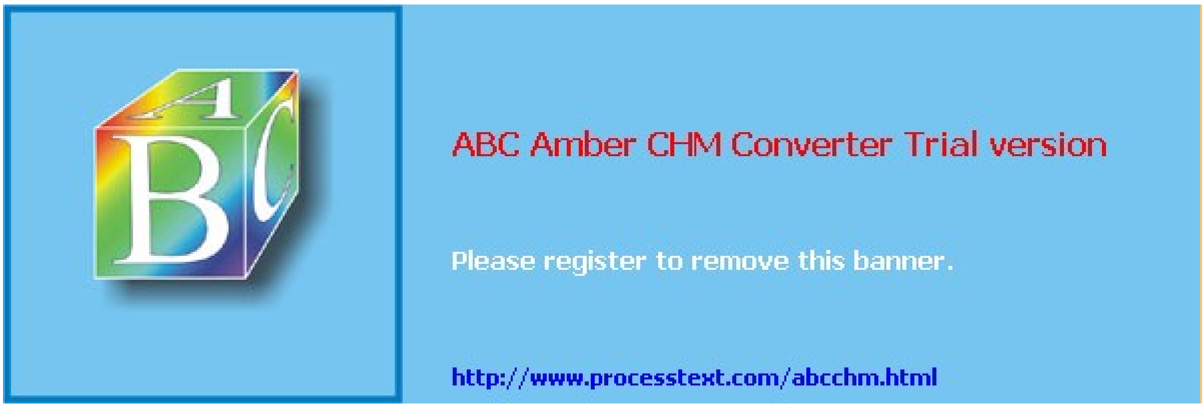
Figure 16-2. Routers interconnect networks.

[View full size image]


Connection to the other networks of the Internet
Router A
Network 2: PPP
Router B
Network 3: Ethernet (IEEE 802.3)
Network 1: Token Ring (IEEE 802.5)
Network 4: Fast Ethernet
Network 5: Wireless LAN (IEEE 802.11)
Router C

Routers are used both to link local area networks and to connect local area networks to the Internet. In addition, networks in the "core" of the Internet, which normally have a much larger geographic reach, are interconnected and linked to access networks through routers, or even built of direct links between routers ("two-node networks").

Routers are often especially designed for this purpose? so-called "dedicated routing devices." However, the Linux kernel also offers the required functionality to let you use a Linux system as a



◂ Previous Next ▸

16.2 Configuration

This section describes the options available to configure routing in Linux. First, this concerns the kernel configuration, which is used, for example, to determine whether advanced features, such as rule-based routing, should be integrated into the kernel. The options available for this configuration are described in Section 16.2.1. Second, you can also modify some routing parameters while the system is running. The setting options available for this in the `proc` file system are discussed in Section 16.2.2. Third, you have to add entries to routing tables and rule lists. The `ip` command, which is described in Section 16.2.3, is a good tool to manage such entries.

16.2.1 Configuring the Kernel

Some routing options can be set when you configure the Linux kernel, before it is compiled. All of them are in the networking options section and will be described briefly in this section below. In addition to the name of the preprocessor constant, which is defined when an option is activated, the label shown in the kernel configurator is given in double quotes. A prerequisite to being able to activate some of these options is that `CONFIG_INET` ("`TCP/IP` networking") should be enabled; without that, routing makes no sense, anyway.

- `CONFIG_NETLINK "Kernel/User netlink socket"`

  Rather than directly influencing the routing mechanism, this option activates the bidirectional netlink interface between the kernel and the user-address space, which is implemented with datagram sockets of the new protocol family, `PF_NETLINK`, and can be used to communicate with different kernel areas. The respective area is selected by an identifier, which is given instead of a protocol when you open the socket. Section 26.3.3 describes more details.

  In connection with routing, the `NETLINK_ROUTE` "protocol identifier" is important, and it can be used by activating the following option. This option is available only provided that `CONFIG_NETLINK` is active:

  - `CONFIG_RTNETLINK "Routing messages"`

    Routing rules and routing tables can be modified by using sockets of the `PF_NETLINK` protocol family and the `NETLINK_ROUTE` "protocol." This interface, which will also be called RT netlink interface below, is used in the ip configuration tool described in Section 16.2.3. Besides, by reading an RT netlink socket, you can "eavesdrop" on changes made to routing tables by other processes.

- `CONFIG_IP_ADVANCED_ROUTER "IP: advanced router"`

  This option has no direct effect; it represents a switch that allows you to select a number of additional options can be used to obtain much more control over the routing procedure. The options `CONFIG_NETLINK` and `CONFIG_RTNETLINK` are activated automatically when you select `CONFIG_IP_ADVANCED_ROUTER`.

  - `CONFIG_IP_MULTIPLE_TABLES "IP: policy routing"`

    This option links the file `fib_rules.o` into the kernel and enables the rule-based routing described in Section 16.1.6. If this option is disabled, then the kernel creates only two routing tables, `local` and `main`, and searches them in this order.

    The following additional options are available in connection with rule-based routing:

    `CONFIG_IP_ROUTE_FWMARK "IP: use netfilter MARK value as routing key"`

    This option allows you to include the `fwmark`, which can be added to certain packets by using packet filter rules (see Section 19.3.5), in the forwarding decision (i.e., you can specify different routes for packets with different packet filter marks). For example, you can make the route selection indirectly

◂ Previous Next ▸

16.3 Implementation

The following discussion divides the routing implementation in Linux into three functional units:

- As described in Section 14.2, `ip_route_input()` and `ip_route_output()` are the two functions invoked when IP packets are handled to run routing-specific tasks; they will be described in Section 16.3.4. These functions are also called "forwarding functions" in the following discussion.
- Routing rules and routing tables together form the so-called forwarding-information base (FIB). Whenever necessary, forwarding functions query the forwarding-information base; this action is also called a forwarding query or FIB request in the following discussion. An FIB request is initiated by calling the `fib_lookup()` function. The implementation of routing rules is strongly encapsulated within the FIB, so routing rules and routing tables will be discussed separately in Sections 16.3.1 and 16.3.2.
- Because consulting the FIB for each single IP packet received or sent would require too much time, there is an additional routing cache that stores the table entries used the most recently and allows fast access to these entries. Section 16.3.3 describes how this routing cache is implemented.

16.3.1 Routing Rules

As we described in Section 16.1.6, rule-based routing uses a set of rules to decide which routing tables should be searched in which sequence for a suitable entry to forward a packet and whether the packet may be forwarded at all. The rules are processed successively by ascending priority value until a decision can be made.

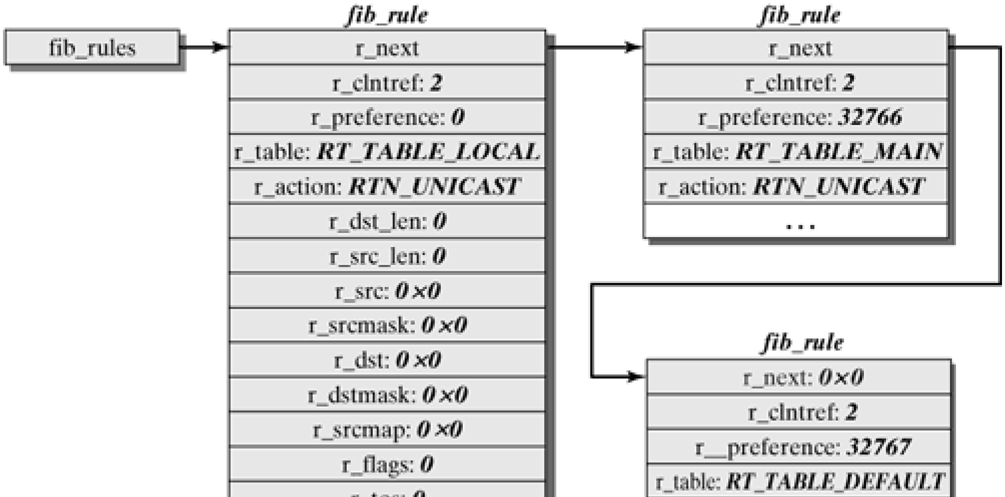
The entire implementation of the rules-processing method, including the data types used, is included in the `fib_rules.c` file. The rather narrow interface is described by some function prototypes and i nline functions in a common header file, `ip_fib.h`. If rule-based routing was disabled in the kernel configuration (`CONFIG_IP_MULTIPLE_TABLES` option; see Section 16.2.1), `then fib_rules.c` is not compiled. In this case, the "replacement functionality" (use of the two routing tables `local` and `main`, in this sequence) is fully handled by the inline functions in `ip_fib.h`.

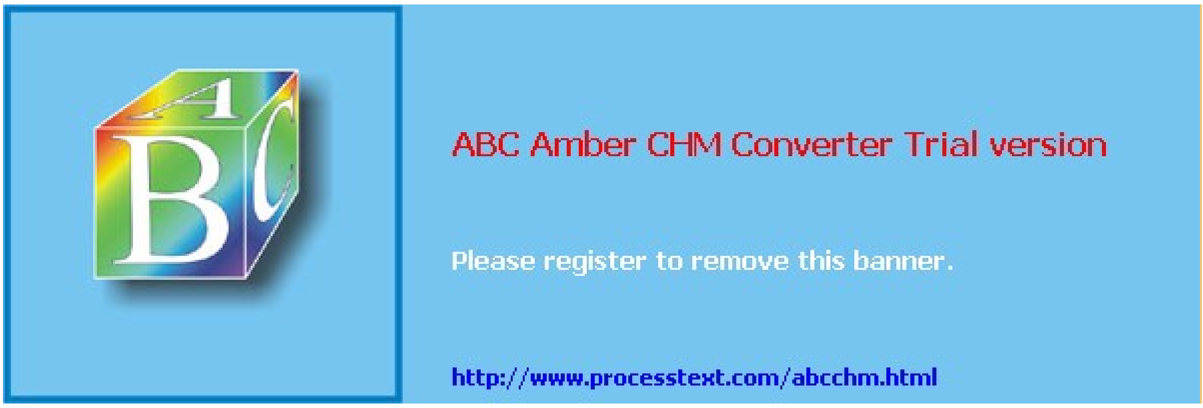
Data Structures

The set of rules is represented in the kernel by a linear list of `fib_rule` structures, sorted in ascending order by priority value and hooked into the static `fib_rules` variable. Initially, this list contains three entries: the `fib_rule` structures `default_rule, main_rule,` and `local_rule,` which are statically defined. Figure 16-6 shows this initial state of the rules list. A read-write spinlock called `fib_rules_lock` is used to regulate access to the list.

Figure 16-6. List with routing rules (initial state). Optional structure entries are not shown.

[View full size image]





◂ Previous Next ▸

Chapter 17. IP Multicast for Group Communication

The history of telecommunication was characterized mainly by two technologies in the past hundred years (before the Internet era began): telephony, and radio and television broadcasting. These two technologies cover two fundamentally different communication areas or needs:

- Individual communication (unicast): Connections between two communication partners, where the direction of data exchange can be unidirectional (simplex) or bidirectional (duplex).

- Mass communication (broadcast): One station sends data to all stations reachable over a medium, where data distribution is normally unidirectional.

After these two technologies, the Internet followed as the third communication technology, changing the telecommunication world. Though the Internet was initially designed for individual communication, the protocols and mechanisms added at the beginning of the nineties introduced a new communication form: group communication (multicast). Multicast makes possible an efficient data distribution to several receivers. By contrast with the mass communication of (radio) broadcasting, where data is distributed to all participants within one single transmission medium, group communication delivers data units only to those receivers explicitly interested in this data. In addition, group communication in the Internet (IP multicast) enables each Internet computer to send data directly to the members of a multicast group.

Consequently, in the designing of mechanisms and protocols, two specific tasks can be deduced for the functionality of group communication in Internet systems:

- managing memberships in communication groups; and

- efficient distribution of data packets to all members of a group.

The first task is solved by the Internet Group Management Protocol (IGMP), which has to be supported by each multicast-capable Internet computer. Section 17.3 introduces IGMP and its implementation in Linux systems. For the second task, we have to distinguish between end system and forwarding systems. Section 17.4 will discuss how both types are supported in Linux. As with Internet routing, group communication also separates clearly between forwarding and routing. There are different multicast routing algorithms, including the Distance Vector Multicast Routing Protocol (DVMRP), which will be introduced in Section 17.5.2 as a representative example for these algorithms, using the `mrouted` daemon.

◂ Previous Next ▸

◀ Previous Next ▶

17.1 Group Communication

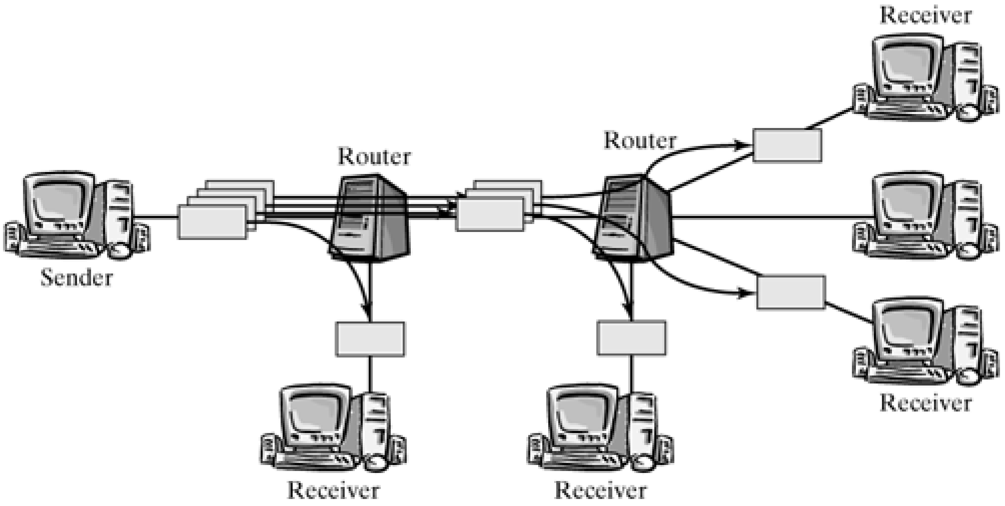
Before we introduce the details of IP multicast in Linux, the following sections give a brief summary of the three communication forms: unicast, broadcast, and multicast.

17.1.1 Unicast

Unicast is the classic form of communication between two partners? point to point. In the context of this book, this means that two computers communicate with each other only. When a unicast communication service is used to transmit data to several receivers, then this has to be done independently of one another in several transmit processes. This means that the cost for the data transport increases in proportion to the number of receivers, as shown in Figure 17-1.

Figure 17-1. In unicast communication, the packet is sent to each receiver separately.

[View full size image]



Naturally, if there is a large number of receivers, this cost leads to an extreme load on the network, and so this technique is unsuitable for the distribution of large data volumes, such as multimedia data. Broadcast communication represents a better solution in some cases.

17.1.2 Broadcast

Broadcasting means that all participants in a communication network that can be reached over a specific medium receive the distributed data packets, regardless of whether they are interested in it. Examples for broadcast communication include the broadcasting of television and radio broadcasting programs, and advertisements in the mailboxes of homes.

At first sight, broadcast communication looks expensive. However, a closer look reveals that it is supported by the network technologies, especially in local area networks (LANs). In fact, each communication is a broadcast communication in local area networks, because the local network technologies (Ethernet, token ring, etc.) are broadcast media, where data packets are distributed to all stations. When a packet is received, the MAC destination address is checked to see whether the packet should be further handled by that station. This means that broadcast communication is very easy in local area networks. In fact, it is sufficient to send a packet to the network, so that all stations can receive it.

However, as with advertisements in mailboxes, not everybody will want to receive a broadcast packet they are not interested in. For this reason, though it is simple to broadcast data to a group, this approach is a burden for stations not interested in this data and reduces their performance. This holds also true for wide area network (WAN) traffic: Where point-to-point connections prevail, the benefit of t he simplicity of broadcasting can easily turn into a heavy burden for the networks [Tan97].

17.1.3 Multicast

Multicast communication offers a solution to the problem described in the previous section. It enables a targeted transmission of data packets, in contrast to n single transmissions in unicast, yet it prevents

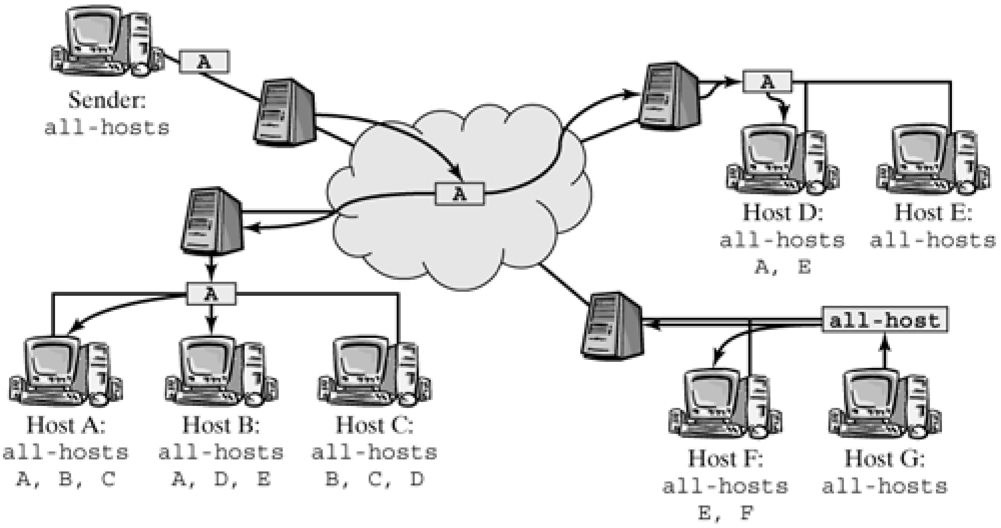
◀ Previous Next ▶

17.2 IP Multicast

IP multicast extends the unicast service of the Internet Protocol to the capability of sending IP packets to a group of Internet computers. This is realized more effectively than sending single unicast packets to the members of a group. The sender addresses the members of a group by a group address, the so-called IP multicast address. (See Section 17.2.1.) Normally, the sender doesn't know who is currently a member of a group, how many members are subscribed to a group, or where these members are located. IP multicast is one of the few implementations of the principle of group communication. Another network technology that also supports group communication is ATM. Figure 17-3 shows the IP multicast scenario in the Internet.

Figure 17-3. IP multicast scenario in the Internet.

[View full size image]



addresses for multicast groups: We basically have to distinguish between multicast communication on the MAC layer and those on the network layer (Internet Protocol). In local area networks, multicast is normally supported by the underlying technology. In this case, multicast packets are transmitted over a broadcast-enabled network, and the connected computers use the group address to decide whether they want to receive the data. Section 17.4.1 describes in detail how multicast is supported in local area networks. In contrast, multicast communication on the IP layer (i.e., between the routers in the Internet) is much harder to implement. One of the most important functions is provided by multicast routing protocols, which organize the efficient distribution of data.

The separation between multicast in the local domain and in the routed network domain can be seen not only in how data are forwarded (data path), but also in how groups are managed. Joining and leaving groups is handled by the Internet Group Management Protocol (IGMP); routers distribute their group information over multicast routing protocols. An end system tells its local multicast router only the IP address of the group it wants to join. The router will then have to find out how it can get the multicast data from the Internet. Joining and leaving of groups for computers in a local area network are handled by the Internet Group Management Protocol, which will be introduced in Section 17.3.

So-called multicast distribution trees are built to distribute multicast packets between routers across the entire network. The data packets are then distributed along these trees to the individual receivers and local area networks. These distribution trees are built by multicast routing protocols (e.g., the Distance Vector Routing Protocol (DVMRP) and Multicast OSPF (MOSPF)).

As in routing in the Internet, the control path and the data path are also separated here, as can be clearly seen in the implementation under Linux. The data path (i.e., forwarding and replicating of multicast packets) is defined by the information in the multicast routing table. The information in the routing table is procured over the control path, by the multicast routing protocols and IGMP. In addition to better structuring, another benefit of keeping the two mechanisms separate is that we can use different routing protocols. In fact, several protocols are currently available in Linux. Section 17.5.2 uses the DVMRP protocol and its implementation in the `mrouted` daemon as a representative example.

◂ Previous Next ▸

17.3 Internet Group Management Protocol (IGMP)

The Internet Group Management Protocol (IGMP) is used to manage group memberships in local area networks. A multicast router should know all groups having members in the local area network. Accordingly, the Multicast Routing Protocol subscribes packets for these groups. The router does not have to know exactly who in the local area network belongs to a group. It is sufficient for the router to know that there is at least one receiver. The reason is that, when the router transports a packet to the local area network, all stations subscribed to this group receive it automatically.

To avoid unnecessary data transmissions, the router checks periodically for multicast groups that are still desired. For this purpose, it sends a membership query to all local computers (i.e., to the all-hosts group) within a specific time interval (approximately every two minutes). Each computer currently interested in a group should then return a reply for each of its groups to the router. As was mentioned earlier, the router is not interested in knowing who exactly is a member of a group; it is interested only in knowing whether there is at least one member in the LAN. For this reason, and to prevent all computers from replying at the same time, each computer specifies a random delay time, and it will reply when this time expires. The first computer to reply sends its message to the router and to all other local computers of the specified group. Cleverly, it uses the multicast group address for this message. This means that the other computers learn that the router has been informed, so that they don't have to reply. The router has to continue forwarding data for this group from the Internet to the local area network.

Naturally, if a computer wants to join a group, it does not have to wait for a membership query; it can inform the router immediately about the group it wants to join. Section 17.3.3 describes how exactly the IGMP protocol works.

In addition to the tasks discussed above, IGMP is used for other things. The following list summarizes everything the IGMP is used for:

- Query a multicast router for groups desired in a LAN.
- Join and leave a multicast group.
- Exchange membership information with neighboring or higher-layer multicast routers.

17.3.1 Formatting and Transporting IGMP Packets

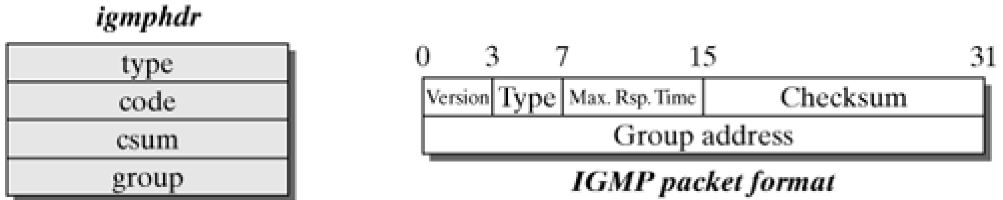
IGMP messages are transported in the payload field of IP packets, and the number 2 in the Protocol field of the IP packet header identifies them as IGMP messages. They are always sent with the TTL value one, which means that they cannot leave the area of a subnetwork and so means that IGMP manages group memberships only within a subnetwork. To distribute this information beyond these limits, we have to use multicast routing protocols.

Figure 17-5 shows the format of IGMP packets; it includes the following fields:

- Version: Number of the IGMP version used.
- Type: Type of the IGMP message.
- Max. Response Time: This field is used differently, depending on the IGMP version. (See Section 17.3.2.)
- Checksum: Checksum of the IGMP message.

Figure 17-5. The IGMP packet format and its representation in the Linux kernel.

[View full size image]



◂ Previous Next ▸

17.4 Multicast Data Path in the Linux Kernel

This section describes how multicast data packets are processed in the Linux kernel. To get a good insight into matters, we will first explain the path a multicast packet takes across the kernel, then discuss different aspects of the data path. We will begin on the MAC layer, to see how multicasting is supported in local area networks and to introduce the following IP multicast concepts:

- virtual network devices,
- multicast routing tables, and
- replicating of data packets.

As we introduce the implementation, we will emphasize differences between multicast-capable end systems and multicast routers.

17.4.1 Multicast Support on the MAC Layer

In general, IEEE-802.x LANs are broadcast-enabled: each data packet is sent to each participant. Each network adapter looks at the MAC destination address to decide whether it will accept and process a packet. This process normally is handled by the network adapter and doesn't interfere with the central processor's work. The central processor is stopped by an interrupt only when the adapter decides that a packet has to be forwarded to the higher layers. This means that the filtering of packets in the network adapter take load off the CPU and ensures that it will receive only packets that are actually addressed to the local computer.

Filtering undesired MAC frames works well in the case of unicast packets, because each adapter should know its MAC address. However, how can the card know whether the computer is interested in the data of a group when a multicast packet arrives? In case of doubt, the adapter accepts the packet and passes it on to the higher-layer protocols, which should know all subscribed groups. The next question is whether multicast packets use the MAC address at all. The MAC format supports group addresses, but how are they structured?

There is a clever solution for IP multicast groups to solve the problems described above. On the one hand, this solution prevents broadcasting of multicast packets; on the other hand, it concurrently filters IP groups on the MAC layer. The method, described here, is simple, and it relieves the central processing unit from too many unnecessary interrupts.

IP multicast packets are packed in MAC frames before they are sent to the local area network, and they contain a MAC group address. The MAC address is selected so that it gives a clue about which multicast group the packet could belong to. Figure 17-9 shows how this address is structured; it contains the following elements:

- The first 25 bits of the MAC address identify the group address for IP multicast.

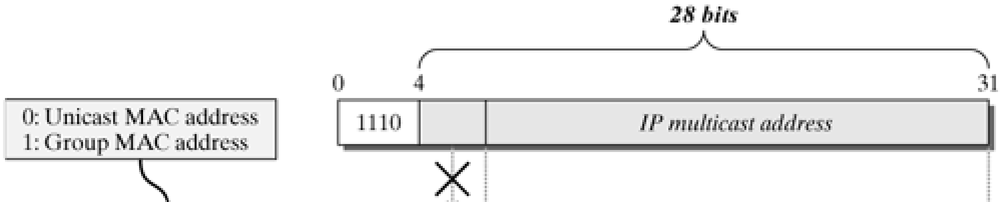
  The first byte (`0x01`) shows that the address is a group MAC address, where the last bit is decisive. Notice that the address shown in Figure 17-9 is represented in the network byte order.

  The next 17 bits (`0x005E`) state that the MAC packet carries an IP multicast packet. The identifier here would be different for other layer-3 protocols.

- The last 23 bits carry the last 23 bits of the IP multicast address.

Figure 17-9. Mapping an IP multicast group address to an IEEE-802 MAC address.

[View full size image]



◂ Previous Next ▸

17.5 Multicasting in Today's Internet

Multicasting was a thing unheard of at the advent of the Internet, and neither group addresses nor protocols to manage groups or multicast routing were available. In fact, the most important prerequisites to implementing an efficient group communication service were missing. The Internet was a pure unicast network.

Several proposals in this field were made [Deer91] when the Internet community had started to think that such a service was necessary, at the beginning of the nineties. Eventually, IP multicast was born when the Internet Group Management Protocol and the address class D were standardized. In addition, multicast routing protocols were proposed, so that nothing was actually impeding the introducing of the new communication form. However, though the Internet had evolved into an enormous global network during the last twenty years, it was still a unicast network, and gradually each system connected to the Internet would have had to be extended to IP multicast support. This change would certainly have taken several years to complete. In addition, the new technology had not yet been tested extensively. Consequently, a decision was made to build a multicast test network within the unicast Internet, the so-called MBone (Multicast Backbone On the Internet), rather than converting to multicast from scratch.

17.5.1 The Multicast Backbone (MBone)

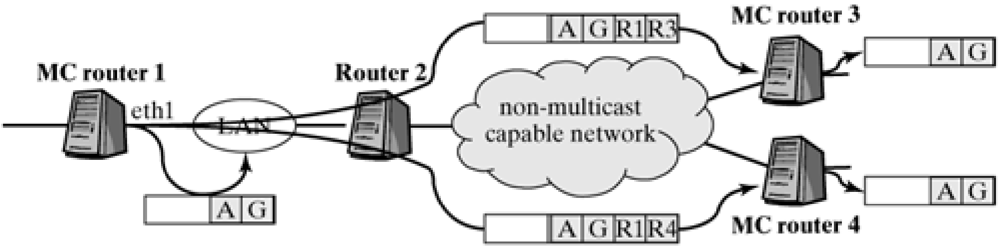
The Internet Engineering Task Force (IETF) ran a pilot transmission session to officially introduce MBone in March 1992. Since then, more than 10,000 subnetworks have been connected to this network worldwide. MBone enables the connected multicast-enhanced subnetworks to run IP multicasting over the existing Internet, even though the Internet itself is not multicast capable.

