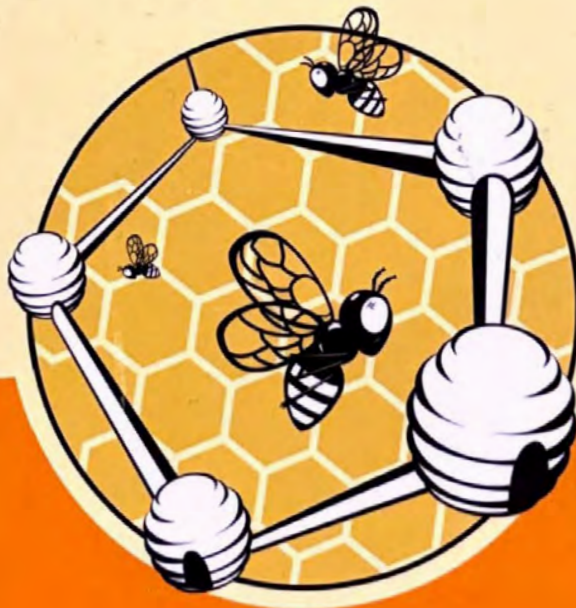


The

TCP/IP GUIDE

**A COMPREHENSIVE, ILLUSTRATED
INTERNET PROTOCOLS REFERENCE**



Charles M. Kozierok



**NO STARCH
PRESS**

THE TCP/IP GUIDE

**A Comprehensive, Illustrated
Internet Protocol Reference**

by Charles M. Kozierok



**NO STARCH
PRESS**

San Francisco

THE TCP/IP GUIDE. Copyright © 2005 by Charles M. Kozierok.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.



Printed on recycled paper in the United States of America

1 2 3 4 5 6 7 8 9 10 - 08 07 06 05

No Starch Press and the No Starch Press logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Publisher: William Pollock

Production Manager: Susan Berge

Cover and Interior Design: Octopod Studios

Developmental Editor: William Pollock

Technical Reviewers: Fernando Gont, Barry Margolin

Copyeditors: Marilyn Smith, Lisa Theobald, Mark Nigara

Compositor: Riley Hoffman

Proofreader: Stephanie Provines

For information on book distributors or translations, please contact No Starch Press, Inc. directly:

No Starch Press, Inc.

555 De Haro Street, Suite 250, San Francisco, CA 94107

phone: 415.863.9900; fax: 415.863.9950; info@nostarch.com; <http://www.nostarch.com>

The information in this book is distributed on an "As Is" basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

Library of Congress Cataloging-in-Publication Data

Kozierok, Charles.

The TCP/IP guide : a comprehensive, illustrated internet protocols reference / Charles Kozierok.
p. cm.

Includes index.

ISBN 1-59327-047-X

1. TCP/IP (Computer network protocol) I. Title.

TK5105.585.K69 2005

004.6'2--dc22

2004008863

To comprehend the relationship between connections and circuits, you must recall the layered nature of modern networking architecture (as I discuss in some detail in Chapter 5). Even though packets may be used at lower layers for the mechanics of sending data, a higher-layer protocol can create logical connections through the use of messages sent in those packets.

TCP/IP has two main protocols that operate at the transport layer of the OSI Reference Model. One is TCP, which is connection-oriented; the other, the User Datagram Protocol (UDP), is connectionless. TCP is used for applications that require the establishment of connections (as well as TCP's other service features), such as FTP; it works using a set of rules, as described earlier, by which a logical connection is negotiated prior to sending data. UDP is used by other applications that don't need connections or other features, but do need the faster performance that UDP can offer by not needing to make such connections before sending data.

Some people consider the layering of a connection-oriented protocol over a connectionless protocol to be like a simulation of circuit switching at higher network layers; this is perhaps a dubious analogy. Even though you can use a TCP connection to send data back and forth between devices, all that data is indeed still being sent as packets; there is no real circuit between the devices. This means that TCP must deal with all the potential pitfalls of packet-switched communication, such as the potential for data loss or receipt of data pieces in the incorrect order. Certainly, the existence of connection-oriented protocols like TCP doesn't obviate the need for circuit-switching technologies, though you will get some arguments about that one too.

The principle of layering also means that there are other ways that connection-oriented and connectionless protocols can be combined at different levels of an internetwork. Just as a connection-oriented protocol can be implemented over an inherently connectionless protocol, the reverse is also true: a connectionless protocol can be implemented over a connection-oriented protocol at a lower level. In a preceding example, I talked about Telnet (which requires a connection) running over IP (which is connectionless). In turn, IP can run over a connection-oriented protocol like Asynchronous Transfer Mode (ATM).

Messages: Packets, Frames, Datagrams, and Cells

Many networking technologies are based on packet switching, which involves the creation of small chunks of data to be sent over a network. Even though *packet* appears in the name of this method, the data items sent between networked devices are most generically called *messages*. *Packet* is one of a variety of similar words that are used in different contexts to refer to messages sent from one device to another.

In some cases, the different terms can be very useful, because the name used to refer to a particular message can tell you something about what the message contains, as you will see shortly. In particular, different message names are usually associated with protocols and technologies operating at specific layers of the OSI Reference Model. Thus, the use of these different names can help clarify discussions that involve multiple protocols operating at different layers.

Unfortunately, these terms can also cause confusion, because they are not always applied in a universal or even consistent manner. Some people are strict about applying particular message designations only to the appropriate technologies

where they are normally used, while others use the different terms completely interchangeably. This means that you should be familiar with the different message types and how they are normally used, but you should still be prepared for the unexpected.

The most common terms used for messages are the following:

Packet This term is considered by many to correctly refer to a message sent by protocols operating at the network layer of the OSI Reference Model. So you will commonly see people refer to *IP packets*. However, this term is commonly also used to refer generically to any type of message, as I mentioned earlier.

Datagram This term is basically synonymous with *packet* and is also used to refer to network layer technologies. It is also often used to refer to a message that is sent at a higher level of the OSI Reference Model (more often than *packet* is).

Frame This term is most commonly associated with messages that travel at low levels of the OSI Reference Model. In particular, it is most commonly seen used in reference to data link layer messages. It is occasionally also used to refer to physical layer messages, when message formatting is performed by a layer 1 technology. A frame gets its name from the fact that it is created by taking higher-level packets or datagrams and “framing” them with additional header information needed at the lower level.

Cell Frames and packets, in general, can be of variable length, depending on their contents; in contrast, a *cell* is most often a message that is fixed in size. For example, the fixed-length, 53-byte messages sent in ATM are called cells. Like frames, cells are usually used by technologies operating at the lower layers of the OSI model.

Protocol Data Unit (PDU) and Service Data Unit (SDU) These are the formal terms used in the OSI Reference Model to describe protocol messages. A PDU at layer N is a message sent between protocols at layer N. It consists of layer N header information and an encapsulated message from layer N+1, which is called both the *layer N SDU* and the *layer N+1 PDU*. After you stop scratching your head, see the “Data Encapsulation, Protocol Data Units (PDUs), and Service Data Units (SDUs)” section in Chapter 5 for a discussion of this.

I should also point out that there are certain protocols that use unusual names, which aren’t used elsewhere in the world of networking, to refer to their messages. One prominent example is TCP, which calls its messages *segments*.

KEY CONCEPT Communication between devices on packet-switched networks is based on items most generically called messages. These pieces of information also go by other names such as packets, datagrams, frames, and cells, which often correspond to protocols at particular layers of the OSI Reference Model. The formal OSI terms for messages are *protocol data unit (PDU)* and *service data unit (SDU)*.

In this book, I have made a specific effort not to imply anything about the nature of a message solely based on the name it uses, but I do follow the most common name used for a particular technology. For example, messages sent over Ethernet

are almost always called Ethernet frames—they are not generally called Ethernet datagrams, for example. However, I do not structure discussions so that the type of name used for a message is the only way to determine what sort of message it is.

Message Formatting: Headers, Payloads, and Footers

Messages are the structures used to send information over networks. They vary greatly from one protocol or technology to the next in how they are used, and as just described, they are also called by many different names. Shakespeare had the right idea about names, however. The most important way that messages differ is not in what they are called but in terms of their *content*.

Every protocol uses a special *formatting method* that determines the structure of the messages it employs. Obviously, a message that is intended to connect a web server and a web browser is going to be quite different from one that connects two Ethernet cards at a low level. This is why I separately describe the formats of dozens of different protocol messages in various areas of this book.

While the format of a particular message type depends entirely on the nature of the technology that uses it, messages on the whole tend to follow a fairly uniform overall structure. In generic terms, each message contains the following three elements (see Figure 1-3):

Header Information that is placed before the actual data. The header normally contains a small number of control-information bytes, which are used to communicate important facts about the data that the message contains and how it is to be interpreted and used. It serves as the communication and control link between protocol elements on different devices.