The solution offered by MBone is relatively simple: It builds a virtual multicast network over the conventional Internet, which understands only unicasting, and connected systems communicate over multicast-capable routers (multicast routers). As soon as there is a nonmulticast network between them, multicast routers bridge this situation by a so-called IP-in-IP tunnel. This tunnel consists of a unicast connection used to transport multicast traffic. For this purpose, the multicast router packs a multicast packet into another IP packet at the beginning of the tunnel and sends it as a normal unicast IP packet over the network to the tunnel output. The multicast router at that end of the tunnel removes the outer unicast packet and sends the multicast packet to the multicast-capable network.

This method led to the formation of many multicast-capable islands interconnected by tunnels over the conventional Internet. Figure 17-14 shows an example for the basic MBone architecture. Technically, MBone is a virtual overlay network on top of the Internet. Similar overlay networks have been built to study other Internet technologies, including 6Bone (Six-Bone) for IPv6 and QBone to study quality of service (QoS) mechanisms.

Figure 17-14. MBone consists of multicast islands connected by tunnels.

[View full size image]



17.5.2 Accessing MBone Over the `mrouted` Daemon

The `mrouted` daemon is a tool you can use to connect to MBone. It enables you to build tunnels to other MBone nodes and ensure connectivity. In addition, this daemon enables multicast routing for multicast packets within or at the boundaries of a multicast network. The standard implementation of `mrouted` in UNIX uses the Distant Vector Multicast Routing Protocol (DVMRP; see Section 17.5.3).

Like all daemons, `mrouted` operates in the user-address space and can be started and stopped at

◀ Previous Next ▶

17.6 Multicast Transport Protocols

So far, we have actually discussed only unreliable and connectionless multicast transmissions based on UDP. This type of transmission is generally the most frequently used application of multicast, mainly because it is much easier to handle. Nevertheless, there are application cases for connection-oriented and reliable multicast communication, and so extensive interesting research work is undertaken in this field.

Because the tasks involved in the reliable and connection-oriented transmission of multicast data correspond mainly to the tasks of a transport protocol and these work on top of the IP Multicast, a layer-3 service, the protocols developed so far are normally called multicast transport protocols. The most important tasks of a transport protocol, including connection management, flow control, error correction, and congestion control, are relatively complex and expensive for unicast communication, and point-to-multipoint communication adds special problems to this situation. For example, consider the sender implosion problem, which occurs when many receivers return acknowledgements for received data packets to the sender, overloading the sender with an enormous data volume.

We will not discuss multicast transport protocols any further at this point, because there is currently no protocol used as a standard under Linux. We do list a few protocols and research projects here. Some of these protocols have been implemented and evaluated. However, none of these protocols is e specially suited for all multicast applications; each one has specific benefits and drawbacks.

- Real-Time Transport Protocol (RTP)? for real-time and multimedia applications.
- Scalable Reliable Multicast (SRM)? is currently used by the White Board tool.
- Uniform Reliable Group Communication Protocol (URGC)? supports reliable and in-order communication.
- Muse? an application-specific protocol for multicast news.
- Multicast File Transfer Protocol (MFTP)? works much like the File Transfer Protocol (FTP).
- Local Group Concept (LGC)? uses a hierarchy of local groups to prevent sender implosion.

◀ Previous Next ▶

◀ Previous Next ▶

Chapter 18. Using Traffic Control to Support Quality of Service (QoS)

◀ Previous Next ▶

◀ Previous Next ▶

18.1 Introduction

In the Linux world, the term traffic control represents all the possibilities to influence incoming and outgoing network traffic in one way or another. In this context, we normally distinguish between two definitions, although it is often difficult to draw a clear line between the two:

- Policing: "Policing" means that data streams are monitored and that packets not admitted by a specified strategy (policy) are discarded. Within a networked computer, this can happen in two places: when it is receiving packets from the network (ingress policing) and when it is sending packets to the network.

- Traffic shaping: "Traffic shaping" refers to a targeted influence on mostly outgoing traffic. This includes, for example, buffering of outgoing data to stay within a specified rate, setting priorities for outgoing data streams, and marking packets for specific service classes.

The traffic-control framework developed for the Linux operating system creates a universal environment, which integrates totally different elements for policing and traffic shaping that can be in terconnected. These elements can even be dynamically loaded and unloaded as a module during active operation. We describe this framework in detail below, but limit the discussion of the implementation of elements in this framework to a single example. Subsequently, we will describe configuration options in the user space.

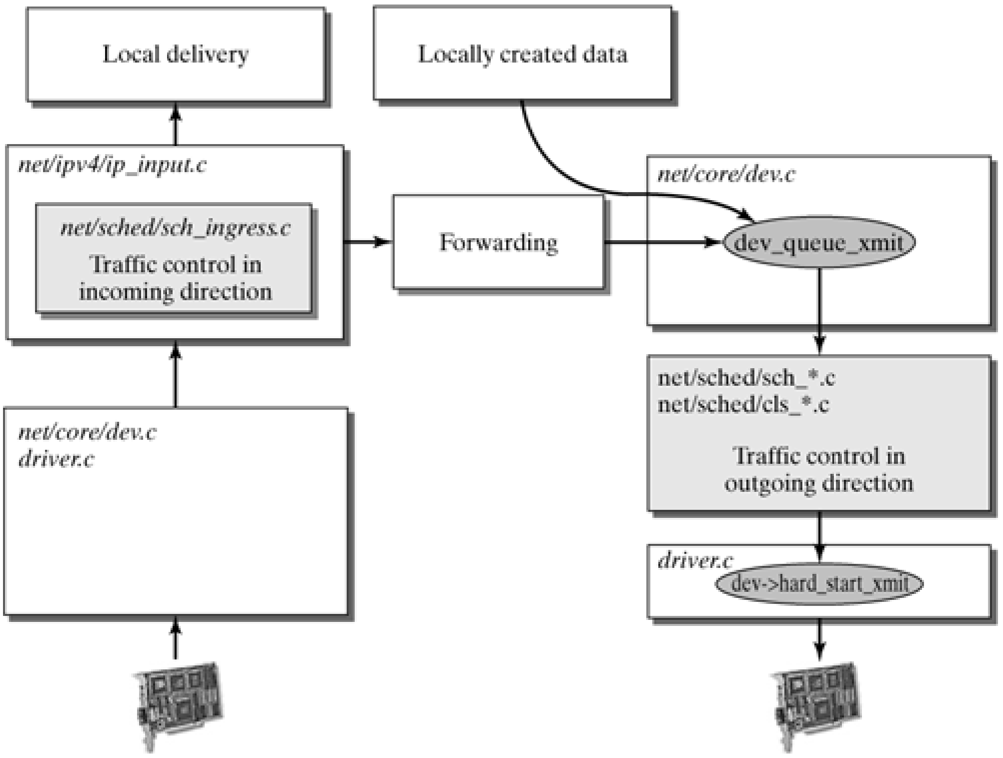
◀ Previous Next ▶

18.2 Basic Structure of Traffic Control in Linux

Figure 18-1 shows where traffic control is arranged in the Linux kernel. Traffic control in the incoming direction is handled by the functions from the file `net/sch/sch_ingress.c` before incoming packets are passed to higher protocol layers or forwarded over other network cards within the kernel.

Figure 18-1. Traffic control in the Linux kernel.

[View full size image]



The largest part of traffic control in Linux occurs in outgoing direction. Here, we can use and interlink different elements for policing and traffic shaping.

◂ Previous Next ▸

18.3 Traffic Control in the Outgoing Direction

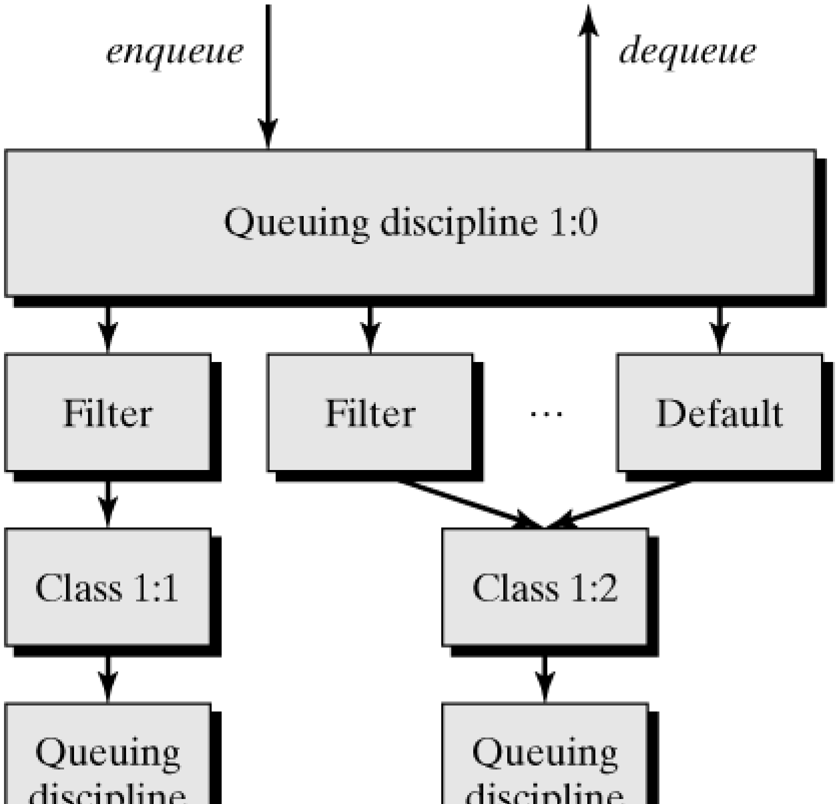
The traffic-control framework defines three basic elements:

- Queuing discipline: Each network device is allocated to a queuing discipline. In general, packets to be sent are passed to a queuing discipline and sorted within this queue by specific rules. During a search for packets ready to be sent, these packets can be removed no earlier than when the queuing discipline has marked them as ready for transmission. The algorithm used within a queuing discipline remains invisible to the outside. Examples for queuing disciplines include simple FIFO buffers and token buckets. More elaborate queuing disciplines can also manage several queues. Queuing disciplines are defined in files with names beginning with `sch_` (in the `net/sched` directory).
- Classes: Queuing disciplines can have several interfaces, and these interfaces are used to insert packets in the queue management. This allows us to distinguish packets by classes. Within one single queue discipline, we could allocate packets to different classes (e.g., to handle them with different priorities). Classes are defined within the queuing discipline (i.e., also in files with names beginning with `sch_`).
- Filters: Filters are generally used to allocate outgoing packets to classes within a queuing discipline. Filters are defined in files with names beginning with `cls_`.

Much as with a construction kit, single elements can be connected, even recursively: Other queuing disciplines, with their corresponding classes and filters, can be used within one single queuing discipline.

Figure 18-2 shows an example for the resulting traffic-control tree. On the outside, we first see only the `enqueue` and `dequeue` functions of the upper queuing discipline. In this example, packets passed via the function `enqueue ()` are checked one after another by the filter rules and allocated to the class visited by the filter for the first time. If none of the filter rules matches, then a default filter can be used to define an allocation system. Behind the classes there are other queuing disciplines. Because this is a tree, we also speak of the parent of a queuing discipline. For example, the queuing discipline 1:0 is a so-called outer queuing discipline and the parent of the classes 1:1 and 1:2. The queuing disciplines 2:0 and 3:0 are also called inner queuing disciplines.

Figure 18-2. Example for a tree consisting of queuing disciplines, classes, and filters.



◀ Previous Next ▶

18.4 Kernel Structures and Interfaces

The interfaces available for queuing disciplines and filters are mostly independent of the functionality available within an element.

18.4.1 Handles

All elements within the traffic-control tree can be addressed by 32-bit identifiers called `handles`. For example, the instances of the queuing disciplines discussed further below are marked with 32-bit identifiers, divided into a major number and a minor number. However, these numbers have nothing to do with the major and minor numbers for device files. These identifiers are unique for each network device, but they can occur more than once for several network devices.

In contrast, the minor number for a queue discipline is always null, except for input queuing discipline number `ffff:fff1 TC_H_INGRESS` (in `include/linux/pkt_sched.h`) and the top queue of output queuing discipline number `ffff:ffff TC_H_ROOT`. Major numbers are assigned by the user and are in the range from `0x0001` to `0x7fff`. If the user specifies major number `0`, then the kernel allocates a major number between `0x8000` und `0xffff`.

For classes, the major number corresponds to the associated queuing discipline, while the minor number specifies the class within that queuing discipline. In this case, the minor number can be in the range from `0x0` to `0xffff`. Minor numbers are unique only within all classes of a queuing discipline.

`include/linux/pkt_sched.h` defines several macros you can use to mask major and minor numbers.

18.4.2 Queuing Disciplines

The functions supplied by a queuing discipline are defined in the `Qdisc_ops` structure in `include/net/pkt_sched.h`:

```
struct Qdisc_ops {
        struct Qdisc_ops          *next;
        struct Qdisc_class_ops    *cl_ops;
        char                      id[IFNAMSIZ];
        int                       priv_size;

        int                       (*enqueue) (struct sk_buff *, struct
Qdisc *);
        struct sk_buff *          (*dequeue) (struct Qdisc *);
        int                       (*requeue) (struct sk_buff *, struct
Qdisc *);
        int                       (*drop) (struct Qdisc *);

        int                       (*init) (struct Qdisc *, struct rtattr
*arg);
        void                      (*reset) (struct Qdisc *);
        void                      (*destroy) (struct Qdisc *);
        int                       (*change) (struct Qdisc *, struct rtattr
*arg);

        int                       (*dump) (struct Qdisc *, struct sk_buff
*);
};
```

The first four entries are a link to a list (`struct Qdisc_ops *next;`), a reference to the class-related operations (`struct Qdisc_class_ops *cl_ops`), which will be described later. They represent an identifier (`char id [IFNAMSIZ]`) and values used internally.

The following functions are available externally:

◂ Previous Next ▸

18.5 Ingress Policing

The file `net/sched/sch_ingress.c` implements a queuing discipline designed for ingress policing. Its structure is similar to that of other queuing disciplines, and the exported functions are similar to the functions described in the previous section.

However, rather than buffering packets, this queuing discipline classifies packets to decide whether a packet will be accepted or discarded. This means that the queuing discipline actually assumes a firewall or Netfilter functionality. This functionality also reflects in the return values of the `enqueue()` function, which are converted to Netfilter return values, as shown in the following excerpt from the function `ingress_enqueue()` (`net/sched/sch_ingress.c`):

```
case TC_POLICE_SHOT:
     result = NF_DROP;
     break;
case TC_POLICE_RECLASSIFY: /* DSCP remarking here ? */
case TC_POLICE_OK:
case TC_POLICE_UNSPEC:
default:
     result = NF_ACCEPT;
     break;
```

First, the function `register_qdisc()` registers the functions of the queuing discipline with the network device. Subsequently, the function `nf_register_hook()` hooks them into the hook `NF_IP_PRE_ROUTING`.

Next, additional filters can be appended to this particular queuing discipline. These filters can access functions from `net/sched/police.c` to check on whether a data stream complies with a token bu cket.

◂ Previous Next ▸

◂ Previous Next ▸

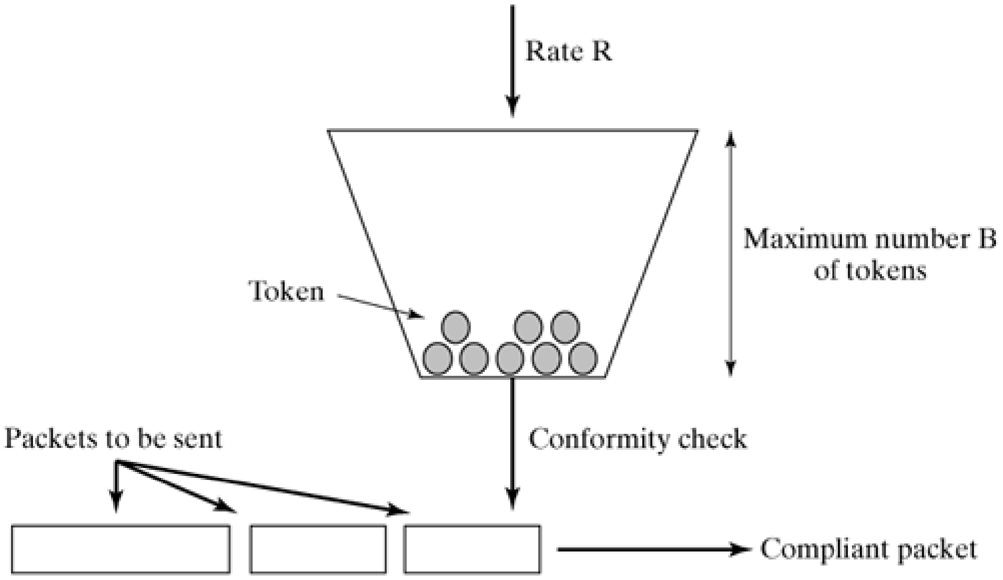
18.6 Implementing a Queuing Discipline

This section describes how we can implement a queuing discipline. We will use the token-bucket filter as an example, because it represents a fundamental element of many traffic-shaping approaches.

18.6.1 The Token-Bucket Filter

A token-bucket filter is used to control and limit the rate and burst (when a specified data rate is briefly exceeded) of data streams. Figure 18-4 illustrates the basic idea of a token bucket. In this model, the determining parameters are the rate, R, at which a token bucket is filled with tokens, and the maximum number of tokens, B, this token bucket can hold. Each token represents a byte that may be sent. Subsequently, the token bucket declares a packet to comply with the rate and burst parameters, if the number of tokens in the token bucket corresponds at least to the length of the packet in bytes.

Figure 18-4. Model of a token bucket.



If a packet is compliant, it may be sent. Subsequently, the number of tokens in the token bucket is reduced by a number corresponding to the packet length. If a noncompliant packet is deleted immediately, then the token bucket runs a traffic-policing process. In contrast, if the packet is held back until sufficient tokens have accumulated in the token bucket, we talk of traffic shaping.

A real-world implementation will realize this model differently, so that the computing cost is less, though the result is the same. It would not make sense to increment a counter representing the number of tokens several times per second, even when there is no packet to send. Instead, computations are made only provided that a packet is ready to be sent and waiting at the input of the token bucket. In this case, we can compute how many tokens have to be present in the token bucket at that point in time. To do this computation, we need to know when the last packet was sent and what the filling level of the token bucket was after that. The current number of available tokens is calculated from the sum of tokens available after the last transmission, plus the tokens arrived in the meantime (i.e., plus the interval, multiplied by the rate, R). Notice that the number of available tokens can never be larger than B. If the number of tokens computed in this way corresponds to at least the length of the waiting packet, then this packet may be sent. Otherwise, instead of sending the packet, a timer is started. This timer expires when more packets can be sent as a sufficient number of tokens has arrived. The timer has to be initialized to an appropriate interval, which can be easily calculated from the number of tokens still missing and the rate, R, at which the bucket is filled with more tokens.

Such a token-bucket filter is implemented within the traffic-control framework in the file `net/sched/sch_tbf.c`. However, this is an extension (i.e., a dual token bucket). More specifically, two token buckets are arranged back to back, in a frequently used arrangement, to guarantee a mean rate and limit bursts. The first token bucket is set to a rate, R, corresponding to the desired mean data rate, and the second token bucket is set to the peak rate and a significantly smaller number of tokens,

18.7 Configuration

This section describes how the traffic-control elements are configured from within the user space. To configure traffic-control elements, the `tc` tools are used. This toolset is a command-line configuration program (available in [Kuzn01] as part of the `iproute2` package). In addition, the RT netlink interface is used to pass configuration information to the kernel.

18.7.1 The RT Netlink Interface

The RT netlink interface is fully described in Chapter 26. For the purposes of this section, it is sufficient to know that the RT netlink interface is used to pass a pointer to the `rtattr` (in `include/linux/rtnetlink.h`) structure to the `init()` or `change()` functions of the traffic-control framework. The function `rtatr_parse` (`net/core/rtnetlink.c`) can be used to structure the data passed, and various macros, including `RTA_PAYLOAD` und `RTA_DATA` (`include/linux/rtnetlink.h`), can be used to print this information. The `tcmsg` (`include/linux/rtnetlink.h`) structure defines traffic-control messages that can be sent over the RT netlink interface from within the user space.

18.7.2 The User Interface

The `tc` program provides a command-line user interface to configure the Linux traffic control. This tool is available from [Kuzn01].

The `tc` tool enables you to set up and configure all elements of the traffic-control framework discussed here, such as queuing disciplines, filters, and classes. To be able to use the Differentiated Services support in Linux, we first have to set the entry `TC_CONFIG_DIFFSERV=y` in the `Config` file in the `iproutes/tc` directory. If the kernel version and the version of your `tc` tool match, then calling `make` in the same directory should enable you to compile successfully.

Depending on the element we want to configure, we now have to select the appropriate element, together with additional options:

```
Usage: tc [ OPTIONS ] OBJECT { COMMAND | help } where OBJECT :=
{ qdisc | class | filter }
       OPTIONS := { -s[tatistics] | -d[etails] | -r[aw] | -b[atch] file }
```

A detailed description of all additional options would go beyond the scope and volume of this book. You can use the `help` command (e.g., `tc qdisc add tbf help`) to easily obtain information. In addition, you can find an overview of ongoing work in the field of more comfortable user interfaces in [Alme01].

◀ Previous Next ▶

Chapter 19. Packet Filters and Firewalls

◀ Previous Next ▶

◂ Previous Next ▸

19.1 Introduction

Each network packet handled by a Linux computer passes a number of distinctive points within the network implementation on its way through the Linux kernel before it either is delivered to a local process or leaves the computer for further routing. Direct access to the packet stream in the kernel opens up a large number of ways to manipulate packets, which are also suitable for implementing a security strategy in the network. For example, functions were built into the routing code early in the course of the Linux development. These functions allow the system administrator to influence how packets are handled, depending on their source and destination addresses. In addition to the pure filtering function, which lets you drop certain packets completely, this also includes more complex manipulations, including address-conversion mechanisms (Network Address Translation? NAT) or the support of transparent proxies. After its introduction in the form of `ipfwadm` in Linux Version 1.2, this packet-filter code later underwent two complete revisions to ensure better manageability, extension of the control options, and better integration of additional functionality (e.g., NAT). This chapter discusses the differences between the packet-filter architecture of the current Linux Version 2.4 and that of the previous Linux Version 2.2.

19.1.1 The Functional Principle of a Firewall

In its original meaning, the term firewall denotes a fire-resistant wall constructed to prevent the spread of fire. In connection with computer networks, a firewall is a protection mechanism used in a specific and exactly limited network (e.g., a corporate intranet) at a transition point from a neighboring network (generally to the Internet) to protect the intranet against dangers from the outside.

A firewall consists normally of two types of components:

- Packet filters are normally implemented in routers and monitor the entire network traffic flowing through these routers. These routers use a well-defined set of rules (e.g., address information contained in a packet header) to decide which packets can pass and which will be dropped.

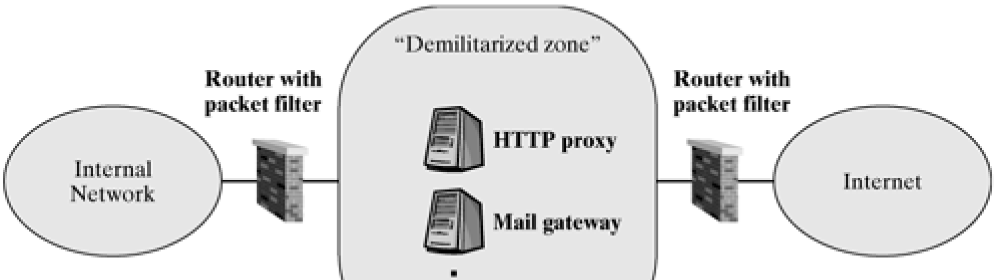
  In the case of IP networks, packet-filter rules normally refer at least to the IP source and destination addresses, the transport protocol (TCP or UDP), the TCP or UDP source and destination ports, and some TCP flags (for TCP; particularly the SYN flag, which can be used to see whether a packet is a connection-establishment request).

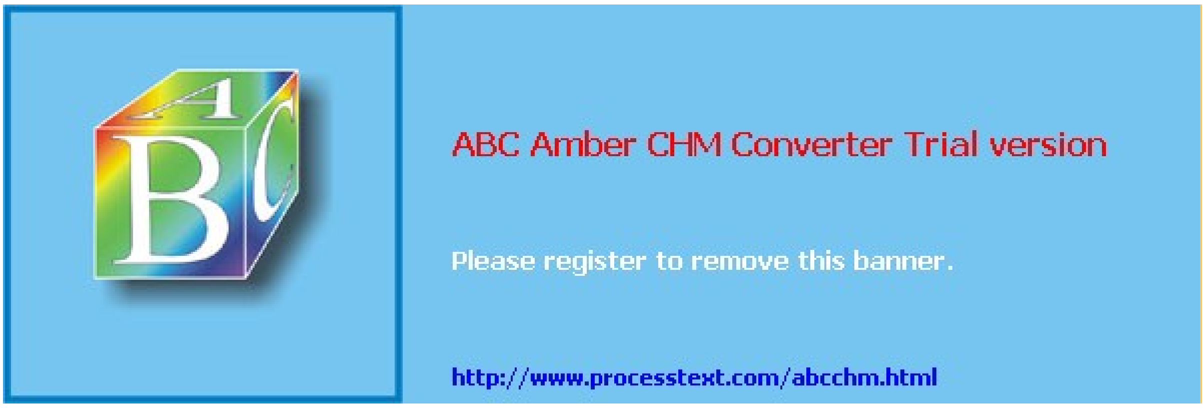
- Application gateways or proxies (e.g., mail relays and HTTP proxies) act as mediators between the communicating application processes and can implement fine-grained, application-specific access control.

A complete firewall configuration (see Figure 19-1) normally consists of an inner router with packet-filtering functionality, which forms the transition to the network to be protected; an outer router with packet-filtering functionality, which forms the transition to the external network; and a number of application gateways located in an independent local area network between these routers. This network within the firewall is normally called a demilitarized zone (DMZ) or screened subnet. If gateways are available for all required application protocols, then the packet filters can be configured so that no packets are forwarded directly between the internal and the external networks. Instead, exclusive communication is between the internal network and the DMZ and between the external network and the DMZ.

Figure 19-1. Structure of a firewall.