Data The actual data to be transmitted, often called the *payload* of the message (metaphorically borrowing a term from the space industry!). Most messages contain some data of one form or another, but some messages actually contain none. They are used for only control and communication purposes. For example, these may be used to set up or terminate a logical connection before data is sent.

Footer Information that is placed after the data. There is no real difference between the header and the footer, as both generally contain control fields. The term *trailer* is also sometimes used.

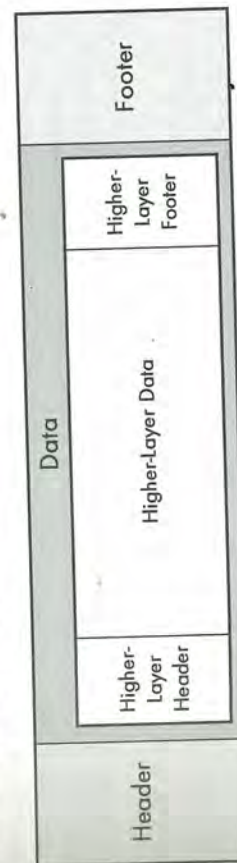


Figure 1-3: Network message formatting In the most general of terms, a message consists of a data payload that will be communicated, bracketed by a set of header and footer fields. The data of any particular message sent in a networking protocol will itself contain an encapsulated higher-layer message containing a header, data, and a footer. This “nesting” can occur many times as data is passed down a protocol stack. The header is found in most protocol messages; the footer only in some.

Since the header and footer can contain both control and information fields, you might rightly wonder what the point is of having a separate footer anyway. One reason is that some types of control information are calculated using the values of the data itself. In some cases, it is more efficient to perform this computation as the data payload is being sent, and then transmit the result after the payload in a footer. A good example of a field often found in a footer is redundancy data such as cyclic redundancy check (CRC) code, which can be used for error detection by the receiving device. Footers are most often associated with lower-layer protocols, especially at the data link layer of the OSI Reference Model.

KEY CONCEPT The general format of a networking message consists of a *header*, followed by the *data* or *payload* of the message, followed optionally by a *footer*. Header and footer information is functionally the same except for its position in the message; footer fields are only sometimes used, especially in cases where the data in the field is calculated based on the values of the data being transmitted.

Generally speaking, any particular protocol is concerned with only its *own* header (and footer, if present). It doesn't care much about what is in the data portion of the message, just as a delivery person worries only about driving the truck and not so much about what it contains. At the beginning of that data will normally be the headers of other protocols that were used higher up in the protocol stack; this, too, is shown in Figure 1-3. In the OSI Reference Model, a message handled by a particular protocol is said to be its PDU; the data it carries in its payload is its SDU. The SDU of a lower-layer protocol is usually a PDU of a higher-layer protocol. The discussion of data encapsulation in Chapter 5 contains a full explanation of this important concept.

Message Addressing and Transmission Methods: Unicast, Broadcast, and Multicast

In a networking technology that uses messages to send data, you must undertake a number of tasks in order to successfully transmit the data from one place to another. One is simply *addressing* the message—putting an address on it so that the system knows where it is supposed to go. Another is *transmitting* the message, which is sending it to its intended recipient.

There are several different ways of addressing and transmitting a message over a network. One way in which messages are differentiated is in how they are addressed and how many recipients will receive them. The method used depends on the function of the message and also on whether or not the sender knows specifically or generally whom they are trying to contact.

To help explain these different methods, I will use a real-world analogy. Consider a social function with 300 people that is being held in a large hall. These people are mingling and having different conversations. There are different kinds of messages that you may need to send in this setting, as is the case with networks.

Physical Layer (Layer 1)

The lowest layer of the OSI Reference Model is layer 1, the *physical layer*; it is commonly abbreviated PHY. This layer is the only one where data is physically moved across the network interface. All other layers perform functions to create messages that implement various protocols, but these messages must all be transmitted down the protocol stack to the physical layer, and they are eventually sent out over the network.

First, a bit of clarification. The name *physical layer* can be a bit problematic because it suggests that this layer relates only to the actual network hardware, which is not the case. While some people say that the physical layer is the network interface cards and cables, this is not actually true. The physical layer defines a number of network functions in addition to interfaces with hardware cables and cards.

People also suggest that all network hardware belongs to the physical layer. Again, this isn't strictly accurate. All hardware must have *some* relation to the physical layer in order to send data over the network, but hardware devices generally implement multiple layers of the OSI model in addition to the physical layer. For example, an Ethernet network interface card performs functions at both the physical layer and the data link layer.

The physical layer technologies deal with the actual ones and zeros that are sent over the network. For example, repeaters, conventional hubs, and transceivers all operate at the physical layer. These devices have no knowledge of the contents of a message; they simply take input bits and send them as output. The physical layer is responsible for the following:

Hardware Specifications Definition The details of operation of cables, connectors, wireless radio transceivers, network interface cards, and other hardware devices are generally a function of the physical layer (although also partially the data link layer, layer 2).

Encoding and Signaling The physical layer is responsible for various encoding and signaling functions that transform the data from bits that reside within a computer or another device into signals that can be sent over the network.

Data Transmission and Reception After encoding the data appropriately, the physical layer actually transmits the data, and of course, receives it. (This applies equally to wired and wireless networks, even if there is no tangible cable in a wireless network.)

Topology and Physical Network Design The physical layer is also considered the domain of many hardware-related network design issues, such as local area network (LAN) and wide area network (WAN) topology.

While the physical layer of a network primarily defines the hardware it uses, it is also closely related to the data link layer. Thus, it is not generally possible to define hardware at the physical layer independently from the technology being used at the data link layer. For example, Ethernet is a technology that describes specific types of cables and network hardware, but the physical layer of Ethernet can be isolated

from its data link layer aspects only to a point. Though Ethernet cables are the physical layer, the cables' maximum length is related closely to message format rules that exist at the data link layer.

Furthermore, some technologies perform functions at the physical layer that are normally more closely associated with the data link layer. For example, it is common to have the physical layer perform low-level (bit-level) repackaging of data link layer frames for transmission. Error detection and correction may also be done at layer 1 in some cases, though most people would consider these layer 2 functions.

In many technologies, a number of physical layers can be used with a data link layer. The classic example is Ethernet, for which dozens of different physical layer implementations exist. Each implementation uses the same data link layer (possibly with slight variations).

KEY CONCEPT The lowest layer in the OSI Reference Model is the *physical layer*. It is the realm of networking hardware specifications, and is the place where technologies that perform data encoding, signaling, transmission, and reception functions reside. The physical layer is closely related to the data link layer.

Many technologies further subdivide the physical layer into *sublayers* in order to allow different network media to be supported by the same technology, while sharing other functions at the physical layer that are common between the various media. A good example of this is the physical layer architecture used for Fast Ethernet, Gigabit Ethernet, and 10-Gigabit Ethernet.

NOTE In some contexts, the *physical layer technology that's used to convey bits across a network or communications line is called a transport method (not to be confused with the OSI transport layer, layer 4).*

Data Link Layer (Layer 2)

The second-lowest layer (layer 2) in the OSI Reference Model stack is the *data link layer*, often called simply the *link layer*, or abbreviated DLL. The data link layer is where many wired and wireless LAN technologies primarily function. For example, Ethernet, Token Ring, FDDI, and 802.11 (wireless Ethernet or Wi-Fi) are all sometimes called data link layer technologies. The set of devices connected at the data link layer is commonly considered a simple network (as opposed to an internetwork, which is a collection of networks connected at layer 3).

The data link layer is often conceptually divided into two sublayers: *logical link control (LLC)* and *media access control (MAC)*. This split is based on the architecture used in the IEEE 802 Project, which is the IEEE working group responsible for creating the standards that define many networking technologies. By separating LLC and MAC functions, interoperability of different networking technologies is made easier, as explained in the discussion of networking models in Chapter 5.

The following are the key tasks performed at the data link layer:

Logical Link Control (LLC) Logical link control refers to the functions required for the establishment and control of logical links between local devices on a network. This is usually considered a sublayer; it provides services to the network layer

above it and hides the rest of the details of the data link layer, which allows different technologies to work seamlessly with the higher layers. Most LAN technologies use the IEEE 802.2 LLC protocol to implement this part of the data link layer.

Media Access Control (MAC) This refers to the procedures used by devices to control access to the network medium. Since many networks use a shared medium (such as a single network cable, or a series of cables that are electrically connected into a single virtual medium), it is necessary to have rules for managing the medium to avoid conflicts. For example, Ethernet uses the CSMA/CD method of medium access control, while Token Ring uses token passing.

Data Framing The data link layer is responsible for data framing, which is the final encapsulation of higher-level messages into *frames* that are sent over the network at the physical layer.

Addressing The data link layer is the lowest layer in the OSI model that is concerned with addressing. It labels information with a particular destination local address. Each device on a network has a unique number that is used by the data link layer protocol to ensure that data intended for a specific machine gets to it properly. This is usually called a *hardware address* (since it is intimately related with low-level hardware) or a *MAC address* (after the MAC function described earlier).

Error Detection and Handling The data link layer handles errors that occur at lower levels of the network stack. For example, a cyclic redundancy check (CRC) field is often calculated based on the frame's contents and then included in it. It can be employed to allow the station receiving data to detect if it was received correctly.

Physical Layer Standards The physical layer and the data link layer are very closely related. The requirements for the physical layer of a network are often part of the data link layer standard that describes a particular technology. Certain physical-layer hardware and encoding aspects are specified by the data link layer technology being used. The best example of this is the Ethernet standard, IEEE 802.3, which specifies not just how Ethernet works at the data link layer, but also its various physical layers.

KEY CONCEPT The second OSI Reference Model layer is the data link layer. This is where most LAN and wireless LAN technologies are defined. Layer 2 is responsible for logical link control (LLC), media access control (MAC), hardware addressing, error detection and handling, and defining physical layer standards. It is often divided into the LLC and MAC sublayers based on the IEEE 802 Project that uses that architecture.

Many types of hardware are associated with the data link layer. Network interface cards typically implement a specific data link layer technology, so they are often called Ethernet cards, Token Ring cards, and so on. There are also a number of network interconnection devices that are said to operate at layer 2 in whole or in part because they make decisions about what to do with data they receive by looking at data link layer frames. These devices include most bridges, switches, and routers, though the latter two also encompass functions performed by layer 3.

Some of the most popular technologies and protocols generally associated with layer 2 are Ethernet, Token Ring, FDDI (plus CDDI), HomePNA, IEEE 802.11, Asynchronous Transfer Mode (ATM), TCP/IP's Serial Line Interface Protocol (SLIP), and TCP/IP's Point-to-Point Protocol (PPP).

Network Layer (Layer 3)

The third-lowest layer of the OSI Reference Model is the *network layer*. If the data link layer defines the boundaries of what is considered a network, the network layer defines how *internetworks* (interconnected networks) function. The network layer is the lowest one in the OSI model that is concerned with actually getting data from one computer to another even if it is on a remote network; in contrast, the data link layer only deals with devices that are local to each other.

While layers 2 through 6 all act as fences between the layers above and below them, the network layer is particularly important in terms of separating higher and lower-layer functions. It is here that the transition really begins from the more abstract functions of the higher layers—which don't concern themselves as much with data delivery—into the specific tasks required to get data to its destination. (The transport layer continues this abstraction transition as you go up the OSI protocol stack.)

Some of the specific jobs normally performed by the network layer include the following:

Logical Addressing Every device that communicates over a network has a logical address associated with it, which identifies the device regardless of its particular location. This is sometimes called a *layer 3 address*. For example, on the Internet, the Internet Protocol (IP) is the network layer protocol and every machine has an IP address. Logical addresses are independent of particular hardware and must be unique across an entire internetwork.

NOTE Addressing is done at the data link layer as well, but those addresses refer to local physical devices.

Routing The defining function of the network layer is routing—moving data across a series of interconnected networks. It is the job of the devices and software routines that function at the network layer to handle incoming packets from various sources, determine their final destination, and then figure out where they need to be sent to get them where they are supposed to go. (You'll find a more complete discussion of routing in the OSI model in the section covering indirect device connection in Chapter 5.)

Datagram Encapsulation The network layer normally encapsulates messages received from higher layers by placing them into *datagrams* (also called *packets*) with a network layer header (the previous chapter discusses encapsulation).

Fragmentation and Reassembly The network layer must send messages down to the data link layer for transmission. Some data link layer technologies limit the length of any message that can be sent. If the packet that the network layer wants to send is too large, the network layer must split the packet up (fragment it), send each

doesn't. Again, this is the case with TCP/IP: There is one main transport layer protocol, TCP, that includes reliability and flow-control features, and a second, User Datagram Protocol (UDP), that doesn't.

Let's look at the specific functions often performed at the transport layer in more detail:

Process-Level Addressing Addressing at the transport layer is used to differentiate between software programs. This is part of what enables many different software programs to use a network layer protocol simultaneously. The best example of transport-layer process-level addressing is the TCP and UDP port mechanism that is used in TCP/IP, which allows applications to be individually referenced on any TCP/IP device.

Multiplexing and Demultiplexing Using the process-level addresses, transport layer protocols on a sending device *multiplex* the data received from many application programs for transport, combining them into a single stream of data to be sent. The same protocols receive data and then *demultiplex* it from the incoming stream of datagrams, and direct each one to the appropriate recipient application processes.

Segmentation, Packaging, and Reassembly The transport layer segments the large amounts of data it sends over the network into smaller pieces on the source machine and then reassembles them on the destination machine. This function is similar to the fragmentation function of the network layer. Just as the network layer fragments messages to fit the limits of the data link layer, the transport layer segments messages to suit the requirements of the underlying network layer.

Connection Establishment, Management, and Termination Transport layer connection-oriented protocols are responsible for the series of communications required to establish a connection, maintain it as data is sent over it, and then terminate the connection when it is no longer required.

Acknowledgments and Retransmissions As mentioned earlier, the transport layer is where many protocols that guarantee reliable delivery of data are implemented. This is done using a variety of techniques, most commonly by combining *acknowledgments* and *retransmission timers*. The sending device starts a timer on each occasion that data is sent; if the data is received, the recipient sends back an acknowledgment to the sender to indicate successful transmission. If no acknowledgment is returned before the timer expires, the data is retransmitted. Other algorithms and techniques are usually required to support this basic process.

Flow Control Transport layer protocols that offer reliable delivery also often implement *flow-control* features. These features allow one device in a communication to specify to another that it must throttle back the rate at which it is sending data. This will prevent the receiver from being bogged down with data. These features allow mismatches in speed between sender and receiver to be detected and handled.

Compression Compression (and decompression) may be done at the presentation layer to improve the throughput of data.

Encryption Some types of encryption (and decryption) are performed at the presentation layer to ensure the security of the data as it travels down the protocol stack. For example, one of the most popular encryption schemes usually associated with the presentation layer is the Secure Sockets Layer (SSL) protocol. (Some encryption is done at lower layers in the protocol stack in technologies such as IPsec.)

The presentation layer is not always used in network communications because these functions mentioned are simply not always needed. Compression and encryption are usually considered optional, and translation features are needed only in certain circumstances. Also, the presentation layer's functions may be performed at the application layer.

NOTE Since its translation job isn't always needed, the presentation layer is commonly skipped by actual protocol stack implementations; in such implementations protocols at layer 7 may talk directly with those at layer 5. This is part of the reason why all of the functions at layers 5 through 7 may be included in the same software package, as described in the overview of layers and layer groupings in the previous chapter.

KEY CONCEPT The sixth OSI model layer is the *presentation layer*. Protocols at this layer take care of manipulation tasks that transform data from one representation to another, such as translation, compression, and encryption. In many cases, no such functions are required in a particular networking stack; if so, there may not be any protocol active at layer 6, so layer 7 may deal with layer 5.

Application Layer (Layer 7)

At the very top of the OSI Reference Model stack of layers, you find layer 7, the *application layer*. Continuing the trend that you saw in layers 5 and 6, this one is also named very appropriately. The application layer is the one that is used by network applications. These programs are what actually implement the functions performed by users to accomplish various tasks over the network.

It's important to understand that what the OSI model calls an application is not exactly the same as what you normally think of as an application. In the OSI model, the application layer provides services for user applications to employ.

For example, when you use your web browser, that actual software is an application running on your PC. It doesn't really reside at the application layer. Rather, it makes use of the services offered by a protocol that operates at the application layer, which is called the Hypertext Transfer Protocol (HTTP). The distinction between the browser and HTTP is subtle but important.

Not all user applications use the network's application layer in the same way. Sure, your web browser, email client, and Usenet newsreader do, but if you open a file over the network with a text editor, that editor is not using the application layer—it just sees a file addressed with a name that has been mapped to a network somewhere else. The operating system redirects what the editor does, over the network.

Similarly, not all uses of the application layer are by applications. The operating system itself can (and does) use services directly at the application layer.

That caveat aside, under normal circumstances, whenever you interact with a program on your computer that is designed specifically for use on a network, you are dealing directly with the application layer. For example, sending an email message, firing up a web browser, and using a chat program involve protocols that reside at the application layer.