[View full size image]

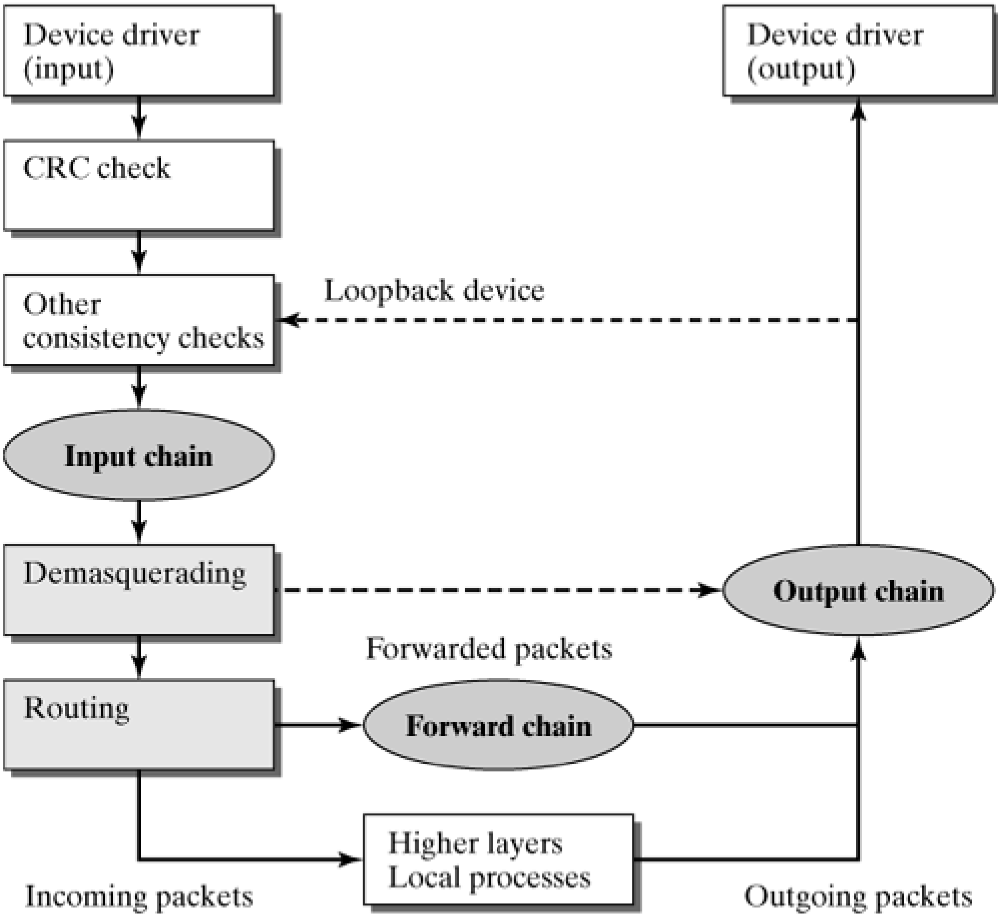




19.2 The Ipchains Architecture of Linux 2.2

`ipchains` is a packet-filtering architecture consisting of an infrastructure in the Linux kernel and a user-space program to manage rules lists, like all packet-filtering architectures currently implemented in Linux. In Linux 2.2, this product is called `ipchains`. (See Figure 19-2.) Section 19.2.1 will discuss its invocation syntax and how we can define rules.

Figure 19-2. The packet-filtering architecture in Linux 2.2 (`ipchains`).



The filtering mechanisms implemented in Linux kernel Version 2.2 divide all data packets into the following three classes, depending on their source and destination addresses:

1. incoming packets? those addressed to the local computer;
2. packets to be forwarded and leaving the local computer over a different network interface based on a routing decision;
3. outgoing packets created in the local computer.

For each of these classes, the network stack of the respective protocol includes a prominent position, and each packet of the corresponding class has to pass it. In each of these positions, there is a hook, where a linked rules list (chain) is hooked, hence the name `ipchains`.

According to the packet class they are allocated to, the rules lists are called input chain, forward chain, and output chain. These chains are organized so that they are processed sequentially, beginning from the first defined rule. If a rule accepts an incoming packet, then this packet is handled according to the branch destination defined in the rule, where Linux Version 2.2 introduced the support of user-defined rules lists. This means that, in addition to the linear processing of rules lists, we can also implement branching. Other possible branch destinations are the following:

- ACCEPT? completes processing of the rules list and releases the packet for further handling.

19.3 The Netfilter Architecture of Linux 2.4

Linux Version 2.4 divides the packet-filtering functionality into two large blocks: The so-called netfilter hooks offer a comfortable way to catch and manipulate processed IP packets at different positions on their way through the Linux kernel. Building on this background, the `iptables` module implements three rules lists to filter incoming, forwarded, and outgoing IP packets. These lists correspond roughly to the rules lists used by `ipchains`. In addition, similar modules are available for other network prot ocols (e.g., `ip6tables` for IP Version 6).

19.3.1 Netfilter Hooks in the Linux Kernel

As was mentioned briefly in Section 19.2.2, the netfilter architecture includes a uniform interface, reducing the cost involved to implement new functions. It is called netfilter hook, which means that it provides a hook for packet-filter code. This section discusses the components of this architecture and its implementation in the Linux kernel. Actually, this section supplies brief instructions to facilitate your writing your own netfilter modules.

Netfilter modules can be loaded into the Linux kernel at runtime, so we need hooks in the actual routing code to enable dynamic hooking of functions. An integer identifier is allocated to each of these netfilter hooks. The identifiers of all hooks for each supported protocol are defined in the protocol-specific header file (`<linux/netfilter_ipv4.h>` or `<linux/netfilter_ipv6.h>`). The following five hooks are defined for IP Version 4 in `<linux/netfilter_ipv4.h>`:

- `NF_IP_PRE_ROUTING (0)`: Incoming packets pass this hook in the `ip_rcv()` function (see Section 14.2.1) before they are processed by the routing code. Prior to that, only a few simple consistency checks with regard to the version, length, and checksum fields in the IP header are done.

  Meaningful opportunities to use this hook result whenever incoming packets should be caught before they are processed? for example, to detect certain types of denial-of-service attacks that operate on poorly built IP packets, or for address-translation mechanisms (NAT), or for accounting functions (counting of incoming packets).

- `NF_IP_LOCAL_IN (1)`: All incoming packets addressed to the local computer pass this hook in the function `ip_local_deliver()`. At this point, the `iptables` module hooks the `INPUT` rules list into place to filter incoming data packets. This corresponds to the input rules list in `ipchains`.

- `NF_IP_FORWARD (2)`: All incoming packets not addressed to the local computer pass this hook in the function `ip_forward()`? that is, packets to be forwarded and leaving the computer over a different network interface.

  This includes any packet the address of which was modified by `NAT`. At this point, the `iptables` module hooks the `FORWARD` rules list into place to filter forwarded data packets. This corresponds to the forward rules list in `ipchains`.

- `NF_IP_LOCAL_OUT (3)`: All outgoing packets created in the local computer pass this hook in the function `ip_build_and_send_pkt()`. At this point, the `iptables` module hooks the `OUTPUT` rules list into place to filter outgoing data packets. This corresponds to the output rules list in `ipchains`.

- `NF_IP_POST_ROUTING (4)`: This hook in the `ip_finish_output()` function represents the last chance to access all outgoing (forwarded or locally created) packets before they leave the computer over a network device. Like the `NF_IP_PRE_ROUTING` hook, this is a good place to integrate accounting functions.

Figure 19-3 shows data packets traveling through different hooks.

Figure 19-3. The packet filtering architecture of Linux 2.4 (netfilter).

[View full size image]

◂ Previous Next ▸

Chapter 20. Connection Tracking

◂ Previous Next ▸

◂ Previous Next ▸

20.1 Introduction

This chapter discusses the connection-tracking module, which forms the basis for extended packet-filter functions, particularly for network address translation (NAT? see Chapter 21) in Linux 2 .4.

The connection-tracking module manages individual connections (particularly TCP connections, but also UDP associations) and serves to allocate incoming, outgoing, and forwarded IP packets to existing connections. A new connection entry is generated as soon as the connection-tracking module registers a connection-establishment packet. From then on, each packet belonging to this connection is uniquely assigned to this connection. For example, this enables the NAT implementation to figure out exactly whether an incoming packet needs a free IP address and port number or one of the addresses and port numbers previously assigned can be used. The connection is deleted after a certain period of time has elapsed without traffic (timeout), which depends on the transport protocol used (i.e., TCP, UDP, or ICMP). Subsequently, the NAT module can reuse the address and port number that have become available.

The connection-tracking module is not limited to transport protocols; it can basically also support complex application protocols. For example, a stateful filter and an address-translation mechanism for active FTP (see Section 19.1.2) can be implemented. For this purpose, the connection-tracking module has to be able to associate newly established data connections with an existing control connection.

20.1.1 Using the Connection-Tracking Module

Two functions can be invoked to access connection entries: `ip_conntrack_get()` and `ip_conntrack_put()`. The `ip_conntrack_get()` function returns a connection entry for an IP packet passed as an `sk_buff` structure and automatically increments the reference counter for this connection. The `ip_conntrack_put()` function informs the connection-tracking module that the p reviously requested connection is no longer needed and decrements the reference counter.

To find a connection entry, we can use a so-called tuple (see Section 20.2.2) instead of an `sk_buff` structure with a complete IP packet. Such a tuple contains only the source and destination addresses and additional protocol information. The `ip_conntrack_find_get()` is used for this purpose.

◂ Previous Next ▸

◂ Previous Next ▸

20.2 Implementation

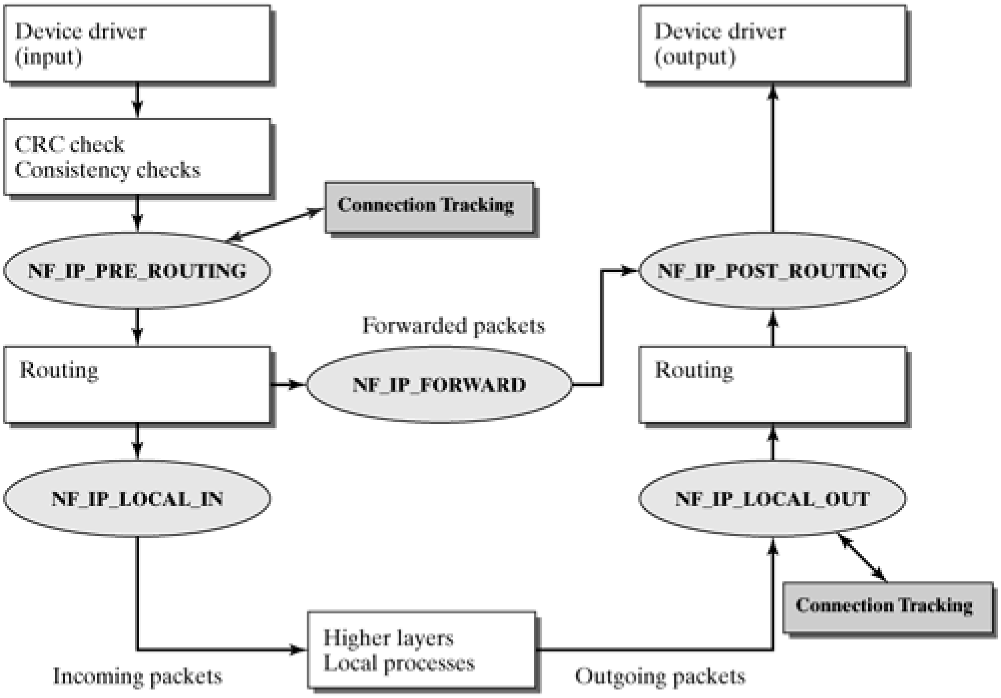
The module interface of the connection-tracking module is located in the file `net/ipv4/netfilter/ip_conntrack_standalone.c`. The file `net/ipv4/netfilter/ip_conntrack_core.c` contains the actual connection-tracking functionality.

20.2.1 Basic Structure

The connection-tracking module hooks itself into the netfilter hooks `NF_IP_PRE_ROUTING` and `NF_IP_LOCAL_OUT` (see Section 19.3.1 and Figure 20-1) with very high priority (the `NF_IP_PRI_CONNTRACK` is set to ?00 in `<linux/netfilter_ipv4.h>`). This means that each incoming packet is first passed to the connection-tracking module. Subsequently, other modules also hooked into these hooks, but with lower priority, get their turns.

Figure 20-1. Netfilter hooks used by the connection-tracking module.

[View full size image]



20.2.2 Connection Entries

`struct ip_conntrack` linux/netfilter_ipv4/ip_conntrack.h

A connection entry is represented in the Linux kernel by an `ip_conntrack` structure, consisting of the following fields:

- A structure of the type `nf_conntrack` (defined in `<linux/skbuff.h>`), which includes a reference counter (`use`) that counts the number of open references to this connection entry.
- Two tuples for forward and reverse direction (`tuplehash[0], tuplehash[1]`), consisting of address and protocol information, which can be used to reference this entry.
- A status field (`status`), containing a bit vector with the following bits:
  - `IPS_EXPECTED`: The connection was expected.
  - `IPS_SEEN_REPLY`: Packets have already occurred in both directions.

◂ Previous Next ▸

Chapter 21. Network Address Translation (NAT)

◂ Previous Next ▸

◀ Previous Next ▶

21.1 Introduction

The Network Address Translation (NAT) mechanism deals generally with the translation of IP addresses. It represents an effective way to encounter the exhaustion of free IP addresses (Version 4) in view of the explosive growth of the Internet. However, transition to IP Version 6 with its much larger address space progresses only slowly, because it was found extremely difficult to convert a decentralized network like the global Internet to a new protocol at one shot. Exactly this is where NAT comes in useful: For example, it allows all users in a local area network to access the Internet and its services, even though there is only one single official IP address available, and only private IP addresses (according to RFC 1918 [RMKG + 96]) are used within the local area network. A router accessible to the network, used by the users to connect to the global Internet, handles the required address mapping.

The NAT implementation in Linux 2.4 consists of two parts: connection tracking, and the actual NAT. Chapter 20 described how the connection-tracking mechanism is implemented.

21.1.1 Important Terminology

One of the most important technical terms in the NAT area is the so-called session flow. A session flow is a set of IP packets, exchanged between two instances and forming a unit in that they are treated equally by a NAT router. Such a session flow is directed to the direction the first packet was sent. For this reason, we speak of original and reverse directions in the following discussion. One good example is a telnet session: The corresponding TCP connection is initiated by the terminal computer, so the original direction of the relevant session flow points from the terminal to the server. T CP/UDP-based session flows can be described uniquely by an {IP source address, source port, IP destination address, destination port} tuple. Similarly, an ICMP session flow can be identified by an {IP source address, IP destination address, ICMP type, ICMP ID} tuple.

[SrHo99] describes three characteristic requirements that should be met by all NAT variants:

- transparent address allocation;
- transparent routing; and
- correct handling of ICMP packets.

The following sections explain each of these requirements and what they mean.

21.1.2 Transparent Address Allocation

Because we can use NAT to connect networks with different address spaces, these addresses from the respective address spaces have to be allocated among them. This allocation can be either static or dy namic. If we use static allocation, the allocations are maintained during the entire operation of a NAT router. Static allocations simplify the address translation, because no state information about specific session flows has to be maintained. In dynamic allocation, the allocation is specified at the time that a session flow is opened. This allocation remains valid until the session is terminated. In some cases, the allocation rule can be extended beyond the IP addresses to include the transport protocol ports. (See Section 21.1.6.) In any event, address allocation should be transparent: The mechanisms should be hidden from the applications in end systems.

21.1.3 Transparent Routing

We use the term "transparent routing" in the following discussion to distinguish the routing functionality of a NAT router from the functionality of a normal router. Transparent routing differs from normal routing in that packets are forwarded between two different address spaces by changing the address information in IP packets and routing to match these modified addresses.

Transparent routing can be divided into three phases: address binding, address translation, and releasing of the address binding.

- Address binding: This phase permanently binds two addresses from the two address spaces to be bound. This should not be confused with the address allocation mentioned above? the address allocation only determines valid bindings. These bindings will then actually exist only

Previous Next

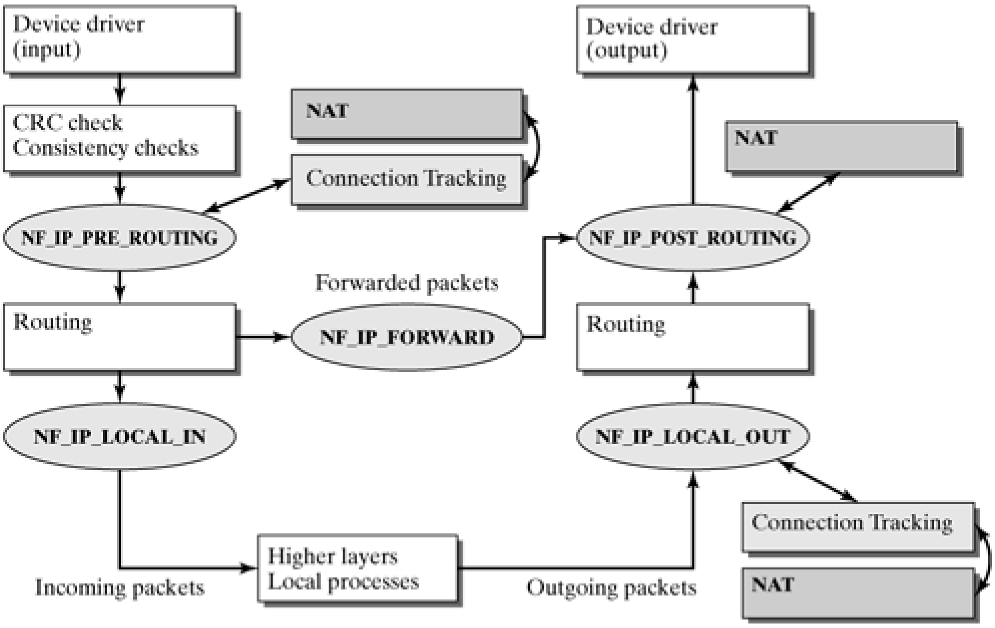
21.2 Configuring NAT in Linux

The iptable_nat.o module implements the unidirectional NAPT variant described in Section 21.1.6. Like the Twice NAT variant, it can change the source and destination addresses of a session flow simultaneously.

To intercept and process packets, NAT uses the infrastructure supplied by the netfilter architecture. (See Section 19.3.1.) Figure 21-2 shows that it hooks itself into the netfilter hooks NF_IP_PRE_ROUTING, NF_IP_POST_ROUTING, and NF_IP_LOCAL_OUT for this purpose. The NAT module is invoked as soon as a packet traverses the appropriate hook, and a pointer to the sk_buff structure is passed, together with the packet. For configuration purposes, it is important that the source address be translated at the NF_IP_POST_ROUTING hook while the destination address is being translated in one of the other two hooks.

Figure 21-2. Netfilter hooks used by the NAT module.

[View full size image]

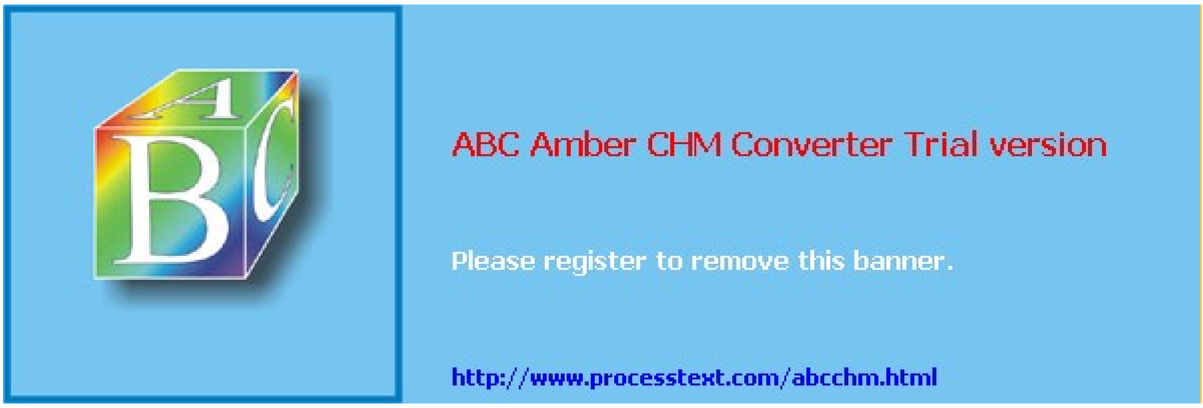


The first, preliminary versions of the new netfilter architecture allowed you to configure NAT by using an independent tool called ipnatctl. More recently, this functionality was fully integrated in the iptables tool. iptables can be used to specify rules that control the behavior of the NAT module. As was described in Section 19.2.1, a rule consists of a set of criteria to select session flows (matching rule) and a second part specifying how a session flow should be transformed (binding type or mapping type).

Criteria identical to packet-filter rules are available to select session flows: the IP source and destination addresses, the transport protocol, the port numbers, and the protocol-specific flags. The second part of a NAT rule defines how a session flow should be transformed. To this end, there are additional branch destinations, which are valid in the nat table only. We can use -j SNAT to activate the translation of the source address (source NAT) and -j DNAT to translate the destination address (destination NAT). In addition, we have to use --to-source or --to-destination to specify a range of IP addresses and port numbers, if present, for the address-translation process.

The selection criteria of the source NAT rule are applied to the original packet-address information in the event that both a source NAT and a destination NAT rule apply to the packet, though the destination address has already been changed by the destination NAT at that point. There are additional branch destinations (e.g., -j MAS-QUERADE for masquerading in the Linux 2.2 style) for special cases.

In the example discussed in Section 21.1.6, where source NAT is used to map the internal addresses from the private address range 192.168.1.0?92.168.1.255 to the global address 199.10.42.1, the corresponding iptables invocation would have the following form:



◀ Previous Next ▶

21.3 Implementing the NAT Module

This section first introduces important data structures to manage session flows, allocations, and address bindings. Subsequently, it will explain the functions used to establish and tear down address bindings, to actually translate addresses, and to handle ICMP error messages.

21.3.1 Important Data Structures

All session flows are completely managed by the connection-tracking module described in Chapter 20. A structure of the type `ip_conntrack` is stored for each session flow. (See Section 20.2.2.) This structure includes two data structures of the type `ip_conntrack_tuple_hash`, representing the forward and reverse directions of a session flow. If a session flow is translated by the NAT module, then the `ip_conntrack_tuple_hash` structure for the reverse direction is adapted so that reply packets can be allocated to it properly.

In the example discussed in Section 21.1.6, where the internal address 192.168.1.1 is translated into the global address 199.10.42.1, a connection from port 1200 to port 80 in the WWW server 100.1.1.1 would be represented by the following entries:

- Forward: 192.168.1.1:1200 → 100.1.1.1:80
- Reverse: 100.1.1.1:80 → 199.10.42.1:1200

The connection-tracking module stores a pointer to the relevant data structure of the type `ip_conntrack` in the `sk_buff` of each packet. If the NAT module wants to allocate an IP packet to a session flow, it invokes its own `ip_conntrack_get()` function, which returns the matching `ip_conntrack` structure.

`struct ip_nat_expect` linux/netfilter_ipv4/ip_nat_rule.h

The NAT module has an ordered list, `nat_expect_list`, with data structures of the type `ip_nat_expect`, to enable protocol-specific NAT modules (e.g., for FTP? see Section 21.1.7) to decide when a new session flow was expected, so that it requires special handling. Each of these structures consists essentially of a pointer to a function that actually makes that decision:

```
struct ip_nat_expect
{
struct list_head list;
       /* Returns 1 (and sets verdict) if it has setup NAT for this
          connection */
       int (*expect) (struct sk_buff **pskb,
                      unsigned int hooknum,
                      struct ip_conntrack *ct,
                      struct ip_nat_info *info,
                      struct ip_conntrack *master,
                      struct ip_nat_info *masterinfo,
                      unsigned int *verdict);
};
```

`struct ip_nat_multi_range` linux/netfilter_ipv4/ip_nat.h

The `ip_nat_multi_range` structure is used mainly to specify the set of addresses available for address translation. It contains one or several structures of the type `ip_nat_range`, each specifying a continuous IP address range, and a `rangesize` field that takes the number of contained `ip_nat_range` structures:

```
struct ip_nat_multi_range
{
```

◂ Previous Next ▸

21.4 Interfaces to Extend the NAT Module

The NAT module offers various extension options. These extensions are actually independent modules that can use the registration functions of the NAT module to register and unregister themselves. The following extensions are possible:

- Transport protocols (e.g., TCP): To use a new protocol, we have to write two extension modules? one for connection tracking and one for NAT.
- Helper modules (helpers): To be able to handle application protocols, such as FTP (see Section 21.1.7), properly, we can register helper modules. Again, this requires one helper each for connection tracking and NAT.
- Configuration-tool extensions: In addition, the `iptables` configuration tool has to be extended by the corresponding command-line parameters for each new protocol and each new helper module. We will not discuss this issue any further.

21.4.1 Transport Protocols

The functions `ip_nat_protocol_register()` and `ip_nat_protocol_unregister()` can be used to register a new transport protocol or to unregister an existing protocol. When registering a new protocol, we have to pass a pointer to a structure of the type `ip_nat_protocol` as a parameter.

`struct_ip_nat_protocol` linux/netfilter_ipv4/ip_nat_protocol.h

```
struct_ip_nat_protocol
{
       struct list_head list;

       /* Protocol name */
       const char *name;

       /* Protocol number. */
       unsigned int protonum;

       /* Do a packet translation according to the ip_nat_proto_manip
        * and manip type. */
       void (*manip_pkt) (struct iphdr *iph, size_t len,
              const struct ip_conntrack_manip *manip,
              enum ip_nat_manip_type maniptype);
       /* Is the manipable part of the tuple between min and max incl? */
       int (*in_range)(const struct ip_conntrack_tuple *tuple,
              enum ip_nat_manip_type maniptype,
              const union ip_conntrack_manip_proto *min,
              const union ip_conntrack_manip_proto *max);


       /* Alter the per-proto part of the tuple (depending on
       maniptype), to give a unique tuple in the given range if
       possible; return false if not. Per-protocol part of tuple
       is initialized to the incoming packet. */
       int (*unique_tuple)(struct ip_conntrack_tuple *tuple,
             const struct ip_nat_range *range,
             enum ip_nat_manip_type maniptype,
             const struct ip_conntrack *conntrack);
       unsigned int (*print)(char *buffer,
             const struct ip_conntrack_tuple *match,
             const struct ip_conntrack_tuple *mask);

       unsigned int (*print_range)(char *buffer,
```

◂ Previous Next ▸

Chapter 22. Extending the Linux Network Architecture Functionality? KIDS

component instances: This chapter deals with the introduction of new, dynamically extendable functionalities in the Linux network architecture or in the Linux kernel. We will first show the usual approach to manage dynamically extendable functionalities and the operations generally involved in this approach.

Subsequently, we will use Linux KIDS, the implementation of a construction system to support network services, to show how a dynamically extendable functionality can be managed. Another interesting aspect of Linux KIDS is how its object-oriented concept is implemented in the Linux kernel, considering that the Linux kernel was not designed with object orientation in mind.

In addition, we will explain how the KIDS components can be embedded into the processes of protocol instances over existing interfaces, which means that you can use the functionality of Linux KIDS without having to change the kernel source code. Finally, this chapter describes how you can use a character-oriented device to configure the KIDS construction system, allowing you to easily configure this functionality in the kernel.

◂ Previous Next ▸

22.1 Managing Dynamically Extendable Functionalities

The Linux network architecture is continually extended by new functions and protocol instances. In most cases, such an extension can even be dynamic (i.e., at runtime), as we saw in earlier chapters: network-layer protocols (Section 6.3), transport-layer protocols (Section 14.2.5), and packet filters (Section 19.3).

All we need to implement such a dynamic extendibility are an appropriate interface and structures that manage the registered functionality. Functions like `register_functionality()` and `unregister_functionality()` can be used to register new functionalities with the kernel or remove existing functionalities.

These registering and unregistering functions execute all initialization or cleanup steps required. For example, when removing a functionality, we have to ensure that it is no longer used in any other location of the kernel. This means that it has to use a reference counter (use counter) and check this counter (`use_counter == 0?`) before unregistering a functionality.

The following operations are some of those normally executed in the registration function of an interface and undone, accordingly, in the unregistering functions:

- storing the new functionality of its management structure in a list, hash table, or another data structure;
- reserving memory for the required data structures or procuring other resources (IRQ, DMA, timer, etc.);
- incrementing reference counters;
- creating entries in the proc directory;
- using `printk()` to output status messages.

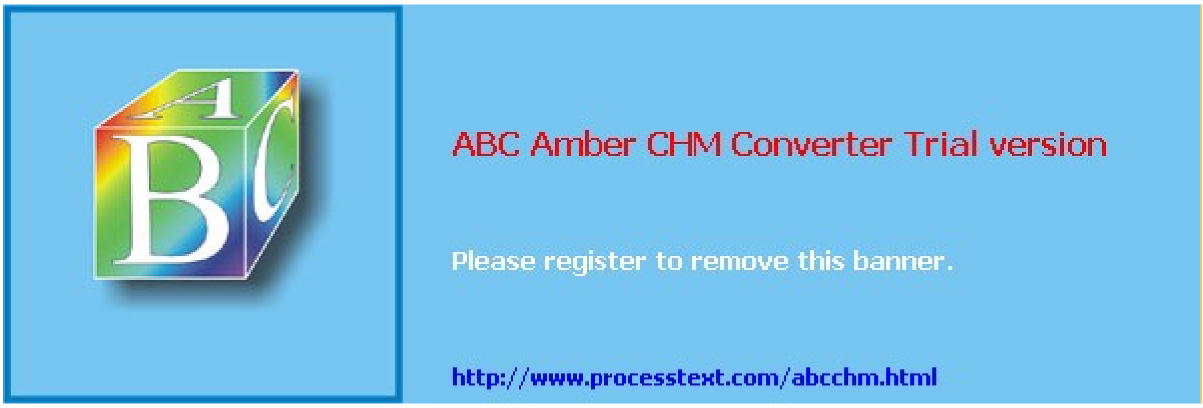
A new functionality (e.g., a new network-layer protocol or a new network device) normally takes many parameters. It would be difficult the pass all of these parameters individually in the registration function, mainly because they are required while the functionality is being used, which is normally outside the registration function. For this reason, a structure to manage the functionality is normally created. When registering a new functionality, this structure can be entered in a list, hash table, or similar management structures. This method ensures that we can access these functionalities and their parameters after the registration. Earlier chapters introduced a large number of such management structures, including the `net_device` structure for network devices and the `packet_type` structure for network-layer protocols.

Such a management structure is filled with the parameters required before it is registered. Subsequently, a pointer to this management structure is passed to the registration function. The elements of these management structures assume two different tasks:

- Configuration data is set before a functionality is registered and passed as (configuration) parameter within the structure. In the `packet_type` structure (Chapter 6), for example, these parameters include the `type` and `func` variables.
- Runtime variables are not explicitly set before a functionality is registered. They are needed later when the functionality is used (e.g., `next` to link structures, or a use counter to count references).

After a registration, of course, we can also use configuration variables as runtime variables, if the initial value is no longer needed.

After this brief overview of the principles of how to manage functionalities registered dynamically, the sections following discuss how the KIDS framework is implemented in the Linux kernel 2.4 (Linux KIDS). This framework relies heavily on the concept of dynamically extendable functionalities, as shown in our Linux KIDS example further below. Another interesting point is that the components and their instances are based on the object-oriented concept, and this approach could probably be adopted for other software projects in the Linux kernel. The next section begins with an introduction of its



◀ Previous Next ▶

22.2 Structure of the KIDS Construction System

KIDS stands for Karlsruhe Implementation architecture of Differentiated Services and was developed for the design, evaluation, and use of quality-of-service (QoS) mechanisms in networks [Wehr01b]. KIDS is an abstract model describing the structure and interaction of quality-of-service mechanisms, and it allows you to define individual QoS behavior in a flexible way. The KIDS framework was implemented on various platforms, including Linux (kernel Version 2.4) and the OMNeT++ simulation tool. Though Linux already supports some QoS features (see Chapter 18), KIDS introduces several important benefits, as we will see in the further course of this chapter. The next two sections will introduce the general structure of the KIDS framework; however, we will leave out a few details to keep things short? see [Wehr01a] for details.

22.2.1 Elementary QoS Components

KIDS was designed to create a flexible, extendable, and modular framework for implementing individual QoS mechanisms. It is based on the use of components that implement the elementary QoS mechanisms, and so it is easy to combine them to more complex QoS mechanisms. Simple combination of components and ensuring all potential degrees of freedom and easy extendability were the most important factors in the design of the KIDS framework.

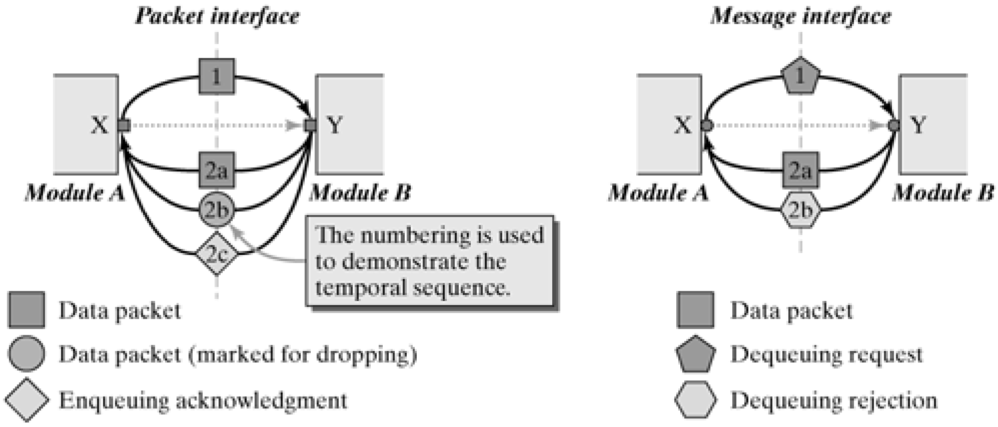
KIDS can be thought of as a construction kit, similarly to the popular Lego system, consisting of components that have different interfaces. Components with similar interfaces can be connected (almost arbitrarily) to form complex constructions (in this case QoS mechanisms).

We can distinguish two different interfaces in the components of the KIDS framework (as seen in Figure 22-1):

- At a packet interface (), a component, A, uses packet output X to pass a data packet it received at its input to the successor component, B. By convention, a KIDS component has but a single input.
- At a message interface (), component A uses message output X to pass a message to component B, requesting the latter for a packet.

Figure 22-1. Interactions at the interfaces between two KIDS components.

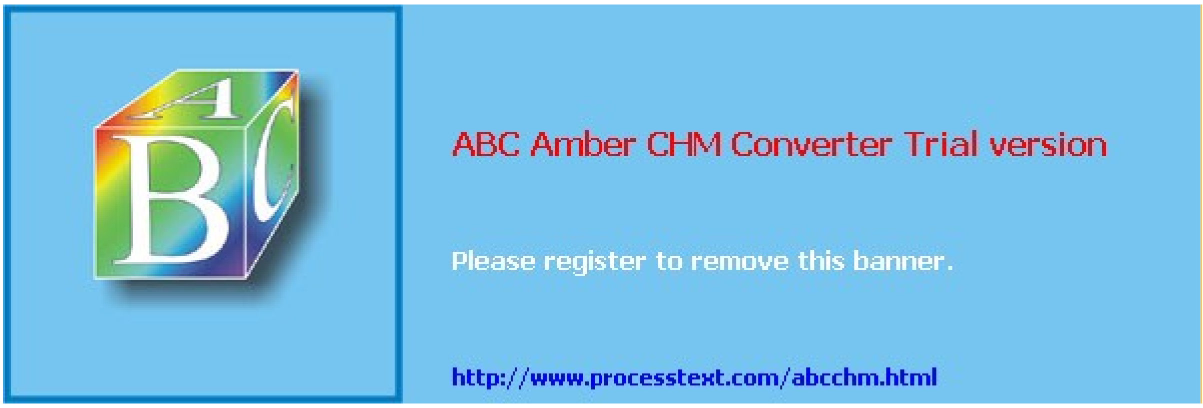
[View full size image]



By using these two interface types, we can distinguish five component classes, as shown in Figure 22-2. Each QoS mechanism can be assigned to one of these classes:

- Operative components (`BHVR`) operate on packets: They receive a packet and operate their algorithm on this packet. The algorithm implemented in a component either changes the packet (active operative component) or studies its output to forward the packet (passive operative component).

  Examples: Token Bucket, Shaper, Marker, Dropper, Classifier, Random Early Detection (RED).
- Queue components (`QUEUE`) are data structures used by components to enqueue or dequeue



22.3 Using the KIDS Example to Extend the Linux Network Architecture

Now that we have given a brief overview of the elements in the KIDS framework, this section will discuss its implementation in the Linux kernel as an example of how the functionality of the Linux network architecture can be extended. We focus our discussion on the design and management of the components: how and why they were designed, and how they are introduced to the kernel at runtime. In addition, we will see how hooks are implemented on the basis of different existing kernel interfaces, which means that we don't have to change the kernel to be able to use KIDS. Finally, we use the `kidsd` daemon as an example to show how components and hooks are configured and how they interact between the kernel and the user level.

22.3.1 Components and Their Instances

The KIDS framework offers different types of components that can be used to implement different QoS mechanisms (e.g., token buckets? see Section 18.6.1). A component can occur more than once within a component chain, and each of these occurrences can have different parameters. This means that we should be able to create an arbitrary number of instances from a component, but still try to keep the memory required by these instances low. This principle reminds us strongly of the object-orientation concept that lets you create an arbitrary number of object instances from a class. Although all of these classes exist independently, they have the same behavior, because they use the same methods.

This means that the component concept of Linux KIDS has an object-oriented character, though it was written in C, a programming language that doesn't support object orientation. The component concept of Linux KIDS consists of the following two parts:

- Components are QoS mechanisms implementing a specific behavior. They are managed in the `bhvr_type` structure of Linux KIDS. This structure contains all properties of a component (e.g., its behavior in the form of pointers to corresponding methods? shown below). These methods are used by several instances of that component concurrently, so they have to be reentrant. Components correspond to the principle of classes in the object-oriented model.

- Component instances are created when we need an instance of a component. To this end, we create a data structure of the type `bhvr`. It stores all information about this component instance? mainly, its individual parameter configuration. The instance should have the component's behavior, so reference is made to the information stored in the `bhvr_type` structure of the component. Component instances correspond to objects (or object instances) in the object-oriented model.

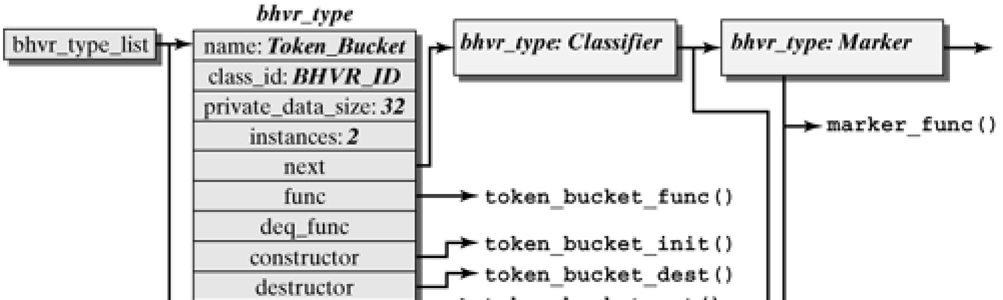
The following discussion introduces how these two structures are built and what the parameters mean. Subsequently, we will see how components can be registered or unregistered dynamically.

`struct bhvr_type` kids/kids_bhvr.h

Figure 22-4 shows how components and their instances interact. The `bhvr_type` structure of the token bucket stores general component information.

Figure 22-4. The `bhvr_type` and `bhvr` structures manage components and their instances.

[View full size image]



◀ Previous Next ▶

Chapter 23. IPv6? Internet Protocol Version 6

◀ Previous Next ▶

◂ Previous Next ▸

23.1 Introduction

Several years have passed since RFC 791 for IPv4 was published. During this time, the requirements on the IP version used in the Internet have changed considerably. For example, the address space for 32-bit IPv4 addresses is almost depleted, in particularly because each mobile device, or even each household device, is expected to get its own IP address. In addition, the transmission technologies in fixed networks have so matured that packet errors have virtually been eradicated. These facts were motivation for further development beyond IPv4 to the protocol for the future Internet, resulting in IPv6. Since 1998, several standards have been introduced for IPv6, and the bases for these standards are the following RFCs:

- RFC 2460 [HiDe98a] specifies IPv6.
- RFC 2373 [HiDe98b] describes the architecture for IPv6 addressing.

◂ Previous Next ▸

◂ Previous Next ▸

23.2 IPv6 Features

The new IP version was improved in many important points, but this protocol has been used in the real world only to a limited extent. One of the reasons is that existing applications cannot run directly on top of IPv6. The most important changes from IPv4 are the following:

- Extended address size: Instead of 32 bits, each IPv6 address contains 128 bits, enabling several hierarchical levels and addressing a much larger number of nodes. In addition, the IPv6 address of each node can be configured automatically. The support of multicast routing has been improved. Moreover, a new address type, the anycast address, was defined, which allows you to send a packet to an arbitrary node from within a group.
- Simplified header format: Some of the IPv4 packet-header fields are no longer supported, or are now optional, to reduce the cost involved in processing IPv6 packets.
- Extension headers: The way IPv6 encodes information is completely different from that in IPv4, enabling more efficient forwarding, less strict limitations with regard to the length of options, and more flexibility for new, future packet options.
- Flow labeling: IPv6 is able to mark packets that belong to a specific stream. This allows the sender to request special handling of these packets, enabling a much better support of service qualities, such as priority handling or real-time services.
- Authentication and data protection: IPv6 specifies extensions to support authentication, integrity, and confidentiality of data.

23.2.1 Addressing

The address space, extended from 2^{32} (IPv4) to 2^{128} (IPv6) addresses, requires a new address notation. The preferred and abbreviated notation is the hexadecimal notation (e.g., `FEDC:BA98:7654:3210:FEDC:BA98:7654:3120`). Each group (i.e., one block between two colons, or between the beginning/end and a colon) represents 16 bits. Leading zeros can be omitted, so one group can consist of one to four hexadecimal numbers. In addition, it is assumed that many consecutive blocks consist of zeros, so a compressed notation was introduced: Each address may contain at most one occurrence of two consecutive colons. In between, as many zeros as necessary are used to reach the full length of an address. The following examples show this notation (the meaning of each of these addresses will be discussed further below):

- A "loopback" address:
-
- `::1 or 0:0:0:0:0:0:0:1 = 0000:0000:0000:0000:0000:0000:0000:0001`

- A normal address:
-
- `F83:5::12 or F83:5:0:0:0:0:0:12`
- `= 0F83:0005:0000:0000:0000:0000:0000:0012`

Alternatively, addresses can be represented in mixed form, composed of the new hexadecimal notation and the decimal IPv4 notation. The format is then `x:x:x:x:x:x:d.d.d.d`, where x represents the hexadecimal groups of IPv6 and `d` stands for the decimal IPv4 convention. One example would be `0:0:0:0:0:FFFF:129.13.64.5` (or `::FFFF:129.13.64.5` in the abbreviated form). How useful this is becomes obvious if you think of embedding IPv4 in IPv6.

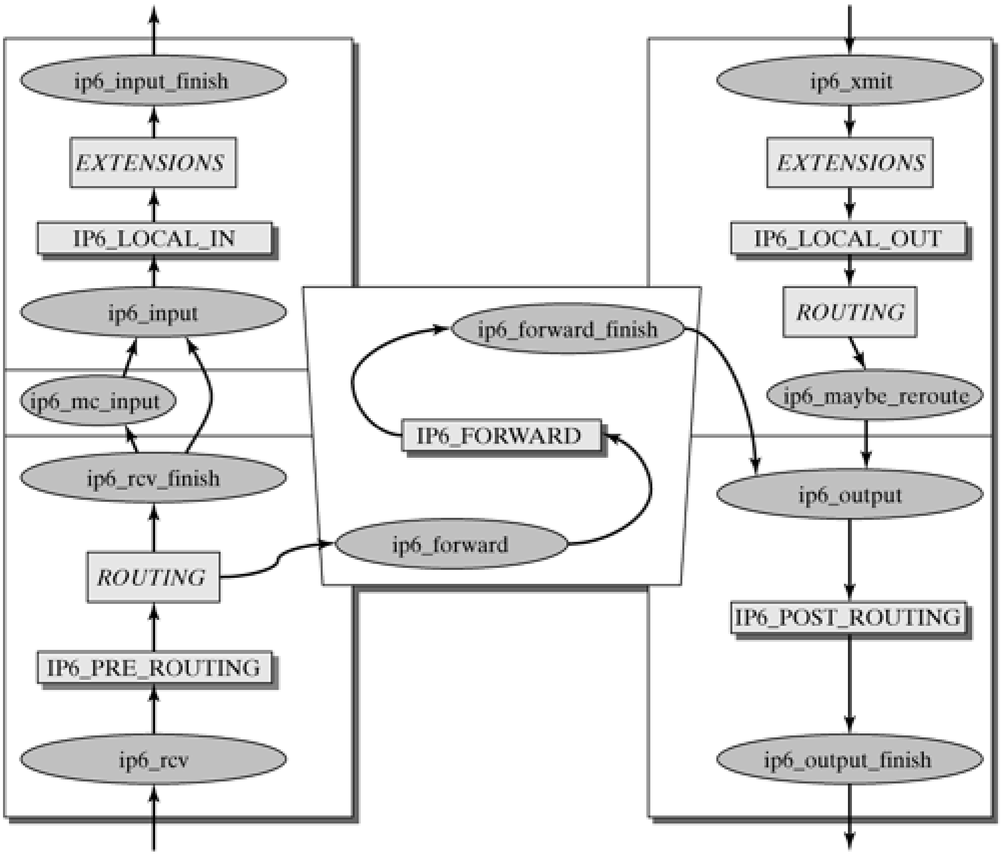
As does IPv4, IPv6 supports unicast and multicast addresses. A new form of communication introduced in IPv6 is anycast. Anycast is a mixture of unicast and multicast: A packet is sent to one computer in a multicast group, where the network itself decides which computer this is. The "broadcast" address of IPv4 `(255.255.255.255)` doesn't exist in IPv6. This functionality, which was used mainly by ARP (Address Resolution Protocol) to resolve IP addresses, is achieved by use of multicast addresses in IPv6. ARP is no longer supported in IPv6. It was replaced by Neighbor Discovery, which was integrated into ICMPv6.

23.3 IPv6 Implementation

The implementation of IPv6 in the Linux kernel is in the `net/ipv6` directory and in the header file `<include/net/ipv6.h>`. The code of IPv4 formed the basis for the IPv6 implementation, so that most things are similar. As in IPv4, packets can reach the IPv6 layer in either of three possible ways. Figure 23-3 shows how a packet travels across the Linux kernel. Packets received by the network card are passed by the function `ipv6_rcv(sbk, dev, pt)` to the data-link layer, and `ip6_xmit(skb)` sends packets created by higher layers or protocols (e.g., UDP or TCP). Finally, special commands, such as `icmpv6_send()`, can be used to create IPv6 packets in the `IP` layer.

Figure 23-3. IPv6 implementation in the Linux kernel.

[View full size image]



23.3.1 Incoming Packets

`ipv6_rcv()` include/net/ipv6.h

`ip6_rcv_finish()` net/ipv6/ip6_input.c

The `ipv6_rcv()` function accepts IPv6 packets incoming from the lower layer. If an incoming packet is addressed to a different computer, it is dropped immediately by `ipv6_rcv()`. If an IPv6 packet is addressed to the local computer, then the first things to do are to check the IPv6 packet header and the packet length and to use the `skb_trim()` function to correct things, if necessary. If a packet-header extension of the type Hop-by-Hop Options follows next, this extension is processed by the function `ipv6_parse_hopopts()`. Subsequently, the NETFILTER call `NF_IP6_PRE_ROUTING` passes the IPv6 packet to the `ip6_rcv_finish()` function, which invokes one of these three functions: `ip6_input(), ip6_mc_input(), ip6_forward()`.

23.3.2 Forwarding Packets

◂ Previous Next ▸

Part V: Layer IV? Transport Layer

◂ Previous Next ▸

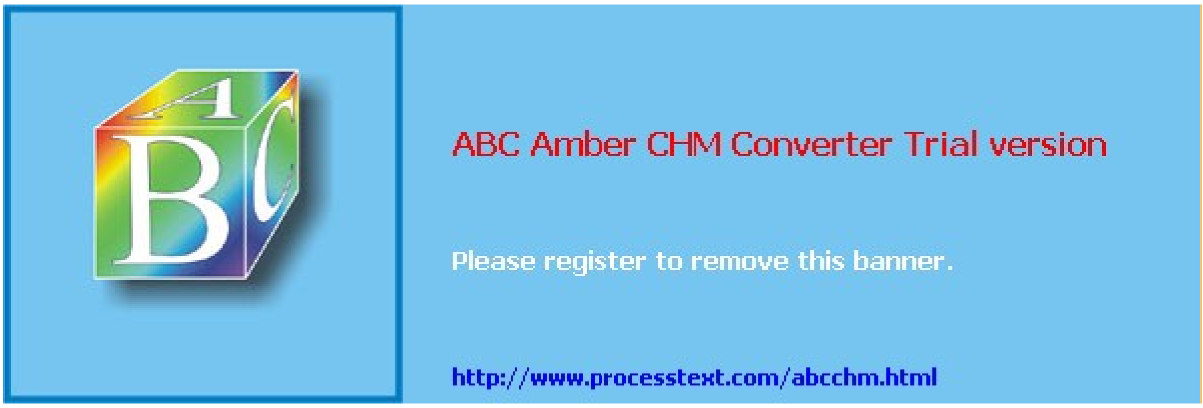
◂ Previous Next ▸

Chapter 24. Transmission Control Protocol (TCP)

The Transmission Control Protocol (TCP) offers a reliable, byte-oriented, connection-oriented transport service, in contrast to the unreliable datagram service used by the Internet Protocol (IP). Providing these abilities makes the TCP transport protocol very complex. A large number of protocol mechanisms are required to achieve the expected service. This chapter introduces these protocol mechanisms and describes how they were implemented in the Linux kernel.

The TCP protocol belongs to the transport layer and can be used as an alternative to the User Datagram Protocol (UDP), which offers a connectionless transport service. (See Chapter 25.) The transport layer is immediately below the application layer. Consumers using the service of the protocol are applications, and they reach the services of the TCP protocol instance over the socket interface introduced in Chapters 26 and 27. To implement the transport service, the TCP layer uses the Internet Protocol (IP). It provides an unreliable, connectionless datagram service, as described in Chapter 14.

◂ Previous Next ▸



◀ Previous Next ▶

24.1 Overview

The protocol units that exchange TCP instances in this way are called segments, and the protocol units of the IP protocol instances are called IP packets or datagrams.

24.1.1 Requirements on TCP

The TCP protocol was developed in the beginning of the eighties to run on top of the IP protocol and provide a byte-oriented, reliable, connection-oriented transport service. The requirements on such a protocol are as follows [Pete00]:

- to guarantee transmission of byte streams;
- to maintain the transmission order when delivering byte streams;
- congestion: to deliver not more than one single copy of each data unit passed for transmission;
- to transport data for an arbitrary length;
- to support synchronization between sender and receiver;
- to support flow control at the receiver's end; and
- to support several application processes in one system.

To meet these requirements, the TCP protocol provides a reliable, connection-oriented, byte-oriented full-duplex transport service allowing two applications to set up a connection, to send data in both directions reliably, and to finally close this connection. Each TCP connection is set up and terminated gracefully, and all data are delivered before a connection is torn down, provided that the IP protocol behaves in a service-compliant way. From an application's view, the TCP service can be divided into the following properties [Pete00, Come00]:

- Connection orientation: TCP provides connection-oriented service where an application must first request a connection to a destination and then use the connection to transfer data.
- Peer-to-peer communication: Each TCP connection has exactly two endpoints.
- Complete reliability: TCP guarantees that the data sent across a connection will be delivered exactly as sent, with no data missing or out of order.
- Full-duplex communication: A TCP connection allows data to flow in either direction and allows either application program to send data at any time. TCP can buffer outgoing and incoming data in both directions, making it possible for an application to send data and then to continue computation while the data is being transferred.
- Byte-stream interface: We say that TCP provides a stream interface in which an application sends a continuous sequence of octets across a connection. That is, TCP does not provide a notion of records, and does not guarantee that data will be delivered to the receiving application in pieces of the same size in which it was transferred by the sending application.
- Reliable connection startup: TCP requires that, when two applications create a connection, both must agree to the new connection; duplicate packets used in previous connections will not appear to be valid responses or otherwise interfere with the new connection.
- Graceful connection shutdown: An application program can open a connection, send arbitrary amounts of data, and then request that the connection be shut down. TCP guarantees to deliver all the data reliably before closing the connection.

24.1.2 The TCP Packet Format

Figure 24-1 shows how a TCP segment is structured. TCP groups data from higher layers and adds a header, as will be described below:

- The 16-bit SOURCE PORT (SRC PORT) field identifies a process in the sending end system.

◀ Previous Next ▶

24.2 Implementing The TCP Protocol Instance

The protocol instance of the Transmission Control Protocol is one of the most complex parts in the Linux network architecture. The protocol uses a large number of algorithms and features that require extensive mechanisms to implement them. This section explains how these mechanisms are implemented and how they interact in the TCP implementation.

First, we will have a look at "normal" receive and transmit processes in the TCP instance, where we will leave out many details. Too much detail would make it difficult at this point to understand the entire process in the TCP instance and the features of each of the TCP algorithms.

Section 24.3 discusses connection management? how TCP connections are established and torn down; Section 24.4 discusses each of the algorithms used to exchange data (e.g., congestion control and window scaling). Finally, Section 24.5 will introduce the tasks of the TCP protocol instance and how its timers are managed.

The TCP protocol instance is extremely complex. It consists of a large number of functions, inline functions, structures, and macros. In addition, the large number of algorithms used within the TCP protocol makes its description rather difficult. For this reason, we will begin with a general overview of the process involved when receiving, and then when sending, a TCP segment. A detailed discussion of the large number of algorithms used in TCP will follow in Section 24.4. In addition, this section assumes that data is exchanged over an existing connection. The complex management of TCP connections is dealt with in Section 24.3.

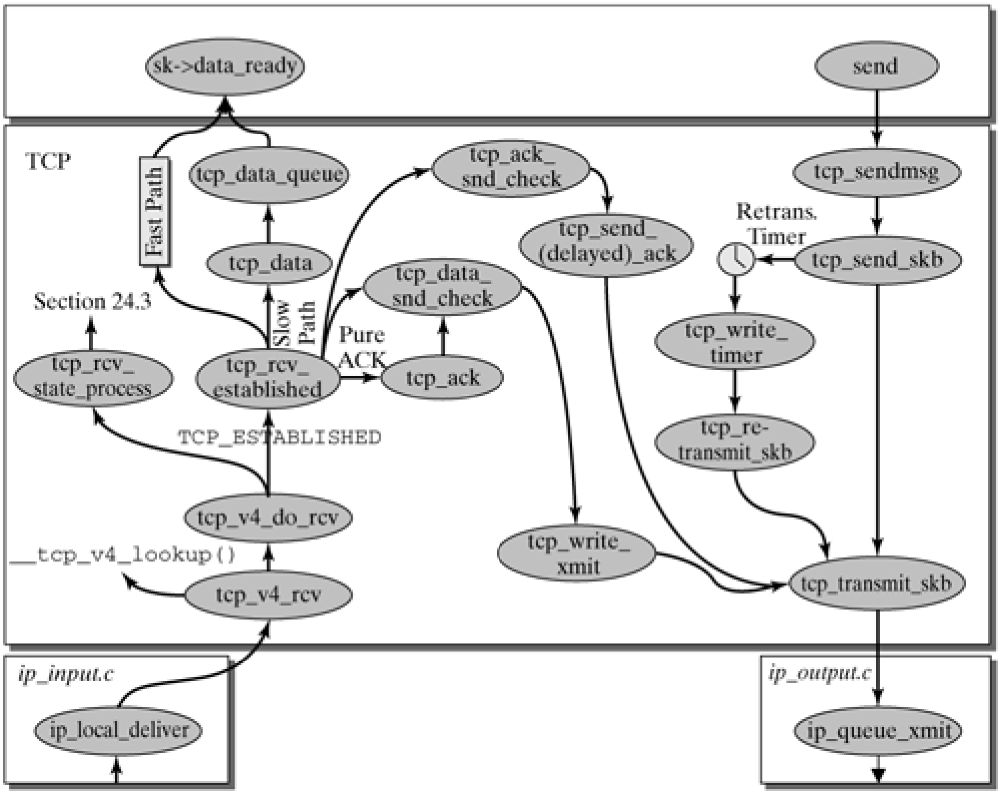
24.2.1 Handling Incoming TCP Segments

The transport protocol for an incoming packet is selected early, by the time it is needed in the IP layer, to be able to pass the packet to the appropriate protocol-handling routine in the transport layer. (See Section 14.2.5.) In the TCP instance, this task is handled by the `tcp_v4_rcv()` function `(net/ipv4/tcp_ipv4.c)`.