NOTE *There are dozens of different application layer protocols. Some of the most popular ones include HTTP, FTP, SMTP, DHCP, NFS, Telnet, SNMP, POP3, NNTP, and IRC. I describe all of these and more in Section III.*

As the top-of-the-stack layer, the application layer is the only one that does not provide any services to the layer above it in the stack—there isn't one! Instead, it provides services to programs that want to use the network, and to you, the user. So the responsibilities at this layer are simply to implement the functions that are needed by users of the network and to issue the appropriate commands to make use of the services provided by the lower layers.

KEY CONCEPT The *application layer* is the seventh and highest layer in the OSI Reference Model. Application protocols that implement specific user applications and other high-level functions are defined at this layer. Since they are at the top of the stack, application protocols are the only ones that do not provide services to a higher layer; they use services provided by the layers below.

As you've seen, the distinctions between the top three layers in the OSI Model are not very clear. In the case of TCP/IP, this is exacerbated by the decision not to separate out the session, presentation, and application layer functions. All of the protocols mentioned earlier are from the TCP/IP protocol family, and some may cover all three of the top three OSI layers, two of them, or one; in the TCP/IP model, they are all just considered applications.

NOTE The client and server roles I have discussed are the traditional ones. The rise of powerful personal computers and widespread Internet access (especially always-on broadband connectivity) has led to a significant blurring of client and server hardware and software. Many client machines now include server software that allows them to, for example, respond to World Wide Web queries from other clients. Also, many file-sharing programs allow clients to communicate using the peer-to-peer structural model. However, most TCP/IP communication is still client/server in nature, so it's important to keep these roles in mind.

TCP/IP Architecture and the TCP/IP Model

The OSI Reference Model's seven layers divide up the tasks required to implement a network, as described in Part 1-2 of this book. However, it is not the only such model. In fact, the TCP/IP protocol suite was developed before the OSI Reference Model; as such, its inventors didn't use the OSI model to explain TCP/IP architecture (even though the OSI model is often used in TCP/IP discussions today). The developers of TCP/IP created their own architectural model, which goes by different names including the *TCP/IP model*, the *DARPA model* (after the agency that was largely responsible for developing TCP/IP), and the *DoD model* (after the United States Department of Defense). Most people call it the TCP/IP model.

Regardless of the model you use to represent the function of a network, the model's functions are pretty much the same. The TCP/IP and the OSI models are really quite similar, even if they don't carve up the network functionality precisely the same way.

Since the OSI model is so widely used, it is common to explain the TCP/IP architecture both in terms of the TCP/IP layers and the corresponding OSI layers. Figure 8-2 shows the relationship between the two models. The TCP/IP model does not address the physical layer, where hardware devices reside. The next three layers—*network interface*, *internet*, and *host-to-host transport*—correspond to layers 2, 3, and 4 of the OSI model. The TCP/IP *application* layer conceptually blurs the top three OSI layers. Note, too, that some people consider certain aspects of the OSI session layer to be part of the TCP/IP host-to-host transport layer.

As shown in Figure 8-2, the TCP/IP model uses four layers that logically span the equivalent of the top six layers of the OSI model. (The physical layer is not covered by the TCP/IP model because the data link layer is considered the point at which the interface occurs between the TCP/IP stack and the underlying networking hardware.) Starting from the bottom, the TCP/IP layers are described in the following sections.

Network Interface Layer

As its name suggests, the network interface layer is where the actual TCP/IP protocols running at higher layers interface to the local network. This layer is somewhat controversial in that some people don't even consider it a legitimate part of TCP/IP, usually because none of the core IP protocols run at this layer. Despite this, the network interface layer is part of the architecture. It is equivalent to the data link layer (layer 2) in the OSI Reference Model (see Chapter 6) and is also sometimes called the *link layer*. You may also see the name *network access layer* used

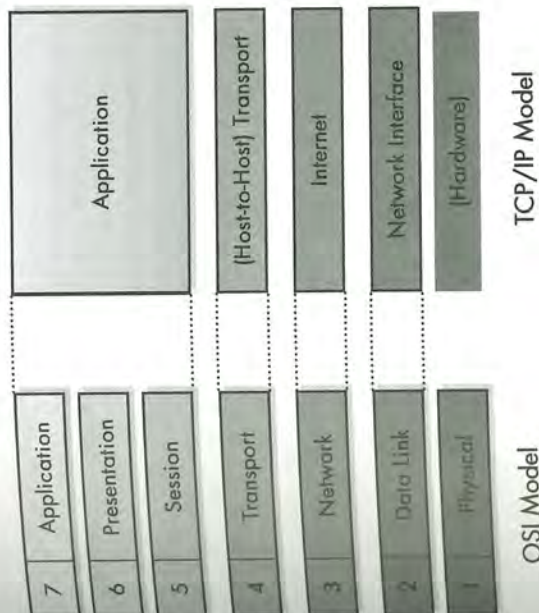


Figure 8-2: OSI Reference Model and TCP/IP model layers The TCP/IP architectural model has four layers that approximately match six of the seven layers in the OSI Reference Model.

On many TCP/IP networks, there is no TCP/IP protocol running at all on this layer, because it is simply not needed. For example, if you run TCP/IP over Ethernet, then Ethernet handles layer 2 (and layer 1) functions. However, the TCP/IP standards do define protocols for TCP/IP networks that do not have their own layer 2 implementation. These protocols, the Serial Line Internet Protocol (SLIP) and the Point-to-Point Protocol (PPP), fill the gap between the network layer and the physical layer. They are commonly used to facilitate TCP/IP over direct serial line connections (such as dial-up telephone networking) and other technologies that operate directly at the physical layer.

Internet Layer

The Internet layer corresponds to the network layer in the OSI Reference Model (thus it is sometimes called the *network layer* even in TCP/IP model discussions). It is responsible for typical layer 3 jobs, such as logical device addressing, data packaging, manipulation and delivery, and routing. At this layer, you find IP, which is arguably the heart of TCP/IP, as well as support protocols such as the Internet Control Message Protocol (ICMP) and the routing protocols (RIP, OSPF, BGP, and so on). IP version 6, the next-generation IP, is also at this layer.

Host-to-Host Transport Layer

This primary job of the host-to-host transport layer is to facilitate end-to-end communication over an internetwork. It is in charge of allowing logical connections to be made between devices that allow data to be sent either unreliably (with no guarantee that it gets there) or reliably (where the protocol keeps track of the data sent and received in order to make sure it arrives, and resends it if necessary). It is also here that identification of the specific source and destination application process is accomplished.

The formal name of this layer is often shortened to just the *transport layer*. The key TCP/IP protocols at this layer are TCP and UDP. The TCP/IP transport layer corresponds to the layer of the same name in the OSI model (layer 4) but includes certain elements that are arguably part of the OSI session layer. For example, TCP establishes a connection that can persist for a long period of time, which some people say makes a TCP connection more like a session.

Application Layer

The application layer is the highest layer in the TCP/IP model. It is a rather broad layer, encompassing layers 5 through 7 in the OSI model. While this seems to represent a loss of detail compared to the OSI model, that's probably a good thing. The TCP/IP model better reflects the somewhat fuzzy nature of the divisions between the functions of the higher layers in the OSI model, which in practical terms often seem rather arbitrary. It really is hard to separate some protocols in terms of which portions of layers 5, 6, or 7 they encompass.

Numerous protocols reside at the application layer. These include application protocols such as HTTP, FTP, and SMTP for providing end-user services as well as administrative protocols like Simple Network Management Protocol (SNMP), Dynamic Host Configuration Protocol (DHCP), and Domain Name System (DNS).

NOTE *The Internet and host-to-host transport layers are usually considered the core of TCP/IP architecture, because they contain most of the key protocols that implement TCP/IP internetworking.*

In the following section, I provide a brief look at each of the TCP/IP protocols covered in detail in this book and offer more detail on where they all fit into the TCP/IP architecture. I also discuss a couple of protocols that, interestingly, don't really fit into the TCP/IP layer model very well at all.

The Internet Protocol Control Protocol (IPCP): An Example NCP

Let's look at the NCP for IP: IPCP. When PPP is set up to carry IP datagrams, IPCP is invoked in the network layer protocol phase to set up an IP NCP link between the two devices. The setup is carried out using the four Configure- messages. For IP, the configuration options can be specified in an IPCP Configure-Request message.

IP Compression Protocol Allows devices to negotiate the use of Van Jacobson TCP/IP header compression, which shrinks the size of TCP and IP headers to save bandwidth. This is similar in concept to the Protocol-Field-Compression and ACK options in LCP.