Figure 24-2 shows how packets are processed in the TCP instance, and Figure 24-3 gives an overview of what happens when the TCP instance receives a segment.

Figure 24-2. Partial representation of how packets are handled in the TCP instance.

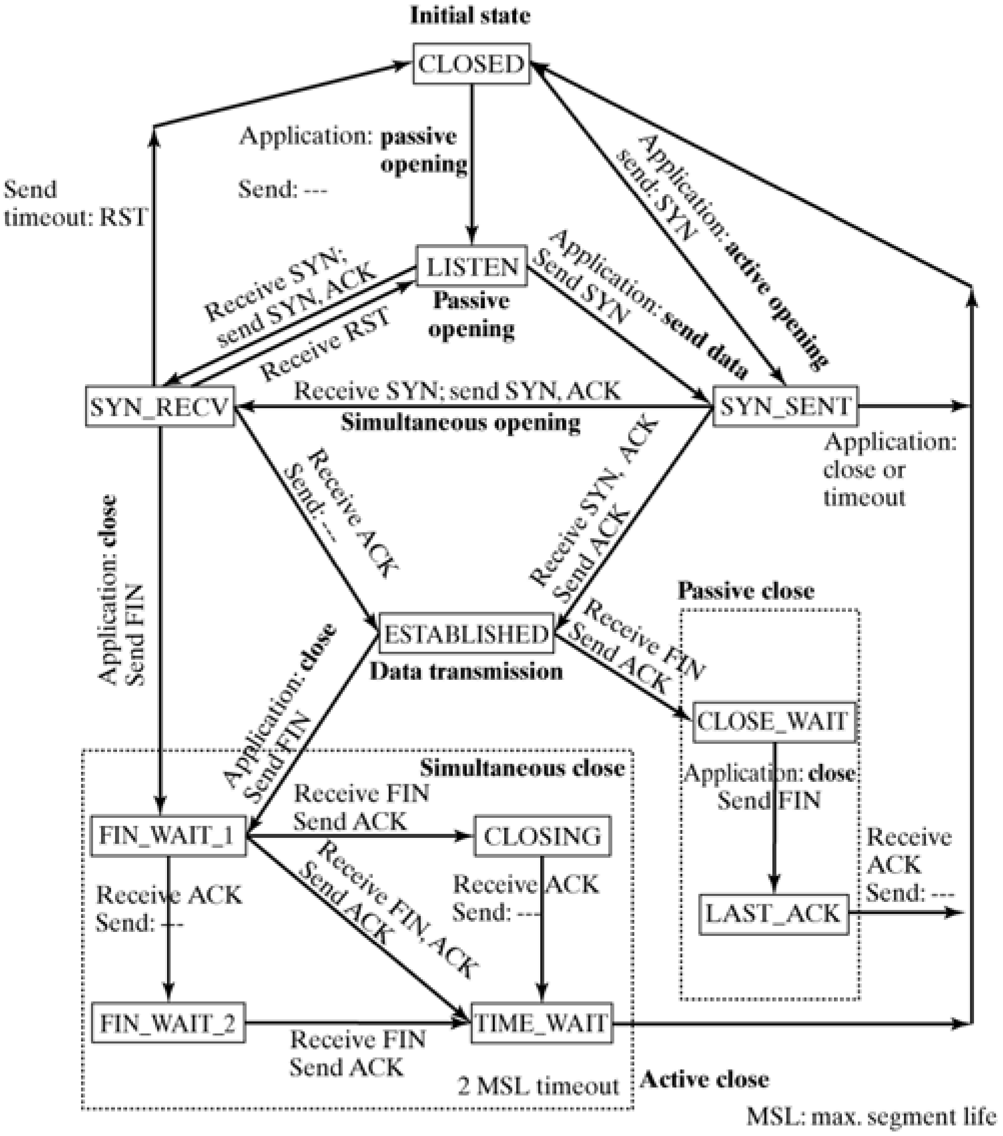
[View full size image]



24.3 Connection Management

Being a connection-oriented protocol that supports a number of additional mechanisms, such as packet transmission in the correct order or urgent data, the TCP protocol is extremely complex. The protocol machine shown in Figure 24-5 is characterized by a total of twelve states. This complexity calls for extensive management of the current state of active connections.

Figure 24-5. The TCP state automaton.



24.3.1 The TCP State Machine

A TCP connection's state is stored in the `state` field of the associated `sock` structure. The response to the receipt of packets is different, depending on the state, so this state has to be polled for each incoming packet. There are three phases: the connection-establishment phase, the data-transmission phase, and the connection-teardown phase. Section 24.4 describes the protocol mechanisms of the data-transmission phase in detail. This section discusses the connection-establishment and connection-teardown phases.

As shown in Section 24.2, `tcp_rcv_state_process() (net/ipv4/tcp_input.c)` is the most important function for connection management, as long as the connection has not yet been established. Packets in the `TIME_WAIT` state are the only packets handled earlier in the `tcp_v4_rcv()` function.

`tcp_rcv_state_process()` net/ipv4/tcp_input.c

◀ Previous Next ▶

24.4 Protocol Mechanisms For Data Exchange

The following subsections introduce several protocol mechanisms of the TCP protocol. However, the TCP protocol uses a large number of algorithms; a detailed description of all of these mechanisms would go beyond the scope and volume of this chapter, but we will discuss selected parts of the TCP instance here.

To begin with, Section 24.4.1 will discuss flow control by use of the sliding-window method. In addition to flow control, we will have a look at the methods for window scaling, zero-window probing, and the PAWS mechanism, including the timestamp option. Subsequently, Section 24.4.2 will discuss methods for detection, handling, and avoidance of congestions: slow-start, congestion-avoidance, fast-retransmit, and fast-recovery methods. Finally, Section 24.4.3 covers methods for load avoidance, concentrating on the Nagle algorithm and the transmission of delayed acknowledgements.

The different timers used by a TCP instance and their management will be discussed in Section 24.5.

24.4.1 Flow Control

The TCP protocol uses flow control to regulate the data flow? the data volume exchanged between a sender and a receiver? on a per-time-unit basis.[1] Flow control limits the number of bytes sent in one communication direction to prevent buffer overflows in the receiving TCP instance and to meet the service consumer requirements. Reasons to limit the data flow by the receiving TCP instance include the following:

- The computing performance of the sending TCP instance can be higher than that of the receiving instance. This means that the sender creates segments faster than the receiving TCP instance can process them. In such a situation, the receive buffer at the receiver's end o ver flows, causing segments to be discarded.
- An application removes data from the socket receive buffer at specific intervals, which means that this buffer empties only occasionally. Examples include applications that output multimedia contents, receiving contents faster than their playback rate.

Consequently, flow control can be used to prevent the receive buffer of a receiving TCP instance from overflowing, which would cause additional incoming packets to be dropped. To implement flow control, the TCP protocol uses the sliding-window mechanism. This mechanism is based on the assignment of explicit transmit credits by the receiver [Pete00]. We will introduce it in the following section.

The Sliding-Window Mechanism

The sliding-window protocol mechanism is used commonly in transport protocols or connection-oriented protocols, because it provides for three important tasks:

- The original order of a set of data segmented and sent in several packets can be restored in the receiver.
- Missing or duplicate packets can be identified by ordering of packets. Together with additional packet-retransmission methods, this enables us to guarantee reliable data transport.
- The data flow between two TCP instances can be controlled by assigning transmit credits. Specifically, it is distinguished between a fixed credit quantity (e.g., in HDLC) and explicit credit assignment (e.g., in TCP).

The following elements are added to the protocol header (using the TCP protocol as our example) to handle these tasks:

- All data is numbered consecutively by sequence numbers. The sequence numbers of the first payload byte in a data packet is carried in the packet header and denotes the sequence number of this segment. The `tp->snd_nxt` variable in the `tcp_opt` structure stores the

[1] This section considers flow control in only one direction of a TCP connection and treats the two TCP instances as a sender and a receiver. In bidirectional data exchange, the flow is controlled separately for each direction, where each instance assumes both the role of a sender and the role of a receiver.

◀ Previous Next ▶

24.5 Timer Management In TCP

In closing, this section briefly discusses how timers are managed in TCP. Timers are used in different positions within the TCP protocol to control retransmissions and to limit the hold time for missing packets.

24.5.1 Data Structures

`struct timer_list` include/linux/timer.h

```
struct timer_list {
       struct list_head list;
       unsigned long expires;
       unsigned long data;
       void (*function) (unsigned long);
       volatile int running;
};
```

The basis for each timer is the `jiffies` variable. As described in Chapter 2, it is updated by Linux every 10 ms.

A timer structure includes a function pointers that takes a behavior function when initialized. This function is invoked when the timer expires. The time at which it expires depends on the `expires` field. This field takes an offset (in the `jiffies` unit) for the current time (also in the `jiffies` unit). The behavior function is invoked when this value is reached.

TCP maintains seven timers for each connection:

- `SYNACK` timer This timer is used when a TCP instance changes its state from `LISTEN` to `SYN_RECV.` The TCP instance of the server initially waits three seconds for an ACK. If no ACK arrives within this time, then the connection request is considered outdated.

- `Retransmit` timer Because the TCP protocol uses only positive acknowledgements, the sending TCP instance has to see for itself whether a segment was lost. It does this by use of the `retransmit` timer, the expiry of which indicates that a segment could have been lost, causing its retransmission.

  The exponential backoff method assumes that retransmissions are caused by a congestion. When segments are retransmitted, the timer value is increased exponentially so as to be able to detect segment losses.

The `retransmit` timer determines when packets have to be retransmitted during a data transmission phase. This value depends on the round-trip time and normally is within the range from 200 ms to two minutes.

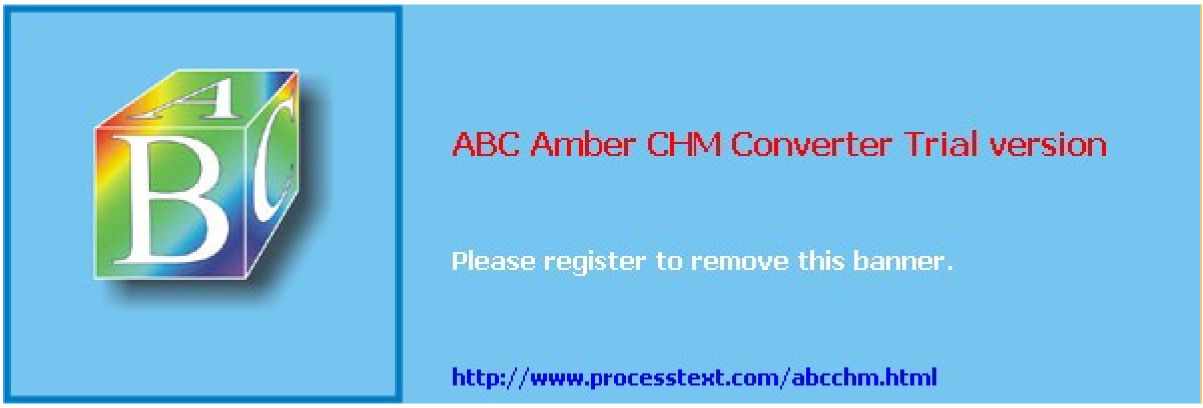
This timer is also used during the establishment of a connection. It is initialized to three seconds. Upon expiry of this time, the backoff mechanism is used five times.

- `Delayed ACK` timer This timer delays the transmission of ACK packets. The value is smaller than 200 ms.

- `Keepalive` timer This timer is used to test whether a connection is still up. It is invoked for the first time after nine hours. Subsequently, nine probes are sent every 75 seconds. If all probes fail, the connection is reset.

- `Probe` timer This timer is used to test for a defined time interval, to see whether the zero window still applies. The value depends on the round-trip time.

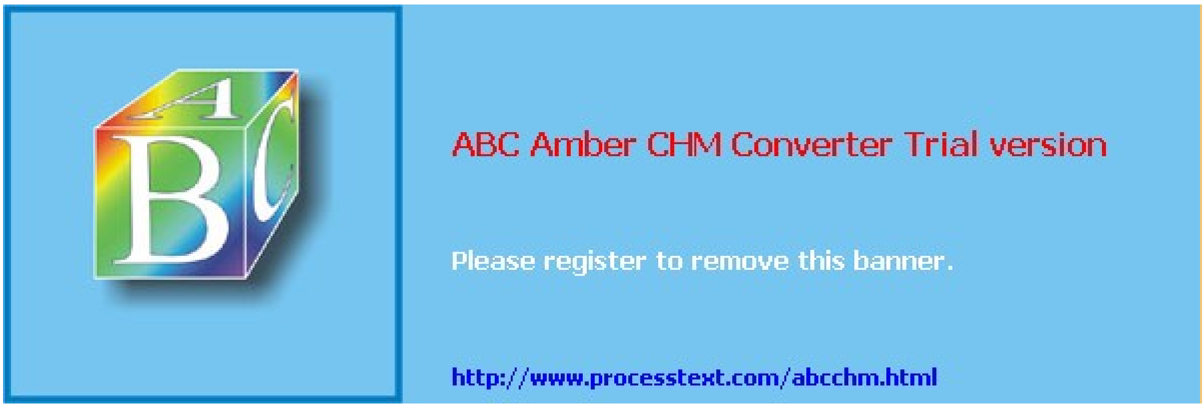
- `FinWait2` timer The expiry of this timer switches the connection from the `FIN_WAIT2` state into the `CLOSED` state, if no `FIN` packet from the partner arrives.



Previous Next

Chapter 25. User Datagram Protocol (UDP)

Previous Next



◂ Previous Next ▸

25.1 Introduction

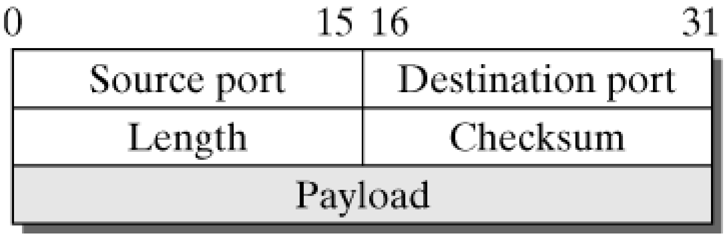
The User Datagram Protocol (UDP) is described in RFC 768 [Post80] and represents a minimal transport protocol. It runs on top of the Internet Protocol (IP) and essentially offers the same functionality as IP itself: an unreliable, connectionless datagram service. In this case, unreliable means that there are no mechanisms to detect and handle lost or duplicate packets. However, packets can be optionally protected against bit errors by using a checksum, which covers both the packet header and the payload of each packet, in contrast to IP. Otherwise, there is only one additional option, compared to IP: Port numbers can be used to address different applications in a specific end system.

On the one hand, UDP is used for transaction-oriented applications, such as the Domain Name System (DNS), where only one request and the relevant reply have to be transmitted, so that it is not worthwhile establishing a connection context, which would mean, for example in TCP, that three additional messages for the establishment and four messages for the tear-down were required. On the other hand, UDP is also used where the reliability of a transmission plays a secondary role, because one is primarily interested in transmitting data easily and quickly. For example, it is normally not a problem if some packets are lost when audio streams are transmitted in small packets. On the contrary, an automatic flow and error control with retransmission of lost packets would often be disturbing to the smooth playback of the stream.

25.1.1 Packet Format

Figure 25-1 shows the format of UDP packets. The header fields are briefly described below.

Figure 25-1. UPD packet format.



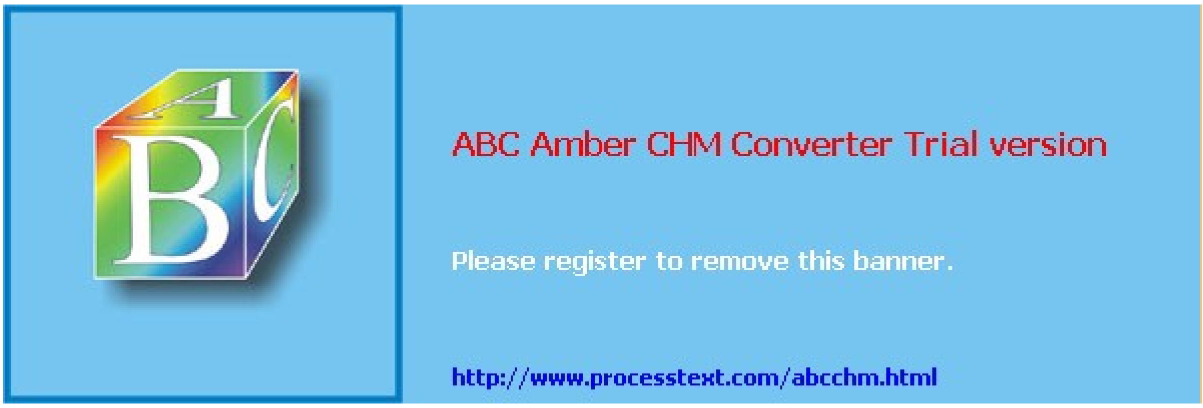
- Source port: The source port is the port number used by the sending process, in the range from 1 to 65535; normally, the receiver of a request sent over UDP will direct its reply to this port. RFC 768 specifies that giving the source port number is optional, and the field can have the value zero, if it is not used. However, for UDP over the socket programming interface in Linux (see Chapters 26 and 27), this is not possible, because a port number different from zero is automatically assigned to each socket, if the user does not state one.

- Destination port: The destination port is used to address the application in the destination system that is to receive a UDP packet.

- Length: The length is specified in octets and refers to the entire UDP packet, consisting of packet header and payload. The smallest possible length is therefore eight octets, and the largest possible UDP packet can transport 65535 ?8 = 65527 payload octets.

- Checksum: As in TCP, the calculation of the checksum includes a pseudo header, in addition to the packet header and the payload. The format of this pseudo header is shown in Figure 25-2 . It includes the IP source and destination addresses, the UDP protocol identifier (17), and the length of the UDP packet. The checksum is computed as a 16-bit ones complement of the one s-complement sum over the data mentioned above, where a zero octet is appended if the octet number is uneven. This method can be implemented efficiently for all processor types (as described in RFCs 1071, 1141, and 1624 [BrBP88, MaKu90, Rijs94]). If the computation results in the checksum zero, the all-1-bit value is transmitted instead, which is equivalent in ones-complement arithmetic. A zero in the checksum field means that the sender has not computed a checksum.

Figure 25-2. Pseudo header format for checksum calculation.


ABC Amber CHM Converter Trial version
Please register to remove this banner.
http://www.processtext.com/abcchm.html

◂ Previous Next ▸

25.2 Data Structures

The implementation of UDP in the Linux kernel does not require any additional or particularly complex data structures. This section describes the data structure used to pass payload at the socket interface, the UDP datagram itself, which is included in the general socket buffer structure, and the data structure instances used to integrate the protocol into the network architecture.

25.2.1 Passing the Payload

The payload is given for the `sendmsg()` system call at the socket interface in the form of an `msghdr` structure, which is checked by the socket interface and copied into the kernel (except for the actual payload that initially remains in the user address space). Otherwise, the structure is passed, as is, to the `udp_sendmsg()` function for sending UDP packets.

`struct msghdr` include/linux/socket.h

```
struct msghdr {
       void                   *msg_name;
       int                    msg_namelen;
       struct iovec           *msg_iov;
       __kernel_size_t        msg_iovlen;
       void                   *msg_control;
       __kernel_size_t        msg_controllen;
       unsigned               msg_flags;
};
```

For sending of UDP packets, `msg_name` is not really a name, but a pointer to a `sockaddr_in` structure (see Section 27.1.1), which contains an IP address and a port number; `msg_namelen` describes the length of this structure. The `msg_iov` pointer refers to an array of `iovec` structures, which reference the payload. This means that this payload can be present in a series of individual blocks, where each block is denoted in an `iovec` structure by its initial address `(iov_base)` and its length `(iov_len)`:

```
struct iovec
{
       void                    *iov_base;
       __kernel_size_t         iov_len;
};
```

The buffer specified by `msg_control` and `msg_controllen` can be used to pass protocol-specific control messages. We will not discuss the format of these messages; see detailed information in the `recv()` system call manpage.

The `msg_flags` element can be used to pass different flags both from the user process to the kernel and in the opposite direction. For example, the kernel evaluates the following flags:

- `MSG_DONTROUTE` specifies that the destination must be in the local area network and that, for this reason, the datagram should not be sent over a router to its destination.
- `MSG_DONTWAIT` prevents the system call from blocking if, for example, there are no data to be received.
- `MSG_ERRQUEUE` means that no packet should be fetched, but instead a detailed error message, which might be available at the socket.

The following flag is an example of flags returned by the kernel to the user process:

- `MSG_TRUNC` indicates that the buffer space provided for receiving was insufficient, so that some of the packet data were lost.

◂ Previous Next ▸

25.3 Sending and Receiving UDP Datagrams

The sending of UDP packets, starting from the system call at the socket interface and running all the way until the completed packet is added to the output queue of the network interface, is handled in just one pass, but the receiving of UDP packets requires two separate steps: Once a packet has been received, `udp_rcv()` first allocates it to a socket in bottom-half context and places it into that socket's receive queue. From there, the packet is fetched via system call of a user process, which is mapped to `udp_recvmsg()`.

Sending UDP Datagrams

`udp_sendmsg()` net/ipv4/udp.c

The function `udp_sendmsg()` is invoked over the socket interface whenever a UDP packet has to be sent: The different kinds of systems calls all lead to this single function's being called. Its parameters are a `sock` structure with the state of the sending `PF_INET` socket, a pointer to a `msg` structure that specifies the receiver and payload, and the payload length in octets.

First, a locally created `udpfakehdr` structure takes the destination address and the destination port from the `msg_name` element of the `msg` structure. The destination doesn't have to be specified explicitly only if a default destination address has previously been assigned to this socket via `udp_connect()`. In this case, the information from the `sock` structure is used instead. The source address and the source port always derive from the `sock` structure. Additionally, they are stored in an `ipcm_cookie` structure. This structure, which we will not describe here in detail, serves later on to pass the addresses, the device identifier, and the IP options (if applicable) to the Internet Protocol.

Any control messages in the `msg_control` element of the `msghdr` structure are processed by calling the function `ip_cmsg_send()`, and the results are registered in the `ipcm_cookie` structure. Control messages can be used to modify the source address or to pass IP options, which will then also be registered in the `ipcm_cookie` structure. If no IP options are specified, then the IP options stored in the `sock` structure, if applicable, will be used.

A routing cache entry has to be procured, so it is also necessary to handle the source routing IP options beforehand; the address of the first intermediate station might need to be used instead of the destination address.

If the socket had previously obtained a routing cache entry by `udp_connect()`, and if the corresponding destination address has not yet been changed in the process of `udp_sendmsg()`, then this routing cache entry is now checked. If this check produces a negative result, or if the destination address was changed, then `ip_route_output()` is used to procure a new routing cache entry (which is then stored in the `sock` structure).

Eventually, the transmission of data is initiated by calling `ip_build_xmit()`, where either `udp_getfrag()` or `udp_getfrag_nosum()` is provided as the callback function for getting data, depending on whether the checksum in the packet header should include the payload.[1] The parameters used here also include the `udpfakehdr` and `ipcm` structures, the routing cache entry, the flags from the `msghdr` structure, and the total length of the UDP packet.

[1] A checksum is computed if the flag no_check in the sock structure is null, which is the case by default. This flag can be set via the socket option SO_NO_CHECK on the SOL_SOCKET level.

The following discussion considers only `udp_getfrag()`, because `udp_getfrag_nosum()` provides the same functionality, but is simpler for omitting the checksum calculation.

`udp_getfrag()` net/ipv4/udp.c

For each IP fragment it generates, `ip_build_xmit()` invokes the callback function `udp_getfrag()` to get the required payload. A pointer to the `udpfakehdr` structure filled by

◂ Previous Next ▸

Chapter 26. The Concept of Sockets

◂ Previous Next ▸

◂ Previous Next ▸

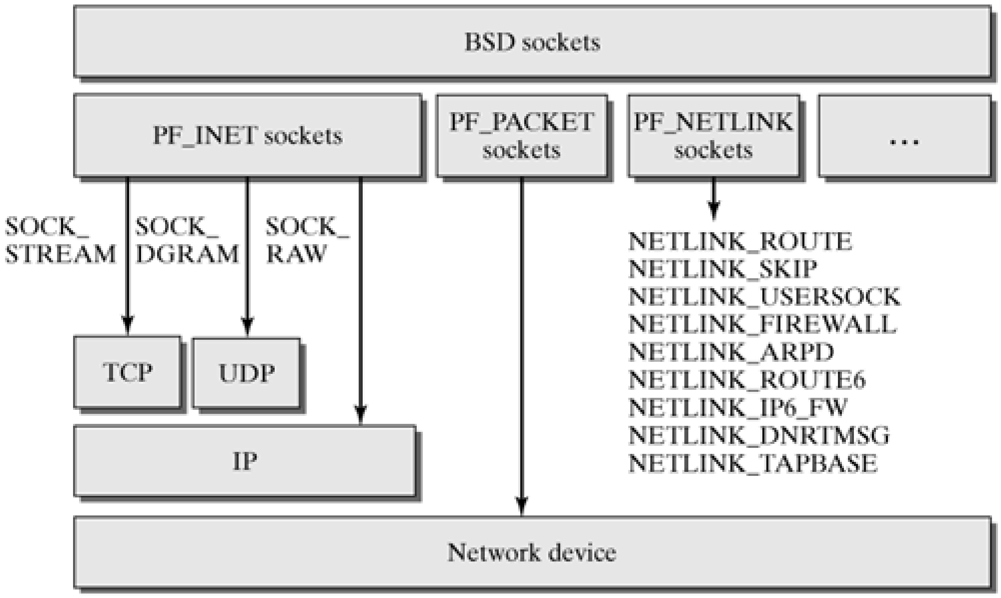
26.1 Introduction

The abstraction of sockets was introduced (based on the BSD version of UNIX, where this interface was used for the first time; also called BSD sockets) to facilitate programming of networked applications. An application can use this uniform interface to send or receive data over a network. This interface looks alike for all protocols, and the desired protocol is selected to match three parameters: `family`, `type`, and `protocol`. Chapter 27 gives a complete overview of all available protocol families (`family`). It also discusses how applications can use the socket interface. In contrast, this chapter gives an overview of the socket implementation in the Linux kernel.

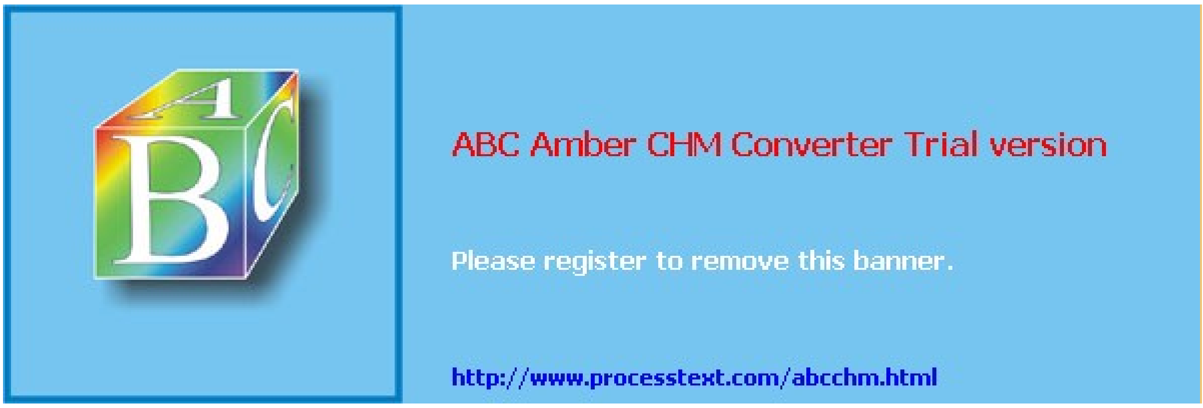
Figure 26-1 gives an overview of how the socket support is integrated into the protocol implementations in the Linux kernel. The BSD socket interface provides a uniform interface upwards to the applications; underneath the interface, however, different protocol families are distinguished by protocol-specific sockets. Currently, one of the most important protocol families, `PF_INET` (protocol family internet) will be described in the following section. In addition, `PF_PACKET` sockets in more recent Linux versions provide an elegant way to send data packets by directly accessing a network device. For example, the use of the packet socket was introduced in Chapter 9. Section 26.3.2 describes the packet socket in more detail. In contrast, the Netlink sockets do not serve for data transmission over a network, but to configure various parts of the Linux network architecture. The part to be configured is selected over the parameters `NETLINK_*` of the socket's `protocol` variable, as shown in Figure 26-1. The third part of this chapter describes the `PF_NETLINK` sockets.

Figure 26-1. Structure of the socket support in the Linux kernel.

[View full size image]



◂ Previous Next ▸



◂ Previous Next ▸

26.2 BSD Sockets

The Linux kernel offers exactly one socket-related system call, and all socket calls of applications are mapped to this system call. The function `asmlinkage long sys_socketcall(int call, unsigned long *args)` is defined in `net/socket.c`. Moreover, a number is assigned in `include/asm/unistd.h (#define __NR_socketcall 102)` and added to a table with system calls in `arch/i386/kernel/entry.S`. The socket function to be addressed can be stated in the `call` parameter of a call. The admissible parameters are defined in `include/linux/net.h`: `SYS_SOCKET, SYS_BIND, SYS_CONNECT, SYS_LISTEN, SYS_ACCEPT, SYS_GETSOCKNAME, SYS_GETPEERNAME, SYS_SOCKET-PAIR, SYS_SEND, SYS_RECV, SYS_SENDTO, SYS_RECVFROM, SYS_SHUTDOWN, SYS_SETSOCKOPT, SYS_GETSOCKOPT, SYS_SENDMSG, SYS_RECVMSG`. From within libraries in the user space, the `sys_socketcall()` call with a specific parameter is mapped to an independent function (e.g., `sys_socketcall (SYS_SOCKET ,...)` becomes the call `socket(...)`).

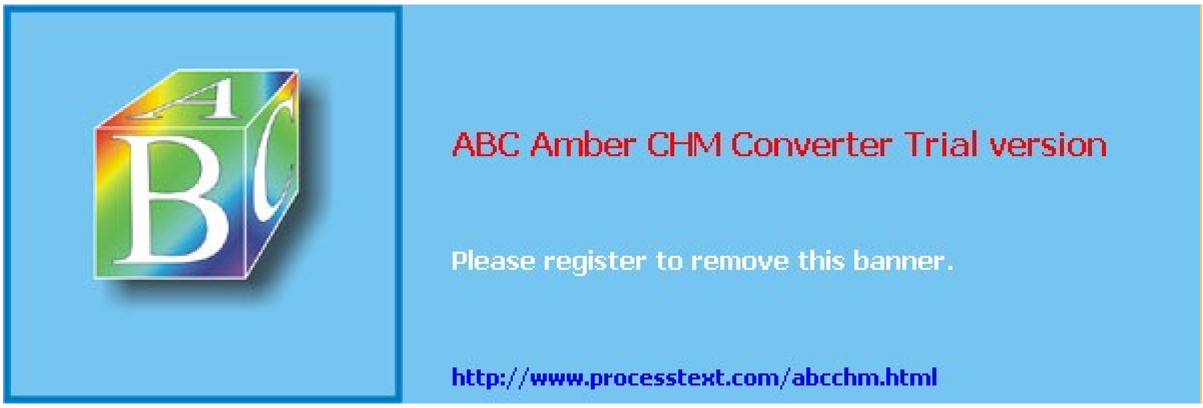
`sys_socketcall()` net/socket.c

The function to be called is selected in the kernel by using a `switch` command in the function `sys_socketcall()`, and the command `copy_from_user()` is used to first copy the function's arguments into a vector, `unsigned long a[6]`:

```
...
if copy_from_user(a, args, nargs[call]))
       return -EFAULT;
a0=a[0];
a1=a[1];
switch(call)
      {
       case SYS_SOCKET:
            err = sys_socket(a0,a1,a[2]);
            break;
       case SYS_BIND:
            err = sys_bind(a0, (struct sockaddr *)a1, a[2]);
            break;
       case SYS_CONNECT:
            err = sys_connect (a0, (struct sockaddr *)a1, a[2]);
            break;
       case SYS_LISTEN:
            err = sys_listen (a0,a1);
            break;
       ...
       }
...
```

The most important structure within the BSD socket support is struct socket. It is defined in `include/linux/net.h`:

```
struct socket {
       socket_state            state;
       unsigned long           flags;
       struct proto_ops        *ops;
       struct inode            *inode;
       struct fasync_struct    *fasync_list; /* Asynchronous wakeup list*/
       struct file             *file;        /* File back pointer for gc*/
       struct sock             *sk;
       wait_queue_head_t       wait;

       short                   type;
```



◂ Previous Next ▸

26.3 Protocol-Specific Sockets

The central structure of all protocol-specific sockets underneath the BSD sockets is `struct sock`. This structure was oriented to TCP/UDP and IP in earlier kernel versions. Along with the addition of other protocols (e.g., ATM), the `sock` structure was extended, and other entries were partially removed from the structure. Initially, this created an extremely unclear construction having a number of entries needed for only a few protocols. Together with the introduction of the three union structures (`net_pinfo, tp_pinfo`, and `protinfo`), each of which contains a reference to protocol options of the matching layer, this situation should gradually improve, and the structure should become easier to understand. The structure is still rather extensive, but we will introduce only the entries of interest in the following sections.

26.3.1 `PF_INET` Sockets

This section describes the initialization on the level of `PF_INET` sockets when an application uses a `socket()` call.

`inet_create()` net/ipv4/af_inet.c

As has been mentioned, this function is invoked by the function `sock_create()` to initialize the `sock` structure. Initially, the state of the BSD socket is set to `SS_UNCONNECTED`, and then the function `sk_alloc()`, which was described in Chapter 4, allocates a sock structure. The protocol family can only be `PF_INET` at this point, but we still have to distinguish by `type` and `protocol`. This is now done by comparing against the information in a list, `struct inet_protosw inetsw_array`, which is created by `inet_register_protosw() (net/ipv4/af_inet.c)` when the kernel sta rts.

Next, the `ops` field of the BSD socket structure can be filled. The `sk` pointer connects the BSD socket to the new `sock` structure, and the latter is connected to the BSD socket by the `socket` pointer. (See `sock_init_data() in net/core/sock.c`.) Similarly to the `proto_ops` structure, the `proto` structure supplies the functions of the lower-layer protocols:

```
struct proto {
void          (*close)          (...);
int           (*connect)        (...);
int           (*disconnect)     (...);
struct sock*  (*accept)         (...);
int           (*ioctl)          (...);
int           (*init)           (...);
int           (*destroy)        (...);
void          (*shutdown)       (...);
int           (*setsockopt)     (...);
int           (*getsockopt)     (...);
int           (*sendmsg)        (...);
int           (*recvmsg)        (...);
int           (*bind)           (...);
int           (*backlog_rcv)    (...);
void          (*hash)           (...);
void          (*unhash)         (...);
int           (*get_port)       (...);
char          name [32];
              struct {
                     int inuse;
                     u8 __pad[SMP_CACHE_BYTES - sizeof(int)];
                     } stats[NR_CPUS];
};
```

Consequently, the `proto` structure represents the interface from the socket layer to the transport protocols. The `hash()` and `unhash()` functions serve to position or find a `sock` structure in a hash

◂ Previous Next ▸

Part VI: Layer V? Application Layer

◂ Previous Next ▸

◀ Previous Next ▶

Chapter 27. Network Programming With Sockets

◀ Previous Next ▶

◂ Previous Next ▸

27.1 Introduction

An application programming interface (API) is required to enable application programmers to access the network functionality implemented in the operating system. One of the most common interfaces to access transport protocols in the UNIX domain is Berkeley sockets or BSD sockets, which obtained their names from the UNIX variant Berkeley Software Distribution, where they were implemented for the first time.

The design of Berkeley sockets (in the following discussion called sockets for short) follows the UNIX paradigm: Ideally, map all objects that are read or write accessed to files, so that they can be processed by use of the regular file write and read operations. Sending or receiving in a communication relationship can be easily mapped to write and read operations. The objects manipulated by such operations in the context of transport protocols are the endpoints of a communication relationship; these are represented by sockets.

27.1.1 Socket Addresses

A communication endpoint in the transport layer is described by three parameters in the Internet: the protocol used, an IP address, and a port number. These parameters therefore have to be allocated to a socket, before it can be used for communication. In building a communication relationship, we additionally have to specify the communication partner's endpoint address.

`struct sockaddr` /usr/include/sys/socket.h

The data structure used to represent socket addresses was kept quite general, because the socket interface can also support other protocols, in addition to Internet protocols:

```
typedef unsigned short sa_family_t;

struct sockaddr
{
       sa_family_t         sa_family;
       char                sa_data[14];
};
```

The `sa_family` element registers the address family (e.g., `AF_INET` for the family of Internet protocols). The exact address format is not yet defined in detail in the general `sockaddr` structure. For this reason, there is a more specific variant for Internet addresses, called `sockaddr_in`.

`struct sockaddr_1n` /usr/include/netinet/in.h

```
struct in_addr {
                    __u32 s_addr;
};

struct sockaddr_in   {
   sa_family_t             sin_family;      /* Address family: AF_INET */
   unsigned short int      sin_port;        /* Port number */
   struct in_addr          sin_addr;        /* Internet address */

   /* Pad to size of 'struct sockaddr' . */
   unsigned char sin_zero[sizeof (struct sockaddr) -
                          sizeof (sa_family_t) -
                          sizeof (uint16_t) -
                          sizeof (struct in_addr)];
};
```

◂ Previous Next ▸

27.2 Functions of the Socket API

This section gives an overview of the most important functions for the socket application programming interface. The original TCP/IP implementation of the BSD UNIX version, which the socket API is derived from, used only the six system calls of the input and output interface for file operations to communicate over networks. Only the next version added the whole number of additional operations that will be discussed below.

27.2.1 Functions for Connection Management

The functions described in this section serve to manage communication relationships: to create and delete sockets, to open and close connections, and so on.

```
int socket (int family, int type, int protocol)
```

Berkeley sockets, or sockets for short, are the basis for communication relationships over the socket interface. A socket represents the endpoint of a communication relationship in an end system and forms the interface between the network protocols and applications. This means that, in a communication using the TCP protocol, the two sockets in the two communicating end systems form the endpoints for this communication. In a multicast communication, more than two sockets normally participate in a communication.

An application can use the `socket()` system call to cause the operating system to create a socket, always the first step in communicating over networks. In the creating of a socket, the required resources are reserved in the operating system, and the type of communication protocol to be used is determined (e.g., TCP or UDP).

The result of a `socket()` system call consists of the socket descriptor? an integer number that uniquely identifies the socket. This descriptor has to be used in all subsequent system calls to identify the socket.

The following parameters are passed with the `socket()` system call:

- `int family` denotes the protocol family used and thus, mainly, the address type used. Constants for the following address families (`AF_...`) are defined:
    - `AF_UNIX`: Sockets for interprocess communication in the local computer.
    - `AF_INET`: Sockets of the TCP/IP protocol family based on the Internet Protocol Version 4
    - `AF_INET6`: TCP/IP protocol family based on the new Internet Protocol, Version 6. (See Chapter 23.)
    - `AF_IPX`: IPX protocol family.
- `int type` denotes the type of the desired communication relationship. Within the TCP/IP protocol family, we mainly distinguish the following three types:
    - `SOCK_STREAM` (stream socket) specifies a stream-oriented, reliable, in-order full duplex connection between two sockets.
    - `SOCK_DGRAM` (datagram socket) specifies a connectionless, unreliable datagram service, where packets may be transported out of order.
    - `SOCK_RAW` (raw socket).
- `int protocol` selects a protocol for the specified socket type, if several protocols with the specified type properties are available. In the `AF_INET` address family, TCP is always selected for the `SOCK_STREAM` socket type, and UDP is always used as the transport protocol for `SOCK_DGRAM`.

◀ Previous Next ▶

27.3 Examples

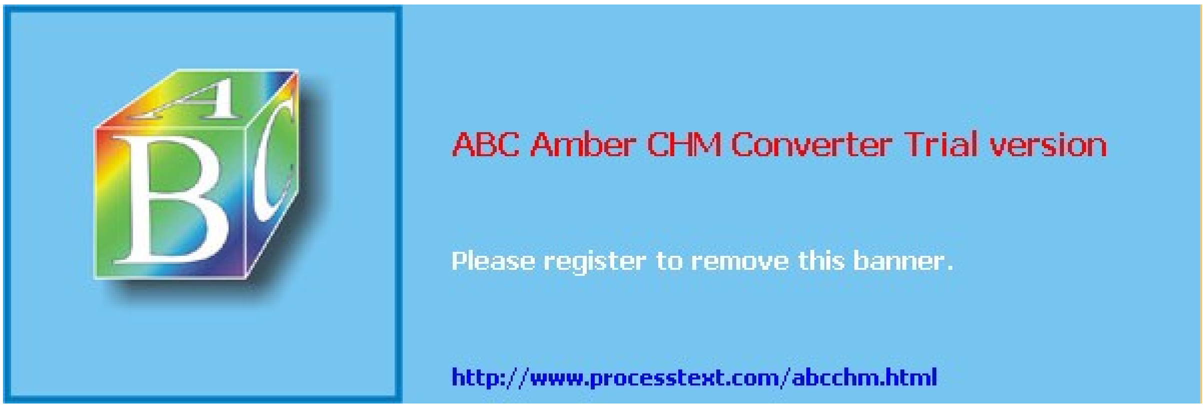
The source text for a complete small sample application, where a client and a server communicate over TCP, is included in Appendix G. A much more detailed description of network programming in UNIX operating systems, including many examples, is found in, for example, [Stev90].

◀ Previous Next ▶

Previous Next

Part VII: Appendices

Previous Next



◂ Previous Next ▸

Appendix A. The LXR Source-Code Browser

The Linux kernel in Version 2.4.9 consisted of 9,837 files, totaling to approximately 3,857,319 lines of source code. Even modifications from one version to the next comprise several megabytes. This volume of Linux source code makes it difficult to navigate to the desired place right away.

To facilitate working with the source code of the Linux kernel, the University of Oslo developed the LXR source-code browser[1] (`http://lxr.linux.no`). This browser is Web-based and represents the source code of the Linux kernel in HTML (HyperText Markup Language). The benefit of HTML, compared to normal source code, is the possibility of integrating hyperlinks that let you reference further information at specific positions, so that navigation between related source-code sections becomes very easy.

[1] LXR is short for Linux Cross (X) Reference.

◂ Previous Next ▸

◀ Previous Next ▶

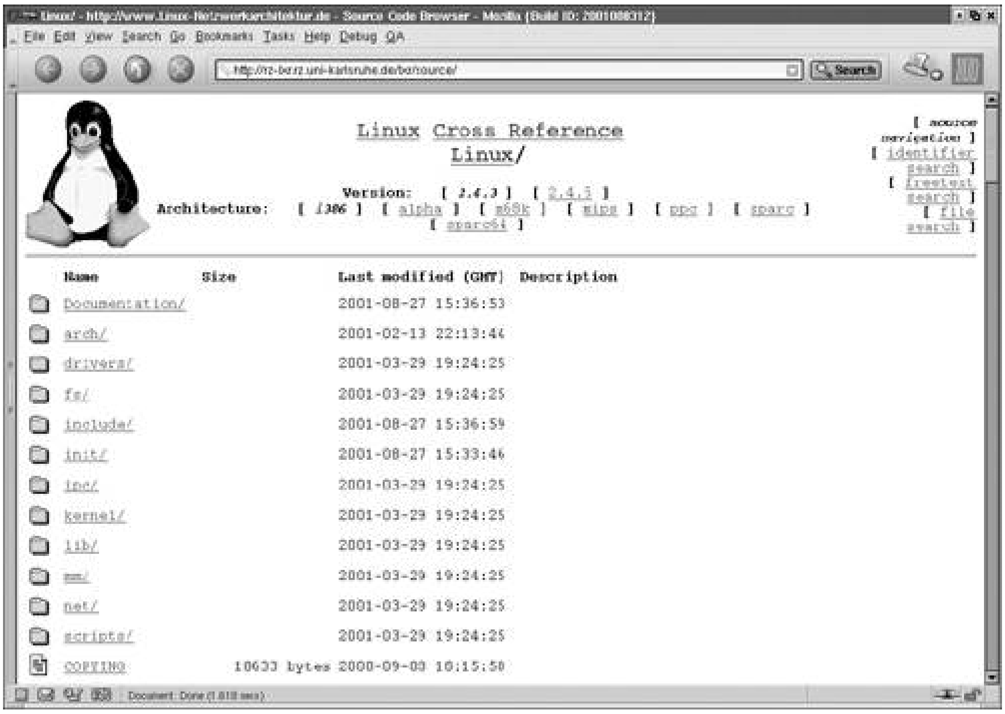
A.1 Functionality

The LXR source-code browser creates a reference file for existing C/C++ source texts. This reference file stores links between different parts of the source code. For initializing of the LXR, the source files are searched for certain keywords, and an index of these keywords is created. The following elements of a C program are recognized as keywords, and their functions are interpreted accordingly:

- Functions and function prototypes;
- global variables;
- type definitions (`typedef`);
- structure definitions (`struct`);
- variant definitions (`union`);
- enumeration types (`enum`); and
- preprocessor macros and definitions (`#define`).

Figure A-1. Browsing the Linux kernel in the LXR source-code browser.

[View full size image]


Linux/ - http://www.Linux-Netzwerkarchitektur.de - Source Code Browser - Mozilla {Build ID: 2001080312}
File Edit View Search Go Bookmarks Tasks Help Debug QA
http://rz-bcr2.uni-karlsruhe.de/bcr/source/
Search
Document: Done (1.818 secs)

```
                    Linux Cross Reference
                          Linux/

                  Version:    [ 2.4.3 ]  [ 2.4.5 ]
Architecture:  [ i386 ]  [ alpha ]  [ m68k ]  [ mips ]  [ ppc ]  [ sparc ]
                              [ sparc64 ]

                                              [ source navigation ]
                                              [ identifier search ]
                                              [ freetext search ]
                                              [ file search ]

   Name              Size         Last modified (GMT)  Description
   Documentation/                 2001-08-27 15:36:53
   arch/                          2001-02-13 22:13:44
   drivers/                       2001-03-29 19:24:25
   fs/                            2001-03-29 19:24:25
   include/                       2001-08-27 15:36:59
   init/                          2001-08-27 15:33:46
   ipc/                           2001-03-29 19:24:25
   kernel/                        2001-03-29 19:24:25
   lib/                           2001-03-29 19:24:25
   mm/                            2001-03-29 19:24:25
   net/                           2001-03-29 19:24:25
   scripts/                       2001-03-29 19:24:25
   COPYING           10633 bytes  2000-09-03 10:15:50
```

Subsequently, all keywords occurring in the source code are stored in an index file.

When a Web browser requests one of the source files, a Web page consisting of the original source-code file with all keywords emphasized by hyperlinks (see Figure A-2) is created. By clicking one of the links, another page is generated, which shows all information about this keyword. (See Figure A-3.) When a function is called, for example, the location (file and line number) of the function declaration, the actual function itself, and all locations where this function is invoked are displayed. Hyperlinks offer an easy way to jump to these locations. Figure A-2 uses the `ip_rcv()` function as an example, to show how this works.

Figure A-2. Browsing the Linux kernel in the LXR source-code browser, using the `ip_rcv()` function as example.

[View full size image]

Linux/net/ipv4/ip_input.c - http://www.Linux-Netzwerkarchitektur.de - Source Code Browser - Mozilla {Build ID: 2001080312}
File Edit View Search Go Bookmarks Tasks Help Debug QA

◀ Previous Next ▶

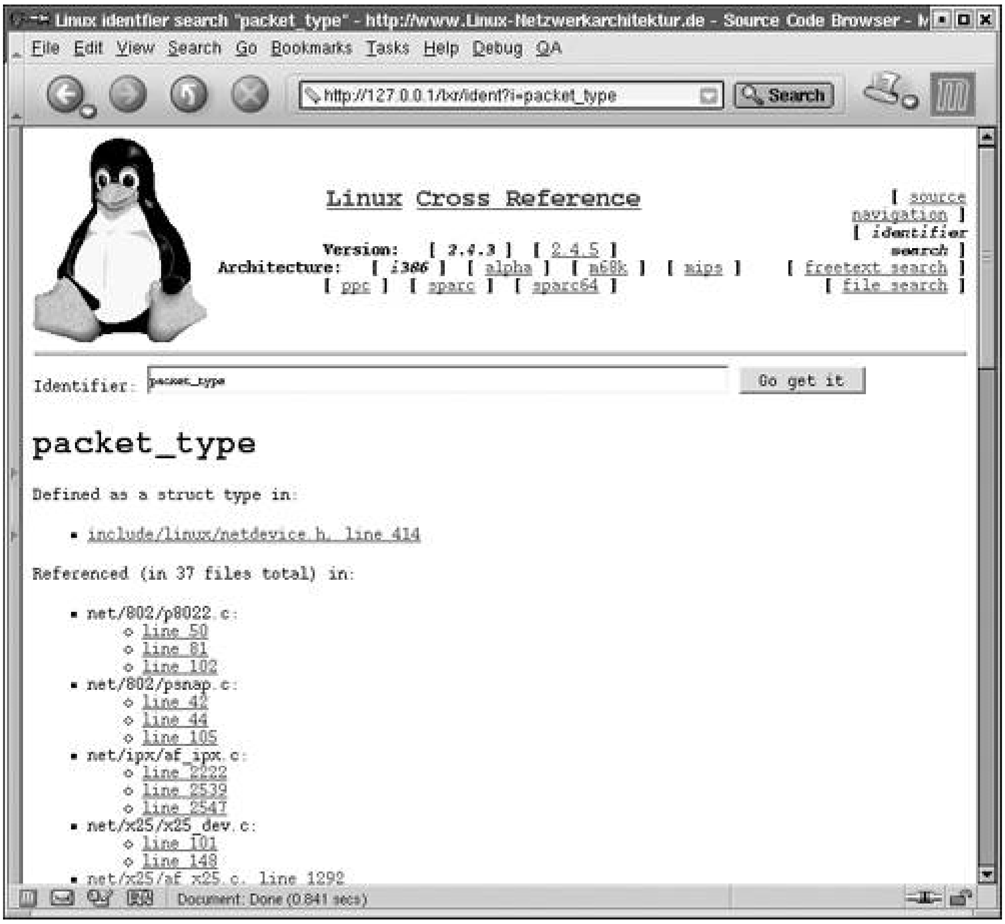
A.2 Installation

Installing the LXR source-code browser is relatively easy for an experienced Linux user. Problems or suggestions for improvement can be published in a mailing list. The following components are required to install the LXR source-code browser:

- The LXR package with the scripts to generate the source-code index and the HTML pages. The version currently most stable is `lxr-0.3`. It can be downloaded from http://lxr.sourceforge.org.

Figure A-4. Information stored for the C structure `packet_type`.

[View full size image]



Linux identfier search "packet_type" - http://www.Linux-Netzwerkarchitektur.de - Source Code Browser - M

File Edit View Search Go Bookmarks Tasks Help Debug QA
http://127.0.0.1/lxr/ident?i=packet_type
Search

Linux Cross Reference

Version: [*2.4.3*] [2.4.5]
Architecture: [*i386*] [alpha] [m68k] [mips] [ppc] [sparc] [sparc64]

[source navigation] [*identifier search*] [freetext search] [file search]

Identifier: packet_type Go get it

packet_type

Defined as a struct type in:

- include/linux/netdevice.h, line 414

Referenced (in 37 files total) in:

- net/802/p8022.c:
  - line 50
  - line 81
  - line 102
- net/802/psnap.c:
  - line 42
  - line 44
  - line 105
- net/ipx/af_ipx.c:
  - line 2222
  - line 2539
  - line 2547
- net/x25/x25_dev.c:
  - line 101
  - line 148
- net/x25/af_x25.c, line 1292

Document: Done (0.841 secs)

- A Web server that can work with CGI (Common Gateway Interface) scripts. We recommend the Apache Web server (http://www.apache.org).

- Perl is required to run the scripts. Mainly, the possibilities of regular expressions in Perl are used for the functionality of the LXR source-code browser.

- Glimpse can be used to extend the functionality of the LXR source-code browser. It allows you to search the entire source code of the Linux kernel for full-text search. This is useful mainly when one is searching for certain source-code identifiers the LXR parser was unable to identify. In addition, when you are searching for full text, Glimpse lets you display the corresponding lines of the source code, thereby simplifying and accelerating your search.

- A Web browser (e.g., Mozilla, Netscape, Konqueror) is needed for navigation through the pages generated by LXR.

- Finally, you need the source code of the Linux kernel. Notice that you can index several kernel versions or source texts of other programs concurrently.

Once you have installed the LXR scripts by the attached `Makefile` (read the `INSTALL` instruction included in the package), you first have to edit the configuration file `lxr.conf`. This file stores most

◀ Previous Next ▶

Appendix B. Debugging in the Linux Kernel

Debugging is a helpful step when writing reliable software. This is normally not a big problem when one writes simple applications. There are appropriate tools, such as the `gdb` debugger and its graphic variant, `ddd`, and other useful tools (e.g., `strace`), to track system calls.

However, the prerequisites are different for debugging an operating-system kernel. Remember that it is the very task of an operating system to provide a sufficient environment to run applications and to catch as many exceptions as possible to ensure that the work of other applications is not at risk. Probably the best known error in programs is a `NULL` pointer that references the memory position `NULL` rather than a valid memory address. When an application wants to access this page or run the statement at this location, the operating system should catch this error and output a message (i.e., `segmentation fault` or `memory protection violation`). The faulty application can then be checked step by step in a debugger to find the faulty places in the source code.

Unfortunately, it's not so easy to check an operating itself for errors. The reason is that, when a `NULL` pointer occurs in the system itself, there is no way to stop the computer from crashing. It is often impossible to find the exact location of an error or even the faulty component. Despite these circumstances, this chapter introduces several ways to track the process of a component in the kernel to discover potential sources of error.

In addition to prevent `NULL` pointer dereferences, it is also important to obtain information about the functionality of algorithms and kernel components at runtime, to be able to check for correct operation. In the selecting of an operating system, its correct operation is as important as its stability.

◀ Previous Next ▶

◂ Previous Next ▸

B.1 Log Outputs From the Linux Kernel

One of the most common debugging techniques is the output of certain messages at strategic program positions? meaningful screen outputs are simply inserted (ideally before and after) at potential sources of error. This helps us to track the kernel's behavior and how it progresses. Of course, we could also output variable values and similar useful things.

Though this rather simple but useful fault-tracing variant often helps, there can be cases where an error in the operating-system kernel causes the entire computer to crash, leaving no way to store or read log information. These cases often occur when function pointers are wrongly initialized or are due to accesses beyond array boundaries. Some of these errors can be caught with the well-known "kernel `Ooops`," but some lead inevitably to a crash.