IP Address Allows the device sending the Configure-Request message either to specify an IP address it wants to use for routing IP over the PPP link or to request that the other device supply it with one. This is most commonly used for dial-up networking links.

Once configuration is complete, data can be sent for the layer 3 protocol corresponding to the NCP negotiated. This is indicated by using the appropriate value for the Protocol field in PPP data frames containing that layer 3 data.

PPP Authentication Protocols: PAP and CHAP

PPP was designed to provide layer 2 connectivity over a variety of serial links and other physical layer technologies, some of which introduce more security concerns than others. For example, suppose you connect two machines in your office with a serial cable and want to run PPP between them. When one of these initiates a PPP link with the other, you don't really need to worry about who's calling. On the other hand, consider an Internet service provider (ISP) using PPP for remote dial-in users. They generally want to allow only their customers to connect.

The PPP protocol suite allows for the use of an optional authentication protocol when devices negotiate the basic link setup. The PPP suite initially defined two authentication protocols: PAP and CHAP. Once an LCP link is set up between two devices, a series of authentication messages are sent using these protocols to verify the identity of the device initiating the link. Only if authentication is successful can the link configuration proceed.

PAP

PAP is a very straightforward authentication scheme, consisting of only two basic steps, as shown in Figure 10-4.

Authentication Request The initiating device sends an Authenticate-Request message that contains a name and a password.

Authentication Reply The responding device looks at the name and password and decides whether to accept the initiating device and continue setting up the link. If so, it sends back an Authenticate-Ack message. Otherwise, it sends an Authenticate-Nak message.

PAP is another example of something that is just too simple for its own good. Chief among its flaws is that it transmits the user name and password in clear text across the link. This is a big "no-no" because eavesdroppers can get the password.

PAP also provides no protection against various security attacks. For example, an unauthorized user could try different passwords indefinitely until he discovered the correct one. PAP also puts control of the authentication squarely on the shoulders of the initiating device (usually a client machine), which is not considered desirable, because this is normally a server function that administrators prefer to manage.

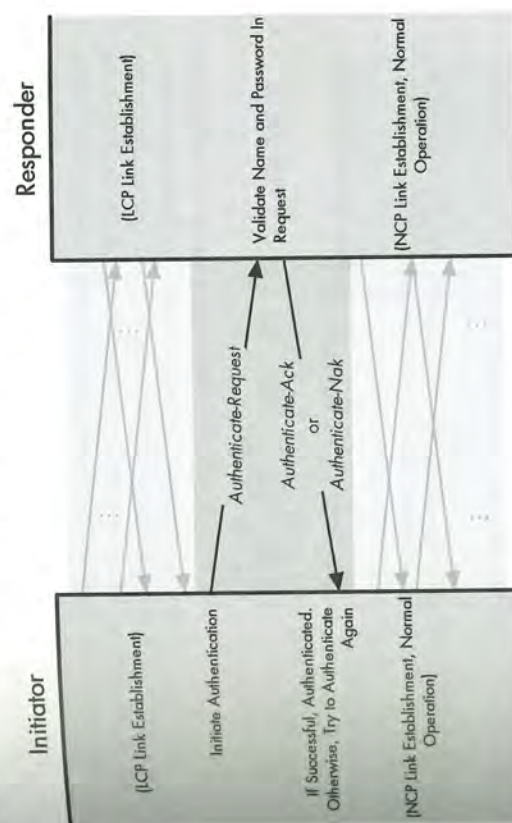


Figure 10-4: PAP authentication PAP uses a simple exchange of a request containing name and password information and a reply indicating whether or not authentication was successful.

CHAP

The most important difference between PAP and CHAP is that CHAP doesn't transmit the password across the link. When using PAP, the initiator (calling client) sends the authenticator (generally the server that is deciding whether to grant authentication) a message saying essentially, "Here's the password I know; see if it matches yours." Each device uses the password to perform a cryptographic computation, and then checks to see if it gets the same result. If so, they know they have the same password.

In CHAP, a basic LCP link is first set up between the initiator and authenticator. The authenticator then takes charge of the authentication process, using a technique called a three-way handshake.

NOTE Three-way handshakes are a fairly common general authentication procedure. The same basic technique is used, for example, in shared key authentication on IEEE 802.11 wireless networking.

The three-way handshake steps are as follows (see Figure 10-5):

Challenge The authenticator generates a frame called a Challenge and sends it to the initiator. This frame contains a simple text message (sometimes called the *challenge text*). The message has no inherent special meaning, so it doesn't matter anyone intercepts it. The important thing is that after receipt of the Challenge, both devices have the same Challenge message.

Response The initiator uses its password (or some other shared secret that the authenticators also know) to encrypt the challenge text. It then sends the encrypted challenge text as a Response back to the authenticator.

Success or Failure The authenticator performs the same encryption on the challenge text that the initiator did. If the authenticator gets the same result that the initiator sent it in the Response, it knows that the initiator had the right password when it did its encryption, so the authenticator returns a Success message. Otherwise, it sends a Failure message.

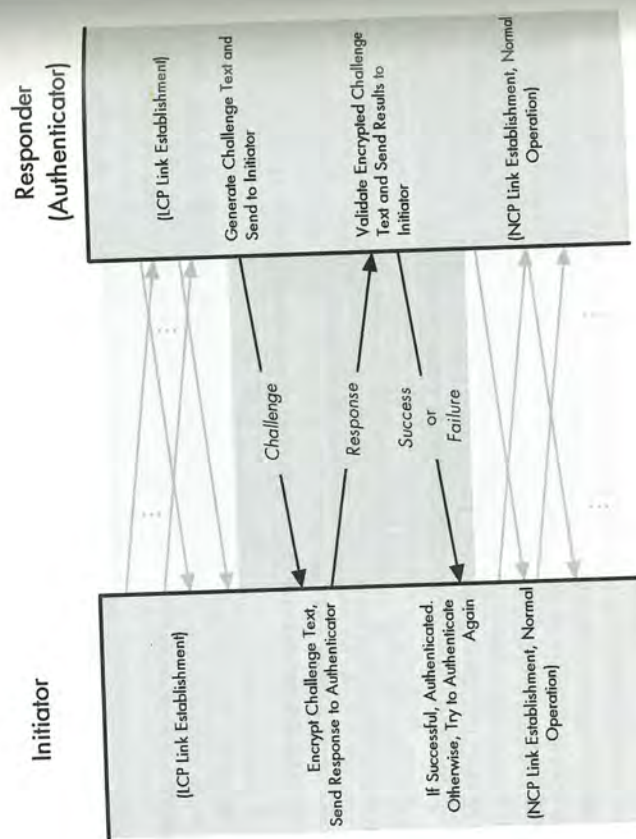


Figure 10-5: PPP Challenge Handshake Authentication Protocol (CHAP) authentication CHAP uses a three-way handshake beginning with a Challenge from the authenticating device. This message is encrypted and returned to the authenticating device, which checks to see if the device trying to authenticate used the correct password (or other shared secret).

The beauty of this is that it verifies that the two devices have the same shared secret, but it doesn't require them to send the secret over the link. The Response is calculated based on the password, but the content of the Response is encrypted, and thus it's much harder to derive the password from. CHAP also provides protection against replay attacks, whereby an unauthorized user captures a message and tries to send it again later on. This is done by changing an identifier in each message and varying the challenge text. Also, in CHAP, the server controls the authentication process, not the client that is initiating the link.

KEY CONCEPT PPP supports two authentication protocols: PAP and CHAP. PAP is a simple request-and-reply authentication protocol that is widely considered to be inadequate because it sends the user name and password in clear text and provides little protection against many security concerns. CHAP uses a three-way handshake procedure and is preferred over PAP in most implementations.

CHAP is not perfect, but it's much better than PAP. In fact, the IETF made a rather strong statement in this regard when it revised the original RFC that described PAP and CHAP to include only CHAP in the new standard. Despite this, PAP is still used in some applications because it is simple. PAP can suffice in situations where security is not a big deal, but CHAP is a much better choice.

NOTE Incidentally, in addition to PAP and CHAP, it is possible to use proprietary authentication schemes. This requires that the appropriate configuration option values be programmed into LCP for placement in the Authentication Protocol configuration option.

0x00FD (hexadecimal) is placed in the PPP Protocol field. When compression is used with multiple links and the links are compressed independently, a different value is used: 0x00FB.

You'll recall that in a regular uncompressed frame, the Protocol field indicates which layer 3 protocol the data comes from. Since you still need to know this, the original Protocol value is actually prepended to the data before compression. When the data is decompressed, this value is used to restore the original Protocol field, so the receiving device knows to which higher layer the data belongs.

For example, if you use IPCP to encapsulate IP data in PPP, the uncompressed frame would have a value of 0x8021 in the Protocol field. This value (0x8021) would be placed at the start of the data to be compressed. The compressed data would be put in a PPP frame with a Protocol value of 0x00FD. The receiving device would see the value 0x00FD in the Protocol field, recognize the frame as compressed, decompress it, and restore the original frame with 0x8021 as the Protocol value. The discussion of the PPP general frame format in Chapter 12 covers this in more detail.

In theory, a compression algorithm can put more than one PPP data frame into a compressed PPP data frame. Despite this, many, if not most, of the algorithms maintain a one-to-one correspondence, putting each PPP data frame into one compressed frame. Note that LCP frames are not compressed, nor are the control frames used for other protocols. For example, a data frame carrying IP traffic would be compressed, but a control frame for IPCP (the NCP for IP) would not be.

Compression can be combined with encryption. In this case, compression is done before encryption.

NOTE The compression performed by CCP has nothing to do with the header compression options that can be negotiated as part of LCP. That type of compression doesn't involve compressing a data stream using a compression algorithm, but rather a simple way of saving space in headers when both ends of a link agree to do so.

PPP Encryption Control Protocol (ECP) and Encryption Algorithms

The PPP authentication protocols Password Authentication Protocol (PAP) and Challenge Handshake Authentication Protocol (CHAP) can be used to ensure the only authorized devices can establish a PPP connection. Once that is done, PPP normally provides no other security to the data being transmitted. In particular, data is normally sent in the clear (unencrypted), thereby making it easy for someone who intercepts it to read.

For important data that must be kept secure, encryption prior to transmission is a good idea. This can be done at higher layers using something like IPsec, but PPP also provides an optional feature that allows data to be encrypted and decrypted at the data link layer itself using two protocol components:

PPP Encryption Control Protocol (ECP) This protocol is responsible for negotiating and managing the use of encryption on a PPP link.

PPP Encryption Algorithms A family of encryption algorithms that perform the actual encryption and decryption of data. Several of these are defined in RFCs, and two devices can also negotiate a proprietary encryption method if they want to use one that isn't defined by a public standard.

KEY CONCEPT PPP includes an optional encryption feature that provides privacy for data transported over PPP. A number of encryption algorithms are supported. To enable encryption, both devices on a PPP link use the *PPP Encryption Control Protocol (ECP)* to negotiate which algorithm to use. The selected algorithm is then used to encrypt and decrypt PPP data frames.

ECP Operation: Encryption Setup

ECP is usually the only part mentioned when encryption in PPP is discussed, but it is actually used only to configure and control the use of encryption; the algorithms do the real work. This technique allows each implementation to choose which type of encryption it wishes to use.

The original ECP defined only a single encryption method, and a couple of others have since been added. Like CCP, ECP is analogous to the NCPs that negotiate parameters specific to a network layer protocol sent on the link, but it deals with how devices encrypt data, rather than how they transport layer 3 traffic. This also means that like the NCPs, ECP is a light version of LCP and works in the same basic way. Once an ECP link is negotiated, devices can send encrypted frames between each other. When no longer needed, the ECP link can be terminated.

ECP uses the same subset of seven LCP message types that the NCPs use, and it adds two more. The use of these messages for each of the life stages of an ECP link is as follows:

Link Configuration Like the NCPs (and also like CCP, of course), encryption configuration is done once ECP reaches the *network layer protocol* phase. The process of setting up encryption and negotiating parameters is accomplished using Configure-Request, Configure-Ack, Configure-Nak, and Configure-Reject messages, as I explained in the description of LCP in Chapter 10, except the configuration options are particular to ECP.

Link Maintenance Code-Reject messages can be sent to indicate invalid code values in ECP frames. The two new message types are Reset-Request and Reset-Ack, which are used to reset the encryption (the ECP link) in the event of a detected failure in decryption.

Link Termination An ECP link can be terminated using Terminate-Request and Terminate-Ack. Again, remember that like the NCP links, the ECP link is set up within an LCP link, so closing it doesn't terminate the LCP link.

ECP Configuration Options and Encryption Algorithms

ECP configuration options are used solely to negotiate the type of encryption algorithm that will be used by the two devices and the specifics of how that algorithm will be employed. The device initiating the negotiation sends a Configure-Request

0x00FD (hexadecimal) is placed in the PPP Protocol field. When compression is used with multiple links and the links are compressed independently, a different value is used: 0x00FB.

You'll recall that in a regular uncompressed frame, the Protocol field indicates which layer 3 protocol the data comes from. Since you still need to know this, the original Protocol value is actually prepended to the data before compression. When the data is decompressed, this value is used to restore the original Protocol field, so the receiving device knows to which higher layer the data belongs.

For example, if you use IPCP to encapsulate IP data in PPP, the uncompressed frame would have a value of 0x8021 in the Protocol field. This value (0x8021) would be placed at the start of the data to be compressed. The compressed data would be put in a PPP frame with a Protocol value of 0x00FD. The receiving device would see the value 0x00FD in the Protocol field, recognize the frame as compressed, decompress it, and restore the original frame with 0x8021 as the Protocol value. The discussion of the PPP general frame format in Chapter 12 covers this in more detail.

In theory, a compression algorithm can put more than one PPP data frame into a compressed PPP data frame. Despite this, many, if not most, of the algorithms maintain a one-to-one correspondence, putting each PPP data frame into one compressed frame. Note that LCP frames are not compressed, nor are the control frames used for other protocols. For example, a data frame carrying IP traffic would be compressed, but a control frame for IPCP (the NCP for IP) would not be.

Compression can be combined with encryption. In this case, compression is done before encryption.

NOTE The compression performed by CCP has nothing to do with the header compression options that can be negotiated as part of LCP. That type of compression doesn't involve compressing a data stream using a compression algorithm, but rather a simple way of saving space in headers when both ends of a link agree to do so.

PPP Encryption Control Protocol (ECP) and Encryption Algorithms

The PPP authentication protocols Password Authentication Protocol (PAP) and Challenge Handshake Authentication Protocol (CHAP) can be used to ensure the only authorized devices can establish a PPP connection. Once that is done, PPP normally provides no other security to the data being transmitted. In particular, data is normally sent in the clear (unencrypted), thereby making it easy for someone who intercepts it to read.

For important data that must be kept secure, encryption prior to transmission is a good idea. This can be done at higher layers using something like IPsec, but PPP also provides an optional feature that allows data to be encrypted and decrypted at the data link layer itself using two protocol components:

PPP Encryption Control Protocol (ECP) This protocol is responsible for negotiating and managing the use of encryption on a PPP link.

PPP Encryption Algorithms A family of encryption algorithms that perform the actual encryption and decryption of data. Several of these are defined in RFCs, and two devices can also negotiate a proprietary encryption method if they want to use one that isn't defined by a public standard.

KEY CONCEPT PPP includes an optional encryption feature that provides privacy for data transported over PPP. A number of encryption algorithms are supported. To enable encryption, both devices on a PPP link use the *PPP Encryption Control Protocol (ECP)* to negotiate which algorithm to use. The selected algorithm is then used to encrypt and decrypt PPP data frames.

ECP Operation: Encryption Setup

ECP is usually the only part mentioned when encryption in PPP is discussed, but it is actually used only to configure and control the use of encryption; the algorithms do the real work. This technique allows each implementation to choose which type of encryption it wishes to use.

The original ECP defined only a single encryption method, and a couple of others have since been added. Like CCP, ECP is analogous to the NCPs that negotiate parameters specific to a network layer protocol sent on the link, but it deals with how devices encrypt data, rather than how they transport layer 3 traffic. This also means that like the NCPs, ECP is a light version of LCP and works in the same basic way. Once an ECP link is negotiated, devices can send encrypted frames between each other. When no longer needed, the ECP link can be terminated.

ECP uses the same subset of seven LCP message types that the NCPs use, and it adds two more. The use of these messages for each of the life stages of an ECP link is as follows:

Link Configuration Like the NCPs (and also like CCP, of course), encryption configuration is done once ECP reaches the *network layer protocol* phase. The process of setting up encryption and negotiating parameters is accomplished using Configure-Request, Configure-Ack, Configure-Nak, and Configure-Reject messages, as I explained in the description of LCP in Chapter 10, except the configuration options are particular to ECP.

Link Maintenance Code-Reject messages can be sent to indicate invalid code values in ECP frames. The two new message types are Reset-Request and Reset-Ack, which are used to reset the encryption (the ECP link) in the event of a detected failure in decryption.

Link Termination An ECP link can be terminated using Terminate-Request and Terminate-Ack. Again, remember that like the NCP links, the ECP link is set up within an LCP link, so closing it doesn't terminate the LCP link.