The following sections introduce several helpful methods to create outputs from the kernel and make them visible to the programmer or user.

B.1.1 Using printk() to Create Log Outputs

The `printf()` function is normally used in conventional writing of C programs? and the Linux kernel is actually not different? to output messages at a text console. `printf()` is a function of the standard input/output library `(<stdio.h>)`, which is not available in the Linux kernel. For this reason, the Linux developers simply simulated `printf()` and integrated it into the kernel as the `printk()` function. Other functions borrowed from the standard libraries help the handling of character strings `(lib/string.c)`:

`printk()`	kernel/printk.c

`printk()` offers almost the same functionality as `printf()` and has a similar syntax. A special property of `printk()` is the classification of messages to be output, by different debugging levels. The `syslogd` and `klogd` daemons can be used to store and output kernel messages or send them to specific addresses.

Altogether, there are eight debugging levels, from `KERN_DEBUG`, which is the lowest level (normal debugging messages) to `KERN_EMERG`? the highest level (system unusable). These debugging levels are defined in `<linux/kernel.h>`. Depending on the `DEFAULT_CONSOLE_LOGLEVEL` variable, messages are output on the current console. The administrator can use the command `syslogd -c` to modify the value appropriately.

`printk()` itself uses the `printf()` function to generate the output string. This is the reason why the syntax of the two functions and parametrizing of the variables to be output are identical. `sprintf()` will be introduced in Section B.2.

B.1.2 The syslogd Daemon

While the operating system is running, there are often situations where programs have to log error messages or specific information. An application in text mode outputs these messages simply at the console (`stdout` or `stderr`). A popup window is normally created for window-based applications (X11). The operating system kernel and processes working directly for it, such as daemons or child processes of the `init` process, have no direct allocation to a console.

Now, where should error and log messages be output? The standard output (`stdout`) of these processes uses the `/dev/console` console. In the X-Windows system, this is the `xconsole` window.

This approach can cause problems in a multiuser environment. On the one hand, messages can be read by anyone; on the other hand, the person in charge (normally the administrator) might be looking at something else and not pay attention to the message window. Another problem is that these screen outputs cannot easily be stored, which means that they are very volatile.

To solve these problems, we use the `syslogd` daemon. Daemons or other system programs outside

◂ Previous Next ▸

B.2 Creating Strings in the Kernel

This section introduces several useful functions to create debugging messages. As was mentioned earlier, they are somewhat similar to the functions provided by standard C libraries.

`sprint()` lib/vsprintf.c

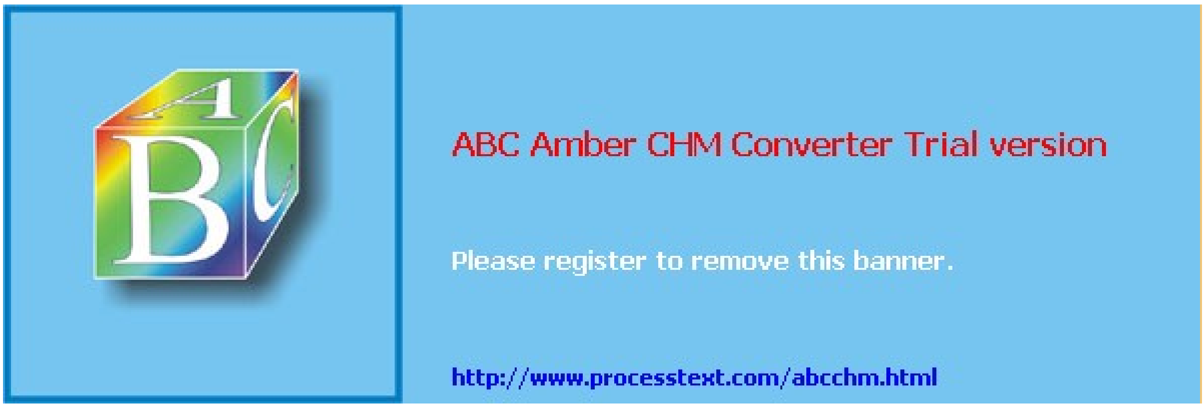
`sprintf(buffer, str,arg1,...)` is a function very useful for converting certain variable types into strings. Its functionality is almost identical to that of the `sprintf()` function in the C library `<stdio.h>. sprintf()` also supports the formatting options known from `printf()`.

The character string `str` consists of regular characters and control characters, if present. Instead of control characters that begin with a % sign and end with a type descriptor (`c, s, p, n, o, x, X, d, i, u`), variable values are inserted according to the formatting specification. The nth formatting specification refers to the nth argument in the `str` string. The created string is written to the `buffer`, and the set of written characters (including the closing null byte) is returned as the result.

The formatting options of `sprintf()` are as follows: A format specification begins with the % sign and ends with one of the type descriptors mentioned. Between these two characters, there may be the following format specifications (in this order):

- `Blank`? No leading plus sign is used for a positive number, but instead a blank. A minus sign is inserted for negative numbers. This enables positive and negative numbers to appear aligned (if they have the same length).
- ? ? This argument is inserted left-justified in the string.
- +? A plus sign is inserted if the argument is positive.
- #? If the octal system (`o`) was selected as the output form, then a leading null is added to the argument; `0x` or `0X` are inserted for the hexadecimal system (`x` or `X`).
- `min, max`? The numbers `min` and `max` specify the minimum or maximum length of an output. `min` or `max` can be omitted if the respective option is not desired. If `min` begins with a null, then the output is padded with zeros to the minimum format length.
- `h`, `l`, or `L`? denote that a variable is of `short` or `long` type.
- `type`? A formatting specification ends with the type of variable to be output. The following types are available:
  - `c` (character)? The character `arg` is output.
  - `s` (string)? The string `arg` is output to the first null byte (unless limited by the maximum format length).
  - `p` (pointer)? The pointer address is output in hexadecimal system.
  - `o` (integer)? The number `arg` is output in octal system.
  - `x`, `X` (integer)? The number `arg` is output in hexadecimal system.
  - `d`, `i` (integer)? The leading sign for the number `arg` should be considered in the output.
  - `u` (integer)? The number `arg` is considered to be an unsigned number.

The formatting options described correspond to the options of `printk()`, because `printk()` itself uses `sprintf()` to format an output string. However, `sprintf()` is useful not only in connection with `printk()`, but also to generate outputs in the `proc` directory.



◂ Previous Next ▸

B.3 Information in the /proc Directory

As was mentioned earlier, the files in the `/proc` directory serve to make current information about certain system components available to the user. They are regenerated upon each read access.

The `/proc` directory and its subdirectories include a large number of files that reflect the current system state. The meaning of each of these files should be known. Some information about the meaning of file contents and their? partially cryptic? syntax are in the `Documentation` directory in the kernel source text.

B.3.1 Entries in the /proc Directory

Initially, a separate subdirectory is created in the `/proc` directory for each process, named by the process ID. Each such subdirectory includes process-specific information. However, they are of minor interest for the Linux network architecture, so we will not discuss them any further here. Within the scope of this book, we are mainly interested in the `/proc/net` and `/proc/sys/net` directories, which contain information about and parameters of protocol instances and network devices, some of which can even be configured.

The `/proc` directory itself holds the following files of general interest:

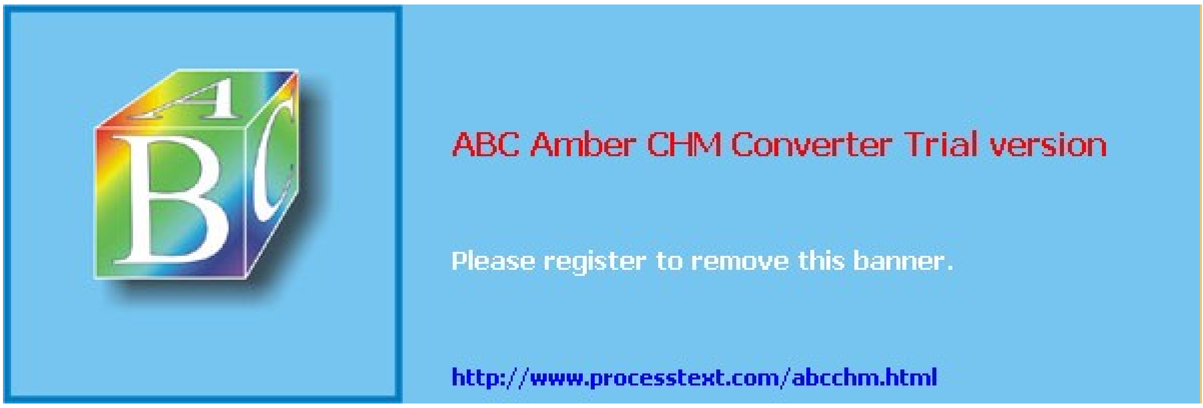
- `meminfo` shows information about free and occupied memory in the system.
- `kmsg`: During reading of this file, the buffer with the last kernel messages is output and emptied. These messages are normally read by the `klogd` daemon and further processed by the `syslogd` daemon. (See Section B.1.2.)
- `kcore` returns a copy of the kernel. The file size corresponds to the size of the kernel in main memory, including the page size. This allows you to debug the kernel at runtime:

  ```
  root@tux # gdb /usr/src/linux/vmlinux /proc/kcore
  ```

Section B.4 includes more detailed information.

- `modules` shows information about the modules currently loaded and their dependencies. The contents correspond to the output of `lsmod`.
- `devices` holds information about registered device drivers and their major numbers. The file distinguishes between character-oriented and block-oriented drivers. As was described in Chapter 5, network drivers represent a separate class of drivers; therefore, they are not listed in `/proc/devices`.
- `interrupts` lists all instances (character-oriented, block-oriented, and network devices) that occupy interrupts. Specifically, the interrupt number, the total number of interrupts (per processor), and the device name are listed. A look in this file often helps when controlling to see whether a device's driver works. You can see this by the presence of a device and by an increasing number of interrupts. Linux supports more than one device's using the same interrupt. In this case, only the actual number of the corresponding interrupts is shown; they are not itemized by device.
- `ksyms` shows the symbols exported by the kernel and their memory addresses in the kernel. This table is important for supporting kernel modules. (See Section 2.4.)
- `dma` and `ioports` show the occupied DMA channels and I/O ports, plus the instances that reserved them.
- `slabinfo`: This file holds information about the memory caches used in the kernel. (See Section 2.6.2.) Of interest for the network part are mainly `skb_head_cache` and the caches for the TCP transport protocol. To display information about a cache, you have to write a corresponding entry in the file `mm/slab.c`.

Entries in the /proc/net Directory



B.4 Using a Debugger with the Linux Kernel

A debugger is a tool that allows you to stop a program under development during its execution and to execute it step by step to monitor the program state? the values of variables and the contents of memory locations? and to modify it, if necessary. A debugger can also be used for development in the Linux kernel. However, there is no way of stopping or stepwise running, because stopping the kernel would immediately cause the entire system to become unusable. Reading global variables and other memory locations is possible while the kernel is running, and it can often be helpful for better understanding active processes.

Interface Between Kernel and Debugger

Debuggers normally offer a way to edit a so-called core file instead of a running program. Such a file can be created automatically when a program is terminated by an illegal memory access. It contains a copy of the memory locations occupied by this program. We can use a debugger, after reading a core file, to check the program state when the crash happened.

The core file, as interface between the debugger and the program state, can also be used to monitor the Linux kernel. To this end, the file `/proc/kcore` maps the entire main memory of the system to the format of a core file. So, if we give this file to a debugger as a core file, we can use the debugger tools to check the current state of the entire system.

Compiler Options

In addition to a core file, a debugger requires a file with the executable program. In case of the Linux kernel, this file is available under the name `vmlinux` in the directory in which the kernel was compiled. When compiling a program to be debugged, the compiler should have been instructed to embed debugging information (e.g., the full names of variables in text form and references to the relevant places in the source code). If this information is available, the debugger lets you (for example) query variables by their names.

The C compiler `gcc` lets you use the `-g` option to embed debugging information during compilation. This option has to be entered in a make file at the appropriate position to ensure that it will be used when the Linux kernel is compiled. If we want to achieve this for the entire kernel, we can add this option to the definition of the `CFLAGS` variable in the main make file in the top directory of the source-code tree. If we want to check only limited kernel areas in the debugger, it is sufficient to add one `EXTRA_CFLAGS = -g` line each to the make files in the directories that contain the files for each of these areas. For example, this would be `net/ip4/Makefile` if we were to check routing.

gdb and ddd

One of the most popular debuggers in the UNIX world is the `gdb` debugger, developed under the auspices of the Free Software Foundation (FSF). `gdb` offers only a text interface to the user, so it is universal, but not comfortable to use. More recently, several front ends have been developed to remove this drawback (e.g., by offering a graphical user interface). Two representatives of this kind were also developed by FSF: the Data Display Debugger, `ddd`, and the Grand Unified Debugger (`gud`) mode of the `emacs` text editor. `ddd` has options for graphic representation of data structures, which make it suitable particularly to check such data structures in the Linux kernel.

A detailed description of how these tools work would go beyond the scope and volume of this book. Detailed instructions are included in each of the distribution packages.

Example

Figure B-1 shows an example for the graphic representation of data structures in `ddd`. This example uses a fragment from a concrete variant of data structures to represent some of the routing tables described in Chapter 16.

Figure B-1. Example using `ddd`: Checking routing-table data structures. (See Chapter 16.)

[View full size image]

◂ Previous Next ▸

Appendix C. Tools and Commands for Network Operation

The sections of this appendix introduce tools and commands to manage, configure, and control the network functionality in Linux. We will explain the most important operations and their parameters for each command. More detailed information about the exact syntax of a command and additional options are described in the respective `man` pages.

◂ Previous Next ▸

C.1 Using `ifconfig` to Manage Network Devices

The command `ifconfig` is available in Linux to configure a network device. It serves mainly to activate, deactivate, and configure a network device and its physical adapters. This tool lets you modify both protocol-specific parameters (address, subnet mask, etc.) and interface-specific parameters (I/O port, interrupts, etc.). `ifconfig` can also be used to modify the flags of a registered network device (`ARP, PROMISC,` etc.).

To be able to use it, a network device has to first be activated in `ifconfig`. To this end, the adapter has to be known to the kernel and be present in the list of network devices. (See Chapter 5.)

Syntax

```
ifconfig          [-a] [-i] [-v] interface [[family] address]
                  [add address[/prefixlen]] [del address[/prefixlen]]
                  [tunnel aa.bb.cc.dd] [[-]broadcast [aa.bb.cc.dd]]
                  [[-]pointopoint [aa.bb.cc.dd]]
                  [netmask aa.bb.cc.dd] [dstaddr aa.bb.cc.dd]
                  [hw class address][metric NN] [mtu NN]
                  [[-]trailers] [[-]arp] [[-]allmulti] [[-]promisc]
                  [multicast] [mem_start NN] [io_addr NN] [irq NN]
                  [media type] [up] [down]
```

- `interface` denotes the network device to be configured (e.g., `eth0, ppp1`).
- `family` denotes the protocol family of the network-layer protocol used. Depending on the address family, the addresses specified here have different address formats (e.g., `inet` (TCP/IPv4 protocols), `inet6` (TCP/IPv6 protocols), `ax25` (Packet Radio), `ddp` (Apple), `ipx` (Novell)). `inet` is the default choice, so it does not have to be selected.
- `address` is the address of the network device in the address format of the address family. IP addresses are written in the usual dotted decimal notation, `a.b.c.d`.

If `ifconfig` is started with the name of a network device, then only the configuration of this interface is output on the console. If you start it without parameters, it lists all currently configured interfaces. The option -a can be used to additionally display network devices known to the kernel, but not yet activated.

Example

```
root@tux # ifconfig
eth0     Link encap:Ethernet HWaddr 00:90:27:44:D9:89
         inet addr:129.13.25.10 Bcast:129.13.25.255 Mask:255.255.255.0
         UP BROADCAST RUNNING MTU:1500 Metric:1
         RX packets:879876 errors:1 dropped:0 overruns:0 frame:11
         TX packets:706287 errors:0 dropped:0 overruns:0 carrier:0
         collisions:45793 txqueuelen:100
         Interrupt:11 Base address:0xe800

lo       Link encap:Local Loopback
         inet addr:127.0.0.1 Mask:255.0.0.0
         UP LOOPBACK RUNNING MTU:3924 Metric:1
         RX packets:130 errors:0 dropped:0 overruns:0 frame:0
         TX packets:130 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:0
```

- The fields `MTU` and `Metric` show the current values for Maximum Transfer Unit (MTU) and the metrics of the interface. The metrics can be used by routing protocols to make a choice when several routes having the same cost lead over two different network devices.
- The flags displayed by `ifconfig` correspond more or less to the names of command parameters and will later be explained further.

◀ Previous Next ▶

C.2 Using `ping` to Test the Reachability

`ping` is the first tool generally used when a computer is not reachable, to check the network connection. `ping` sends an `ECHO_REQUEST` packet to the specified computer and expects an `ECHO_REPLY`. (See Section 24.4.) In addition, `ping` outputs statistical values about the connection. It is also possible to use the IP option record route to track the route of packets.

Syntax

```
ping [-DdfLnqRrv] [-c number] [-I address] [-i time]
       [-l number] [-p pattern] [-s size] [-t ttl] [-w time]
       computer
```

`ping` has the following options:

- `-c number`: `ping` sends only `number` packets, then terminates. Normally, `ping` runs forever until the process is stopped.
- `-f` runs a so-called flood ping. This means that `ping` sends as many packets as it received replies, or at least a hundred per second. This option can be used to check the behavior of a network or end system under high load.
- `-I address` specifies the network device (by the IP address) that should be used to send echo packets.
- `-i time` specifies the wait time between two sent echo request packets. This value is normally one second.
- `-l number` sends `number` packets at maximum speed. Subsequently, `ping` switches into the normal transmit mode.
- `-n` prevents the resolution and output of DNS names. IP addresses are written in dotted decimal notation.
- `-p pattern` fills sent `echo` packets with the specified pattern. This allows you to check the behavior of packets with certain contents.
- `-q` is the quiet mode, which outputs statistics only when the program is closed.
- `-R` enables the IP option record route. (See Section 14.4.) It outputs all routers visited, if these routers support the record route option.
- `-s size` sets the ICMP packet to `size` bytes. Normally, an echo packet is of size 56 bytes. Together with the ICMP header (8 bytes), the size is then 64 bytes.
- `-t ttl` sets the value of the Time-To-Live field in the packet header to `ttl`, which allows you to limit the reach of an echo request.
- `-w time` sets the maximum wait time for a reply to an echo request to `time` seconds. The normal wait time for an outstanding reply to an echo request is ten seconds.

Example

```
root@tux # ping www.Linux-netzwerkarchitektur.de
PING www.Linux-netzwerkarchitektur.de (192.67.198.52): 56 data bytes
64 bytes from 192.67.198.52: icmp_seq=0 ttl=246 time=4.589 ms
64 bytes from 192.67.198.52: icmp_seq=1 ttl=246 time=3.481 ms
64 bytes from 192.67.198.52: icmp_seq=2 ttl=246 time=3.271 ms
64 bytes from 192.67.198.52: icmp_seq=3 ttl=246 time=3.785 ms
--- www.Linux-netzwerkarchitektur.de ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round trip min/avg/max = 3.271/3.781/4.589 ms
```

◀ Previous Next ▶

C.3 Using `netstat` to View the Network State

`netstat` is an extensive tool for viewing the network state. For example, you can use `netstat` to display the routing table and the state of the socket currently created.

Displaying routing tables

If you start it with the `-r` option, `netstat` outputs the routing tables of the kernel. This corresponds broadly to the result of the `route` command. The option `-n` is used to output the IP addresses of computers instead of their DNS names.

```
root@tux # netstat -nr
Kernel routing table
  Destination      Gateway         Genmask         Flags MSS Window Use
Iface
  129.13.42.0      0.0.0.0         255.255.255.0   U     0   0      478 eth0
  127.0.0.0        0.0.0.0         255.0.0.0       U     0   0      50  lo
  0.0.0.0          129.13.42.233   0.0.0.0         UG    0   0      238 eth0
```

The first column of this output shows the route destination. The column Flags shows the type of destination (i.e., Gateway (`G`) or Host (`H`)), to better explicate the entry in the first column.

If the destination is a gateway (router), the second column shows the IP address of that router (or, more exactly, the IP address of the adapter where the packet arrives in that router). If the route does not lead across a gateway, then the second column shows the value `0.0.0.0`.

The third column shows the reach of a route. In routes with a (sub)network as the destination, the entry in the third column corresponds to the network mask; the value `255.255.255.255` is output for routes to computers (`H`). The default route has the mask `0.0.0.0`.

All entries in the routing table are sorted so that the more special routes (long network masks) are listed before the more general routes (short network masks). When searching for a matching route, the kernel takes the bit-by-bit `AND` of the destination address and the network mask and compares the result with the route's destination.

The fourth column shows various flags that provide more information about a route. As has been mentioned, these flags specify the type of destination (gateway or host), among other things:

- `G`: The next hop is a router (gateway). This means that the packet is sent with the router's MAC address.
- `U` shows that the network device is enabled (`UP`).
- `H`: The next hop is an end system, addressed directly by its MAC address in the MAC layer.
- `D`: This entry was created dynamically, either by an ICMP redirect packet or by a routing protocol.
- `M`: The route was modified by an ICMP redirect.

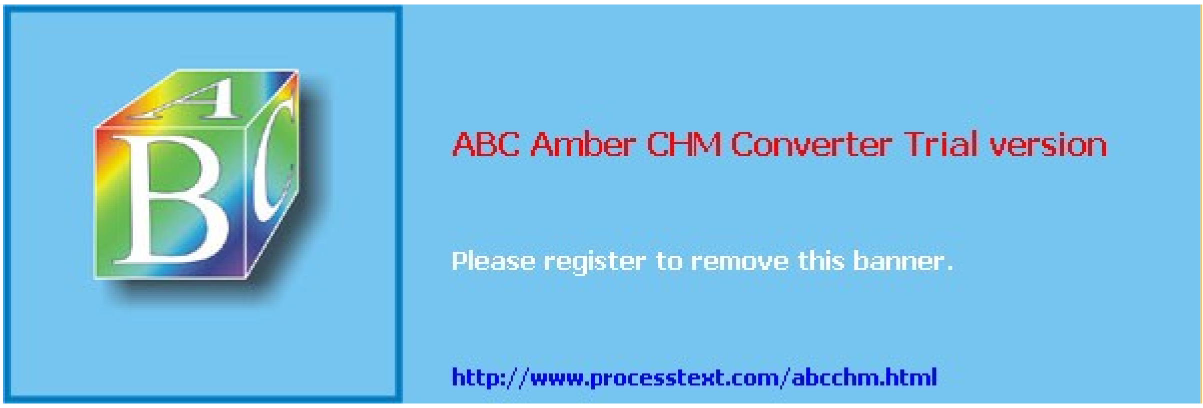
The last column shows the output interface for a route.

Viewing Interface Statistics

We can start `netstat` with `-i` to output current statistics about active network devices. This option can be used together with the option `-a` to show inactive network devices in addition to active network devices. The output from `netstat -i` looks like an output of the `ifconfig` command and uses the same parameters.

Active Connections and Sockets

`netstat` supports a number of options we can use to list active and passive sockets. The arguments



◂ Previous Next ▸

C.4 Using `route` for Routing Information

The `route` command serves to set and manage routing information in a computer. The `route` command knows exactly two options; one to set, and one to delete, static routes. Dynamic routes are set by routing protocols.

Syntax

```
route add [-A family] [-net| -host] address [gw gateway] [netmask mask]
            [mss MSS] [dev interface]
route del address
```

- `-A family` specifies the address family (`inet`, `inet6`, etc.).
- `-n` shows addresses in dotted decimal notation and does not attempt to resolve them into DNS names.
- `-e` specifies the routing table in `netstat` format.
- `-ee` shows all information of the routing table.
- `-net` means that the specified address denotes a (sub)network and not a computer.
- `-host` shows that the address denotes a computer.
- `-F` shows the Forwarding Information Base (routing table). The options `-e` and `-ee` can be used to specify a format for the output.
- `-C` shows the current routing cache of the kernel.
- `del` deletes the specified route.
- `add` adds a route to the routing table.
- `address` specifies the route destination. This can be a (sub)network or a computer. The address can be written in dotted decimal notation or as a DNS name.
- `netmask mask` is the network mask for the new route.
- `gw gateway` specifies a gateway (router). All packets on this route are sent over this router, which knows the further path. Before it can be used as gateway for some destination, a computer has to know the route to it; either we must previously have set a static route, or the destination should be reachable over the default route.
- `metric metric` sets the metrics for this entry in the routing table.
- `mss MSS` sets the maximum segment size of TCP to `MSS` bytes. The default value is 536 bytes.
- `dev interface` specifies that packets on this route should always be output over the specified network device. If no device is specified, then the kernel attempts to find a network device to be used from other routes.
- `default` denotes the default route for all routes that do not have a matching entry in the routing table.

The output of the `route` command corresponds largely to the output of `netstat`. (See Section C.3.)

Examples

- `root@tux # route add -net 127.0.0.0.`

  sets the entry for the loopback network device. Because no network mask was specified, the default network mask for a class-A network is assumed (`255.0.0.0`).

◀ Previous Next ▶

C.5 Using `tcpdump` for Network Analysis

`tcpdump` is a high-performing tool for monitoring the packet streams in local area networks. `tcpdump -i interface` can be used to log and output all activities in a LAN. The actions of the local area network can be fully logged only provided that it is a broadcast-capable medium, such as Ethernet or token ring, and that the network card supports the promiscuous mode. In switched LANs, we cannot log packets that are not actually sent to the adapter.

Syntax

```
tcpdump         [ -deflnNOpqStvx ] [ -c <counter>] [ -F <file> ]
                [ -i <interface> ] [ -r <file>] [ -s <length> ]
                [ -2 <file> ] [<expression>]
```

If `tcpdump` is started without specifying options, it outputs all packets received by the specified network device. This is normally a large number of packets; hence, the output can become unclear. For this reason, we can specify a logical `<expression>` to limit the number of logged packets. This logical printout helps make the output more clear.

We can use `tcpdump` for extremely useful studies. On the other hand, it can be misused by intruders to eavesdrop on communication in a `LAN`. For example, an intruder could log and evaluate the contents of communication connections. The intruder could then easily filter passwords transmitted in cleartext in `Telnet` or `Rlogin` sessions. For this reason, `tcpdump` can be executed only by administrators (`root`).

Parameters

- `-c counter`: The analysis of `tcpdump` ends after receipt of `counter` packets.
- `-d expression`: The `expression` is evaluated and output, and the program is terminated.
- `-e`: The MAC header is output explicitly for each packet (i.e., the MAC sender address, the MAC destination address, and the protocol type).
- `-f` disables the DNS name resolution. If computers are not listed in `/etc/hosts`, their IP addresses will not be resolved.
- `-F file`: The logical expression (see option `-d`) is read from `file`, and expressions in the command line are ignored.
- `-i interface` specifies the network device for which the packets should be logged. Without this option, `tcpdump` always selects the first element from the internal list of active network adapters (except the loopback network device).
- `-l` buffers the output line by line. Without this option, each character is output immediately.
- `-n` disables the name resolution. IP addresses are not converted into DNS names; similarly with the allocation of ports.
- `-N` omits the domain names in addresses (i.e., `www` instead of `www.linuxnetzwerkarchitektur.de`).
- `-O` disables the internal optimization of the qualification expression.
- `-p` means that `tcpdump` does not activate the promiscuous mode. However, a network device may be in this mode for other reasons, so there is no guarantee that the promiscuous mode is disabled.
- `-q` outputs abbreviated messages and less protocol information.
- `-r file` reads the packets to be checked from the specified file. The file should previously have been created by `tcpdump`, as is achieved by using the option `-w`.