ECP Configuration Options and Encryption Algorithms

ECP configuration options are used solely to negotiate the type of encryption algorithm that will be used by the two devices and the specifics of how that algorithm will be employed. The device initiating the negotiation sends a Configure-Request

with one option for each of the encryption algorithms it supports. The other device compares this list of options to the algorithms it understands. It also checks for details relevant to the option to see if it agrees on how that algorithm should be used. It then sends back the appropriate reply (Ack, Nak, or Reject), and a negotiation ensues until the two devices come up with a common algorithm that they both understand. If so, encryption is enabled; otherwise, it is turned off.

The ECP configuration options begin with a Type value that indicates the encryption algorithm. When the Type value is 0, this indicates that the option contains information about a special, proprietary encryption method that isn't covered by any RFC standards, which can be used if both devices understand it. Values in the range from 1 to 254 indicate encryption algorithms that have been defined for use with ECP; at present, only two are defined. Table 11-2 shows the values of the Type field, including the encryption algorithm each corresponds to and the number of the RFC that defines it.

Table 11-2: PPP Encryption Control Protocol (ECP) Compression Algorithms

ECP Option Type Value	Defining RFC	Encryption Algorithm (As Given in RFC Title)
0	—	Proprietary
2	2420	The PPP Triple-DES Encryption Protocol (3DESE)
3	2419	The PPP DES Encryption Protocol, Version 2 (DESE-bis)

NOTE Type value 1 was for the original DES algorithm, which was defined in RFC 1969. It was superseded by DES version 2 in RFC 2419.

Encryption Algorithm Operation: Encrypting and Decrypting Data

Once an encryption algorithm has been successfully negotiated, it is used to encrypt data before transmission and to decrypt data that has been received. To encrypt, the transmitting device takes the data that would normally be put in the Information field of an unencrypted PPP frame and runs it through the encryption algorithm. To indicate that a frame has been encrypted, the special value 0x0053 (hexadecimal) is placed in the PPP Protocol field. When encryption is used with multiple links and the links are encrypted independently, a different value is used, 0x0055.

You'll recall that in a regular unencrypted frame, the Protocol field indicates which layer 3 protocol the data comes from. Since you still need to know this, the original Protocol value is actually prepended to the data before encryption. When the data is decrypted, this value is used to restore the original Protocol field, so the receiving device knows which higher layer the data belongs to.

For example, if you use IPCP to encapsulate IP data in PPP, the unencrypted frame would have a value of 0x8021 (hex) in the Protocol field. This value (0x8021) would be placed at the start of the data to be encrypted. The encrypted data would be put in a PPP frame with a Protocol value of 0x0053. The receiving device would

see the value 0x0053 in the Protocol field, recognize the frame as encrypted, decrypt it, and restore the original frame with 0x8021 as the Protocol value. The discussion of the PPP general frame format in Chapter 12 covers this more completely.

Each encrypted PPP data frame carries exactly one PPP data frame. Note that, unlike what you saw in compression, LCP frames and the control frames used for other protocols *can* be encrypted. Compression can be combined with encryption; in this case, compression is done before encryption.

PPP Multilink Protocol (MP, MLP, MLPP)

Most of the time, there is only a single physical layer link between two devices. However, there are some situations for which there may actually be two layer 1 connections between the same pair of devices. This may seem strange. Why would there be more than one link between any pair of machines?

There are a number of situations in which this can occur. A common one is when two links are intentionally placed between a pair of devices. This is often done to increase performance by widening the pipe between two devices, without going to a newer, more expensive technology. For example, if two machines are connected to each other using a regular analog modem that's too slow, a relatively simple solution is to use two analog modem pairs connecting the machines to double bandwidth.

A slightly different situation occurs when multiplexing creates the equivalent of several physical layer channels between two devices, even if they have only one hardware link between them. Consider ISDN, for example. The most common form of ISDN service (ISDN basic rate interface or BRI) creates two 64,000 bps *B channels* between a pair of devices. These B channels are time division multiplexed and carried along with a D channel on a single pair of copper wire, but to the devices, they appear *as if* there were two physical layer links between devices, each of which carries 64 Kbps of data. And the ISDN primary rate interface (PRI) actually creates 23 or more channels, all between the same pair of hardware devices.

In a situation where you have multiple links, you could just establish PPP over each connection independently. However, this is far from an ideal solution, because you would then have to manually distribute the traffic over the two (or more) channels or links that connect them. If you wanted to connect to the Internet, you would need to make separate connections and then choose which one to use for each action. That isn't exactly a recipe for fun, and what's worse is that you could never use all the bandwidth for a single purpose, such as downloading the latest 100 MB Microsoft security patch.

What you really want is a solution that will let you combine multiple links and use them as if they were one high-performance link. Some hardware devices actually allow this to be done at the hardware level itself. In ISDN, this technology is sometimes called *bonding* when done at layer 1. For those hardware units that don't provide this capability, PPP makes it available in the form of the PPP Multilink Protocol (MP). This protocol was originally described in RFC 1717 and was updated in RFC 1990.

NOTE The PPP Multilink Protocol is properly abbreviated MP, but it is common to see any of a multitude of other abbreviations used for it. Many of these are actually derived from changing the order of the words in the name into Multilink PPP, so you will frequently see this called ML PP, MLPPP, MPPP, MLP, and so forth. These are technically incorrect, but widely used, especially MLPPP. I use the correct abbreviation in this book.

PPP Multilink Protocol Architecture

MP is an optional feature of PPP, so it must be designed to integrate seamlessly into regular PPP operation. To accomplish this, MP is implemented as a new architectural sublayer within PPP. In essence, an MP sublayer is inserted between the regular PPP mechanism and any network layer protocols using PPP, as shown in Figure 11-1. This allows MP to take all network layer data to be sent over the PPP link and spread it over multiple physical connections, without causing either the normal PPP mechanisms or the network layer protocol interfaces to PPP to break.

The column on the left in Figure 11-1 shows the TCP/IP model architecture with corresponding OSI Reference Model layer numbers. The center column shows the normal PPP layer architecture. When MP is used, there are separate PPP implementations running over each of two or more physical links. MP sits, architecturally, between these links and any network layer protocols that will be transported over those links. (In this diagram, only IP is shown because it is most common, but MP can work with multiple network layer protocols, each of which are being sent over each physical link.)

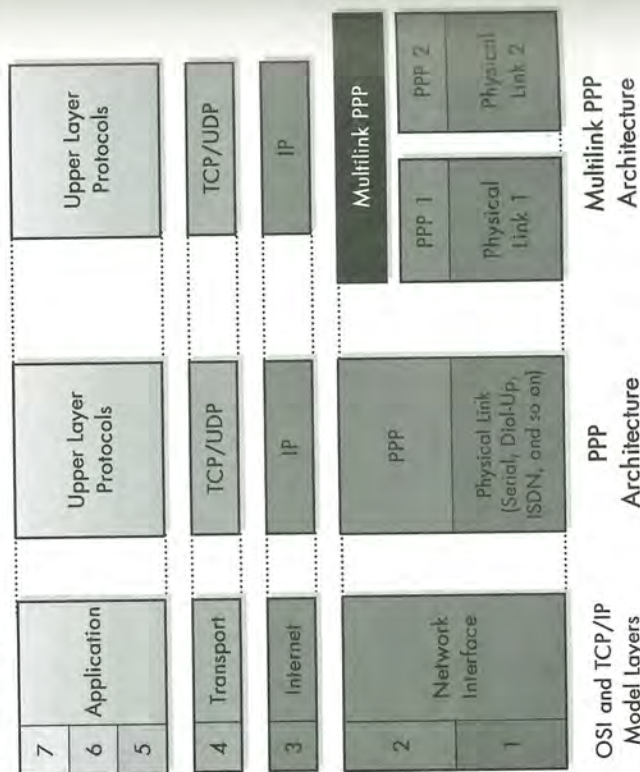


Figure 11-1: Multilink PPP architecture When Multilink PPP is used to combine two or more physical links, it sits architecturally above the PPP layers that operate on each physical link.

KEY CONCEPT The PPP Multilink Protocol (MP) allows PPP to bundle multiple physical links and use them like a single, high-capacity link. It must be enabled during link configuration. Once operational, it works by fragmenting whole PPP frames and sending the fragments over different physical links.

PPP Multilink Protocol Setup and Configuration

To use MP, both devices must have it implemented as part of their PPP software and must negotiate its use. This is done by LCP as part of the negotiation of basic link parameters in the *link establishment* phase (just like LQR, as described earlier in this chapter). Three new configuration options are defined to be negotiated to enable MP:

Multilink Maximum Received Reconstructed Unit Provides the basic indication that the device starting the negotiation supports MP and wants to use it. The option contains a value specifying the maximum size of the PPP frame it supports. If the device receiving this option does not support MP, it must respond with a Configure-Reject LCP message.