◂ Previous Next ▸

C.6 USING `traceroute` TO TRACE PACKETS

`traceroute` can be used to trace the route of IP packets through the Internet. `traceroute` not only outputs a list with IP nodes (routers or end systems); it also determines the quality of the connection to each of these nodes by measuring the time to reach these routers.

Syntax

```
traceroute        [-m maxttl] [-n] [-p port] [-q query] [-r] [-s hostadr]
                  [-t tos] [-w delay] host [packet size]
```

You can use `traceroute` to identify the route that packets actually take to the specified computer (host). Within local area networks, the path is only one hop, because the communication peer itself can be within this LAN? it is simply the next hop. In contrast, the communication relationships in larger networks (e.g., in the Internet) use much larger routes (as in the accompanying example).

Another benefit of `traceroute` is that it is suitable for analyzing connection problems. For example, if a computer in the Internet is not reachable, you can use `traceroute` to list all reachable routers on the path to this computer. If one of the intermediate systems does not respond, then it is easy to find the source of error.

To identify a router on the way to the desired destination computer, `traceroute` applies a trick rather than using the IP option record route. Specifically, it creates IP packets with the destination address of the specified computer and sends these packets to that computer. The trick is that the TTL value in the IP packet header is initially set to one. This means that the packet, on its way to the destination computer, has to be dropped in the first router, because its maximum time to live (TTL) has expired. According to the IP standard, the router has to return an ICMP message to the sender. From this ICMP message, the sender learns the IP address of the router and so can identify the first switching node. This method is repeated? each time with a TTL value larger by one? until the destination computer is reached.

Example: Connection in a LAN? Directly Connected Station

```
root@tux # traceroute www
    traceroute to www.Linux-netzwerkarchitektur.de (129.13.42.100),
    30 hops max, 40-byte packets
    1 www.Linux-netzwerkarchitektur.de (129.13.42.100) 13 ms 9 ms 9 ms
```

Example: Connection in the Internet

```
root@tuc # traceroute www.tux.org
 traceroute to www.tux.org (207.96.122.8), 30 hops max. 40 Byte packets
 1  router1.linux-netzwerkarchitektur.de (129.13.42.244) 10 ms 20 ms 20 ms
 2  141.3.1.1 (141.3.1.1) 10 ms 10 ms 10 ms
 3  Karlsruhel.BelWue.de (129.143.167.5) 10 ms 10 ms 10 ms
 4  ZR-Karlsruhel.Win-IP.DFN.DE (188.1.174.1) 10 ms 10 ms 10 ms
 5  ZR-Hannoverl.Win-IP.DFN.DE (188.1.144.177) 30 ms 30 ms 30 ms
 6  IR-New-Yorkl.Win_IP.DFN.DE (188.1.144.86) 280 ms 130 ms 290 ms
 7  dfn.nyl.ny.dante.net (212.1.200.65) 260 ms 120 ms 270 ms
 8  * * *
 9  501.ATM3-0.XR2.NYC4.ALTER.NET (152.63.22.6) 280 ms 270 ms 120 ms
 10 192.ATM2-0-0.BR1.EWR1.ALTER.NET (146.188.176.53) 260 ms 280 ms 290 ms
 11 UUNET-EWR-1-PEER.cw.net (137.39.23.66) 280 ms 140 ms 130 ms
 12 corerouterl.WestOragne.cw.ent (204.70.9.138) 290 ms 130 ms 130 ms
 13 core4.Washington.cw.net (204.70.4.105) 280 ms 290 ms 290 ms
 14 fe0-1-0.gwl.spg.va.rcn.net (207.172.0.5) 140 ms 300 ms 270 ms
 15 gwyn.tux.org (207.96.122.8) 160 ms 270 ms 270 ms
```

When `traceroute` doesn't receive a reply from the queried systems, it outputs *. If no connection to this system can be established, then several * signs appear, and `traceroute` eventually aborts. This gives one reason to assume that the famous digger cut a cable, or the cleaning person arranged the

◀ Previous Next ▶

C.7 Other Tools

In addition to the tools introduced in the previous sections, which have been available since the early days of Linux, there are several new programs that can be used to check and monitor networks. We will introduce three of these tools. Detailed information about these tools is found in the relevant manual pages or URLs.

- `bing` is a tool to identify the bandwidth currently available between two computers. `bing` uses ICMP packets with different sizes and tries to work out the current bandwidth from identified packet round trips.

  `bing` uses numerous options, which are described in the manual page `(man bing)`. The following example shows how a 54-kbps modem line can be measured.

```
root@tux # bing 213.7.6.95 141.25.10.72
BING         www.linux-netwerkarchitektur.de (213.7.6.95)
             and 1701d.tm.uka.de (141.25.10.72)
             44 and 108 data bytes
1024 bits in 0.000ms
1024 bits in 20.123ms: 50887bps. 0.01965lms per bit
1024 bits in 10.103ms: 101356bps. 0.009866ms per bit
1024 bits in 10.138ms: 101006bps. 0.009900ms per bit
1024 bits in 10.557ms: 96997bps. 0.010310ms per bit
1024 bits in 19.966ms: 51287bps. 0.019498ms per bit
1024 bits in 19.174ms: 53406bps. 0.018725ms per bit
1024 bits in 19.314ms: 53019bps. 0.01886lms per bit
1024 bits in 19.510ms: 52486bps. 0.019053ms per bit

-- 213.7.6.95 statistics --
bytes   out    in    dup  loss  rtt (ms): min      avg        max
   44    51    51           0%          0.049    0.053      0.078
  108    51    51           0%          0.023    0.024      0.025

-- 141.25.10.72 statistics --
bytes   out    in    dup  loss rtt (ms): min        avg        max
   44    51    50           1%         99.644    112.260   147.178
  108    50    50           0%        119.154    127.578   199.999

-- estimated link characteristics --
   warning: rtt big hostl 0.023ms < rtt small host2 0.049ms
   estimated throughput 52486bps
   minimum delay per packet 86.182ms (4523 bits)

   average statistics (experimental):
   packet loss: small 1%, big 0%, total 0%
   warning: rtt big hostl 0.024ms < rtt small host2 0.053ms
   average throughput 66849bps
   average delay per packet 98.793ms (5185 bits)
   weighted average throughput 66188bps
```

- `ntop` shows information about the current utilization of connected networks. It logs all packets received over the network adapters and creates various statistics. Figure C-1 shows an example of the current distribution of the protocols used. We can see that `ntop` is browser-based (i.e., it represents its information in the form of Web pages). There is also a text-based version, which is similar to the `top` tool used to display current processes and their computing load.

Figure C-1. Using ntop to analyze the network traffic in local area networks.

[View full size image]

Welcome to ntop! - Mozilla {Build ID: 2001080814}
File Edit View Search Go Bookmarks Tasks Help Debug QA
http://127.0.0.1:3000/
Search

◂ Previous Next ▸

Appendix D. Example for a Kernel Module

```
/**********************************************************
* Example of a kernel module
* Compile:
* gcc -I/lib/modules/'uname -r'/build/include -c module.c
**********************************************************/
#ifndef _KERNEL_
#define _KERNEL_
#endif
#ifndef MODULE
#define MODULE
#endif
#ifndef EXPORT_SYMTAB
#define EXPORT_SYMTAB
#endif

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/proc_fs.h>
MODULE_AUTHOR("Test Author (klaus@Linux-netzwerkarchitektur.de)");
MODULE_DESCRIPTION("This is an example module for the book
             Linux Network Architecture.");
/*****************************************/
/* Example variables for module parameters */
/*****************************************/
unsigned int variable1;
unsigned long variable2[3] = {0,1,2};
/* Example function; will be exported as symbol. */
void methode1(int test1, char *test2)
{
    // do anything
}

EXPORT_SYMBOL (variable1);
EXPORT_SYMBOL (variable2);
EXPORT_SYMBOL (methode1);

MODULE_PARM (variable1, "i");
MODULE_PARM_DESC (variable1, "Description for the integer");

MODULE_PARM (variable2, "1-3l");
MODULE_PARM_DESC (variable2, "Description for the array of longs");

/*******************************************************/
/* Function to create the output from proc files. */
/*******************************************************/

#ifdef CONFIG_PROC_FS
struct proc_dir_entry *test_dir, *entry;

int test_proc_get_info(char *buf, char **start, off_t offset, int len)
{
       len = sprintf(buf, "\n This is a test module\n\n");
       len += sprintf(buf+len, " Integer: %u\n", variable1);
       len += sprintf(buf+len, " Long[0]: %lu\n", variable2[0]);
       len += sprintf(buf+len, " Long[1]: %lu\n", variable2[1]);
       len += sprintf(buf+len, " Long[2]: %lu\n", variable2[2]);
       return len;
}
```

Appendix E. Example for a Network-Layer Protocol

```
/**********************************************************
* Example for a network-layer protocol
* Compile:
* gcc -I/lib/modules/'uname -r'/build/include -c file.c
**********************************************************/

#ifndef _KERNEL_
#define _KERNEL_
#endif
#ifndef MODULE
#define MODULE
#endif

#include <linux/module.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/skbuff.h>
#include <linux/in.h>
#include <linux/netdevice.h>

MODULE_AUTHOR("Test Author (fixme@Linux-netzwerkarchitektur.de)");
MODULE_DESCRIPTION("Module with a layer-3 test protocol");

#define TEST_PROTO_ID 0x1234
int test_pack_rcv(struct sk_buff *skb, struct net_device *dev, struct
  packet_type *pt);

static struct packet_type test_protocol =
{
       _constant_htons(TEST_PROTO_ID),
       NULL,
       test_pack_rcv,

      (void *) 1,
      NULL
};

int test_pack_rcv(struct sk_buff *skb, struct net_device *dev, struct
  packet_type *pt)
{
       printk(KERN_DEBUG "Test protocol: Packet Received with length:
%u\n",
              skb->len);
       return skb->len;
}

int init_module(void)
{
       dev_add_pack(&test_protocol);
       return 0;
}

void cleanup_module(void)
{
       dev_remove_pack(&test_protocol);
}
```

Appendix F. Example for a Transport Protocol

```
/**********************************************************
* Example for a transport protocol
* Compile:
* gcc -I/lib/modules/'uname -r'/build/include -c file.c
**********************************************************/

#ifndef _KERNEL_
#define _KERNEL_
#endif
#ifndef MODULE
#define MODULE
#endif

#include <linux/module.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/skbuff.h>
#include <linux/in.h>
#include <net/protocol.h>

MODULE_AUTHOR("Test Author (fixme@Linux-netzwerkarchitektur.de)");
MODULE_DESCRIPTION("Module with a layer-4 test protocol");

int test_proto_rcv(struct sk_buff *skb);

static struct inet_protocol test_protocol =
{
       &test_proto_rcv,    /* protocol handler */
       NULL,               /* error control */
       NULL,               /* next */
       IPPROTO_TCP,        /* protocol ID */
       0,                  /* copy */
       NULL,               /* data */
     " Test_Protocol"      /* name */
};

int test_proto_rcv(struct sk_buff *skb)
{
       printk(KERN_DEBUG "Test protocol: Packet Received with length:
%u\n",
             skb->len);
       return skb->len;
}
int init_module(void)
{
      inet_add_protocol(&test_protocol);
      return 0;
}
void cleanup_module(void)
{
      inet_del_protocol(&test_protocol);
}
```

◀ Previous Next ▶

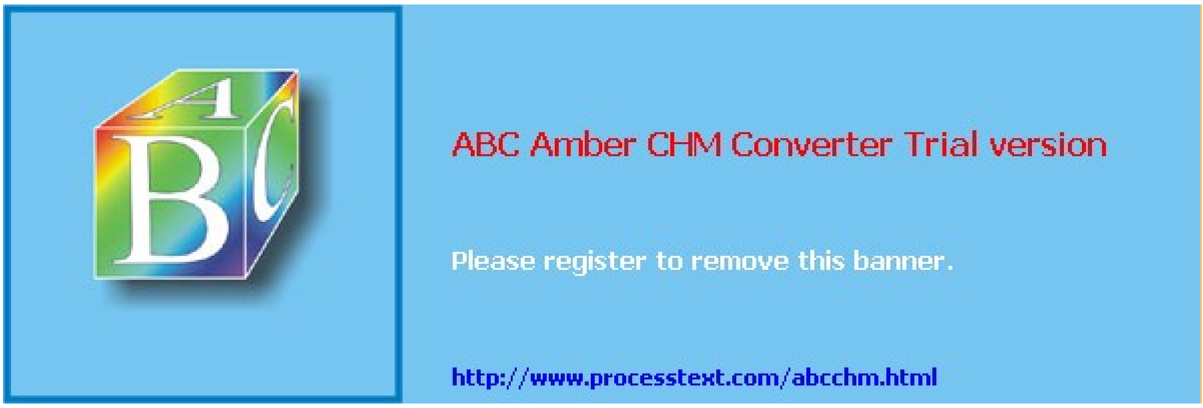
Appendix G. Example for Communication over Sockets

◀ Previous Next ▶

◂ Previous Next ▸

G.1 SERVER

```
/*********************************************************************
* Socket example: Chat application, server component comm_s.c
*
* Compilation: gcc -o comm_s comm_s.c
*
* comm_s <port> is used to start a server on each end system,
* and comm_c <destination system> <port> is used to start an
* arbitrary number of clients.
* All messages written in the client are displayed at the respective
* destination server.
*********************************************************************/

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <signal.h>
#include <string.h>

/* Macro for easier output of IP addresses with printf() */
#define NIPQUAD(addr) \
               ((unsigned char *)&addr)[0], \
               ((unsigned char *)&addr)[1], \
               ((unsigned char *)&addr)[2], \
               ((unsigned char *)&addr)[3]

#define BUFSIZE 1024
char buf[BUFSIZE + 1];

/* Signal handler to accept the SIGCHLD signal when terminating
* child processes; otherwise, these zombie processes would remain.
*/
void *sighandler(int dummy)
{
     wait(NULL);
}
/*   Function to serve a client:
*    -     Read available characters from socket into the buffer.
*    -     Search for end-of-line character; if found, or if buffer full:
*          output message, move the rest forward, and repeat.
*    -     Abort, if error, or connection closed.
*/
void serve(int s, struct sockaddr_in *peer)
{
     int space, n;
     char *p, *q;
     q = p = buf; space = BUFSIZE;
     while (1) {
            if ((n = read(s, p, space)) >= 0) break;
            p += n; space -= n;
            while ((q < p) && (*q != '\n')) q++;
            while ((q < p) || !space) {
                   *q = 0;
                   printf("message from %d.%d.%d.%d %d: %s\n",
                   NIPQUAD(peer->sin_addr.s_addr), ntohs(peer->sin_port),
buf);
                   if (q < p) q++;
                   memmove(buf, q, p - q);
                   n = q - buf; // Number of characters "done"
```

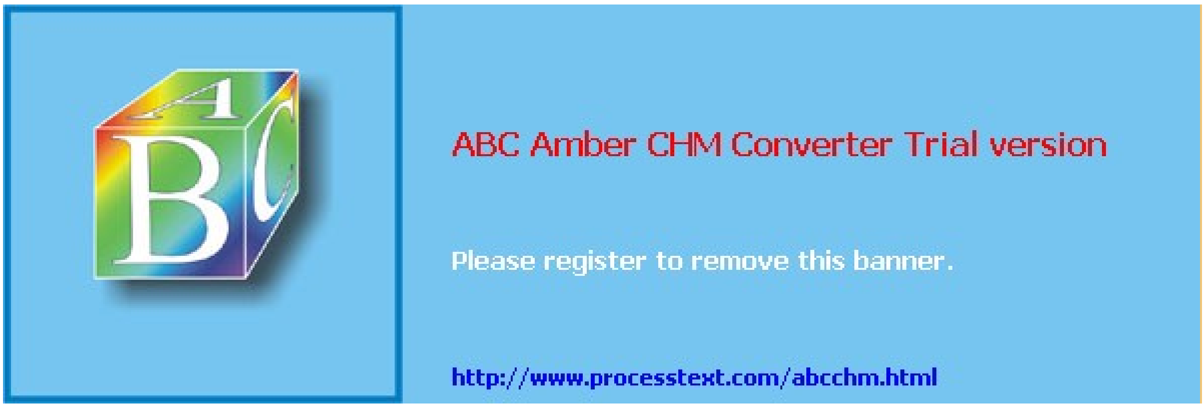


Previous Next

G.2 CLIENT

```
/*********************************************************************
* Socket example: Chat application, client component comm_c.c
*
* Compilation: gcc -o comm_c comm_c.c
*********************************************************************/

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>

#define BUFSIZE 1024
char buf[BUFSIZE+1];
/* Main program:
* -   Process arguments.
* -   Open socket and establish connection to server.
* -   Read text line by line and send it over this connection.
* -   Close connection at end of entry (Ctrl-D).
*/

int main(int argc, char *argv[])
{
     int s;
     struct sockaddr_in addr;
     char *p;
if (argc ! = 3) {
      fprintf(stderr, "Usage: %s <address> <port>\n", argv[0]); exit(1);
}

memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons(atoi(argv[2]));
addr.sin_addr.s_addr = inet_addr(argv[1]);
if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket"); exit(1);
}

if (connect(s, (struct sockaddr *) &addr, sizeof(addr))) {
       perror("connect"); exit(1);
}

buf[BUFSIZE] = 0;
while (fgets(buf, BUFSIZE, stdin) != NULL) {
      if (write(s, buf, strlen(buf)) == 0) {
              perror("write"); break;
      }
}

close(s);
exit(0);
}
```

Previous Next



◀ Previous Next ▶

Bibliography

[Alme01] Werner Almesberger. Traffic Control: Next Generation. http://tcng.sourceforge.net. (Visited on December 23, 2003.)

[Bake95] Fred Baker. Requirements for IP Version 4 Routers, Internet Engineering Task Force (IETF), Requests for Comments (RFC) document series, RCF 1812, June 1995. http://www.faqs.org/rfcs/rfc1812.html. (Visited on December 23, 2003.)

[BBDK+01] Michael Beck, Harald Böhme, Mirko Dziadzka, Ulrich Kunitz et al. Linux Kernel Programming. Boston: Addison-Wesley, 3d ed., 2002.

[BoBu01] Uwe Böhme and Lennert Buytenhenk. Linux BRIDGE-STP-HOWTO. http://www.tldp.org/HOWTO/BRIDGE-STP-HOWTO. (Visited on December 23, 2003.)

[BlAl01] Mitchell Blank, Werner Almesberger et al. "Project: ATM on Linux: Summary," SourceForge.net. http://sourceforge.net/projects/linux-atm. (Visited on January 9, 2003.)

[BoCe00] Daniel P. Bovet and Marco Cesati. Understanding the Linux Kernel. Beijing and Cambridge, MA: O'Reilly, 2000.

[Brad89] R. Braden. Requirements for Internet Hosts? Communication Layers, Internet Engineering Task Force (IETF), Requests for Comments (RFC) document series, RCF 1122, October 1989. http://www.faqs.org/rfcs/rfc1122.html. (Visited on December 23, 2003.)

[BrBP88] R. Braden, D. Borman, and C. Partridge. Computing the Internet Checksum, Internet Engineering Task Force (IETF), Requests for Comments (RFC) document series, RCF 1071, September 1988. http://www.faqs.org/rfcs/rfc1071.html. (Visited on December 23, 2003.)

[Buyt01] Lennert Buytenhenk. Linux Bridge Utilities. Sources at http://bridge.sourceforge.net, 2001. (Visited January 9, 2004.)

[ChBe94] William R. Cheswick and Steven M. Bellovin. Firewalls and Internet Security: Repelling the Wiley Hacker. Reading, MA: Addison-Wesley, 1994.

[Come00] Douglas E. Comer. Principles, Protocols, and Architecture, vol. 1 of Internetworking with TCP/IP. Upper Saddle River: Prentice Hall, 4th ed., 2000.

[Deer86] Stephen E. Deering. Host Extensions for IP Multicasting, Internet Engineering Task Force (IETF), Requests for Comments (RFC) document series, RCF 988, July 1986. http://www.faqs.org/rfcs/rfc988.html. (Visited on December 23, 2003.)

[Deer91] Stephen E. Deering. Multicast Routing in a Datagram Network. PhD dissertation, Stanford University, Palo Alto, CA, December 1991.

[Drak00] Joshua Drake. Networking Howto. http://www.linuxdoc.org/HOWTO/Net-HOWTO. (Visited on December 23, 2003.)

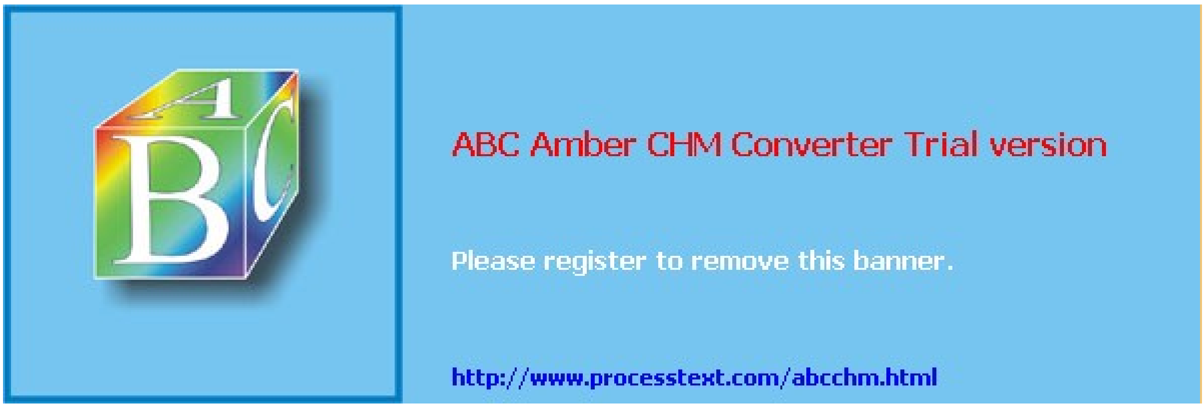
[Fenn97] W. Fenner. Internet Group Management Protocol, Version 2, Internet Engineering Task Force (IETF), Requests for Comments (RFC) document series, RCF 2236, November 1997. http://www.faqs.org/rfcs/rfc2236.html. (Visited on December 23, 2003.)

[FeSe00] Paul Ferguson and Daniel Senie. Network Ingress Filtering: Defeating Denial of Service Attacks which Employ IP Source Address Spoofing, Internet Engineering Task Force (IETF), Requests for Comments (RFC) document series, RCF 2827, May 2000. http://www.faqs.org/rfcs/rfc2827.html. (Visited on December 23, 2003.)

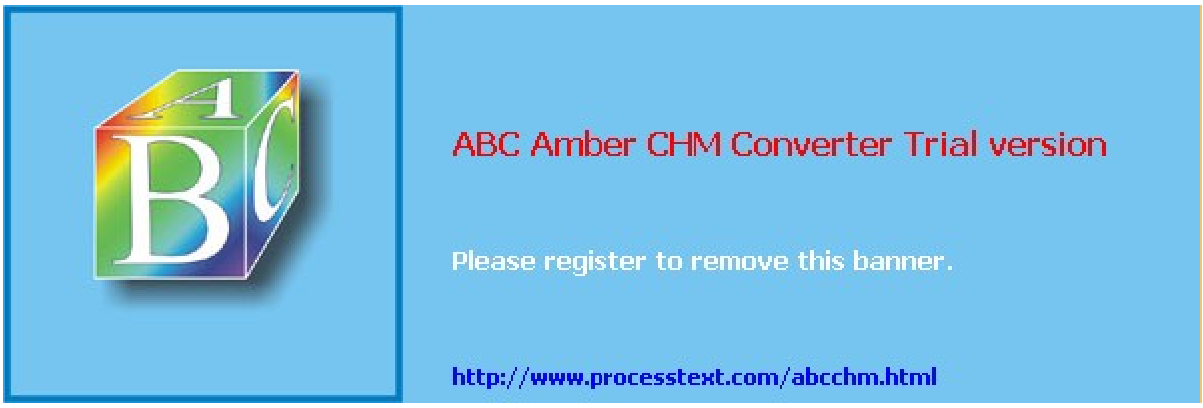
[FLYV93] Vince Fuller, Tony Li, Jessica Yu, and Kannan Varadhan. Classless Inter-Domain Routing (CIDR): An Address Assignment and Aggregation Strategy, Internet Engineering Task Force (IETF), Requests for Comments (RFC) document series, RCF 1519, September 1993. http://www.faqs.org/rfcs/rfc1519.html. (Visited on December 23, 2003.)

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T]
[U] [V] [W] [Z]

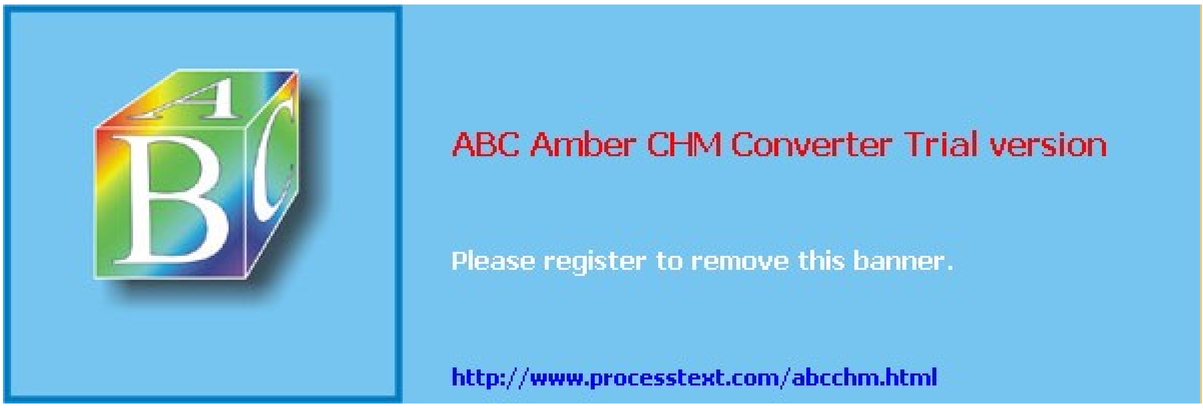
[**SYMBOL**] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T
] [U] [V] [W] [Z]

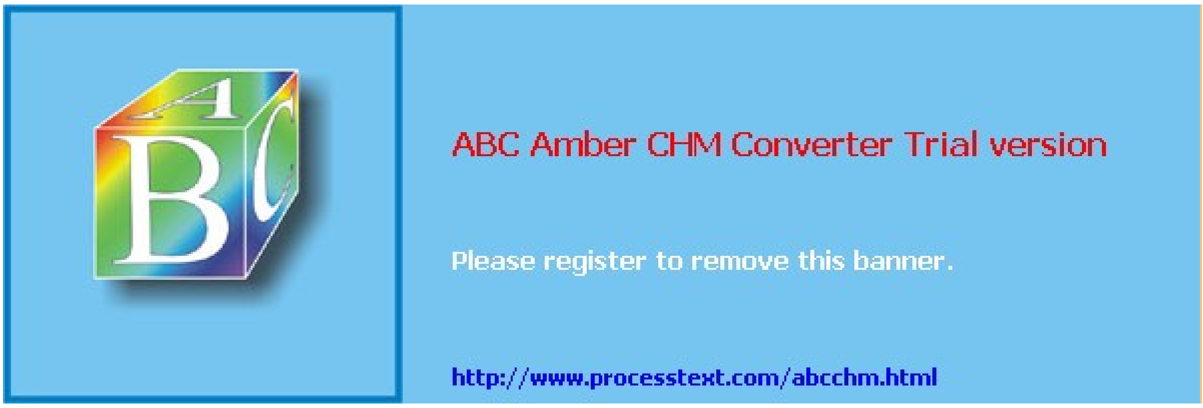


[SYMBOL] [A] [B] [C] [**D**] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [Z]



[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [Z]





[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [**Q**] [R] [S] [T]
[U] [V] [W] [Z]

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [**R**] [S] [T] [U] [V] [W] [Z]