Multilink Short Sequence Number Header Format Allows devices to negotiate the use of a shorter sequence number field for MP frames, for efficiency. (See the section on MP frames in Chapter 12 for a full discussion.)

Endpoint Discriminator Uniquely identifies the system. It is used to allow devices to determine which links go to which other devices.

Before MP can be used, a successful negotiation of at least the Multilink Maximum Received Reconstructed Unit option must be performed on each of the links between the two devices. Once this is done and an LCP link exists for each of the physical links, a virtual *bundle* is made of the LCP links, and MP is enabled.

PPP Multilink Protocol Operation

As mentioned previously, MP basically sits between the network layer and the regular PPP links and acts as a middleman. Here is what it does for each direction of communication:

Transmission MP accepts datagrams received from any of the network layer protocols configured using appropriate NCPs. It first encapsulates them into a modified version of the regular PPP frame, and then takes that frame and decides how to transmit it over the multiple physical links. Typically, this is done by dividing the frame into *fragments* that are evenly spread out over the set of links. These are then encapsulated and sent over the physical links. However, you can also implement an alternative strategy as well, such as alternating full-sized frames between the links. Also, smaller frames typically aren't fragmented, and neither are control frames such as the ones used for link configuration.

NOTE IP datagrams are sometimes called IP packets. Whether datagram or packet is the preferred term seems to depend on whom you ask; even the standards don't use one term exclusively. On the other hand, I have seen IP datagrams called IP frames, and that's definitely not correct. Chapter 1 describes these terms more completely.

IP Datagram Encapsulation

In Chapter 5, which described OSI Reference Model concepts, I looked at ways that protocols at various layers in a networking protocol stack interact with each other. One of the most important concepts in interprotocol operation is the concept of *encapsulation*. Most data originates within the higher layers of the OSI model. The protocols at these layers pass the data down to lower layers for transmission, usually in the form of discrete messages. Upon receipt, each lower-level protocol takes the entire contents of the message received and encapsulates it into its own message format, adding a header and possibly a footer that contain important control information.

You might think of encapsulation as similar to sending a letter enclosed in an envelope. You write a letter and put it in an envelope with a name and address. If you give it to a courier for overnight delivery, the courier takes that envelope and puts it in a larger delivery envelope. In a similar way, messages at higher network layers are encapsulated in lower-layer messages, which can then in turn be further encapsulated.

Due to the prominence of TCP/IP, IP is one of the most important places where data encapsulation occurs on a modern network. Data is passed to IP typically from one of the two main transport layer protocols: the Transmission Control Protocol (TCP) or User Datagram Protocol (UDP). This data is already in the form of a TCP or UDP message with TCP or UDP headers. This is then encapsulated into the body of an IP message, usually called an *IP datagram* or *packet*. Encapsulation and formatting of an IP datagram is also sometimes called *packetizing*—again, the envelope is an obvious comparison.

Figure 21-1 displays this entire process, which looks very similar to the data of the OSI Reference Model as a whole, as shown in Figure 5-5 in Chapter 5. You can see, an upper-layer message is packaged into a TCP or UDP message. This becomes the payload of an IP datagram, shown here with only one header (but can get a bit more complex than this). The IP datagram is then passed down to layer 2, where it is encapsulated into some sort of local area network (LAN) or area network (WAN), or wireless LAN (WLAN) frame, and then converted and transmitted at the physical layer.

If the message to be transmitted is too large to pass through the underlying network, it may first be fragmented. This is analogous to splitting up a large delivery into multiple smaller envelopes or boxes. In this case, each IP datagram carries only part of the higher-layer message. The receiving device must reassemble the message from the IP datagrams.

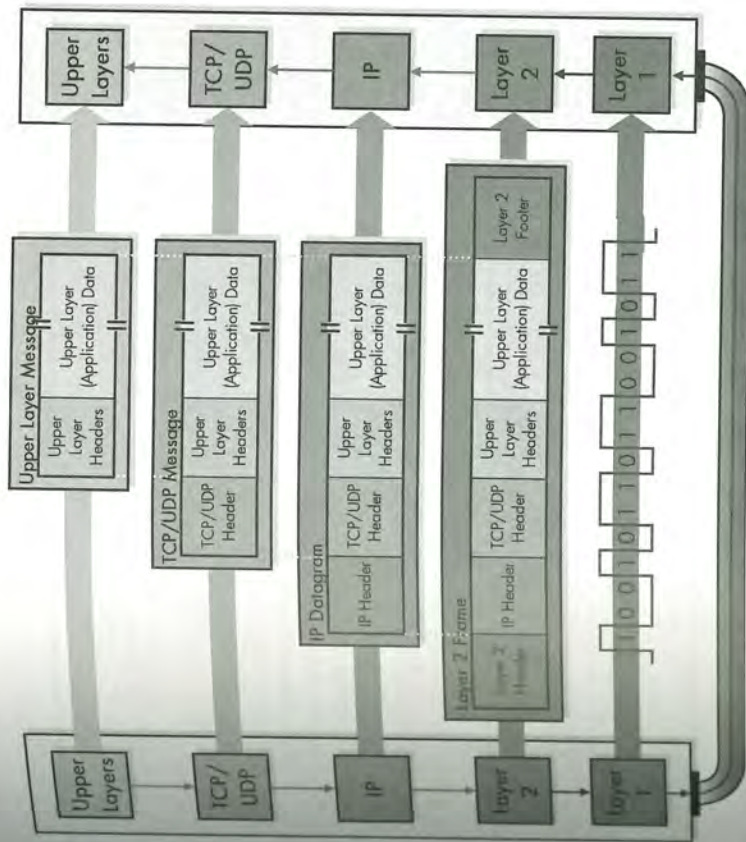


Figure 21-1: IP datagram encapsulation The upper-layer message is packaged into a TCP or UDP message, which becomes the payload of an IP datagram. The IP datagram is then passed down to layer 2, where it is encapsulated in a LAN, WAN, or WLAN frame. It is then converted to bits and transmitted at the physical layer.

The IP datagram is somewhat similar in concept to a frame used in Ethernet or another data link layer, except that IP datagrams are designed to facilitate transmission across an internetwork, while data link layer frames are used only for direct delivery within a physical network. The fields included in the IP header are used to manage internetwork datagram delivery. This includes key information for delivery, such as the address of the destination device, identification of the type of frame, and control bits. The header follows a format that you will examine shortly.

Once data is encapsulated into an IP datagram, it is passed down to the data link layer for transmission across the current "hop" of the internetwork. There it is further encapsulated, IP header and all, into a data link layer frame such as an Ethernet frame. An IP datagram may be encapsulated into many such data link layer frames as it is routed across the internetwork; on each hop, the IP datagram is removed from the data link layer frame and then repackaged into a new one for the next hop. The IP datagram, however, is not changed (except for some control fields) until it reaches its final destination.

IP Datagram General Format

Data transmitted over an internetwork using IP is carried in messages called *IP datagrams*. As is the case with all network protocol messages, IP uses a specific format for its datagrams. Here, I will discuss the IP version 4 (IPv4) datagram format, which was defined in RFC 791 along with the rest of IPv4.

The IPv4 datagram is conceptually divided into two pieces: the *header* and the *payload*. The header contains addressing and control fields, while the payload carries the actual data to be sent over the internetwork. Unlike some message formats, IP datagrams do not have a footer following the payload.

Even though IP is a relatively simple, connectionless, and unreliable protocol, the IPv4 header carries a fair bit of information, which makes it rather large. It is at least 20 bytes long, and with options it can be significantly longer. The IP datagram format is described in Tables 21-1, 21-2, and 21-3, and illustrated in Figure 21-2.

Table 21-1: Internet Protocol Version 4 (IPv4) Datagram Format

Field Name	Size (Bytes)	Description
Version	1/2 (4 bits)	Identifies the version of IP used to generate the datagram. For IPv4, this is the number 4. This field ensures compatibility between devices that may be running different versions of IP. In general, a device running an older version of IP will reject datagrams created by newer implementations, under the assumption that the older version may not be able to interpret newer datagram correctly.
IHL	1/2 (4 bits)	Specifies the length of the IP header, in 32-bit words. This includes the length of any options fields and padding. The normal value of this field when no options are used is 5 (5 32-bit words = 5 * 4 = 20 bytes). Contrast this with the longer Total Length field in this table.
TOS	1	A field designed to carry information to provide quality-of-service features, such as priority delivery for IP datagrams. This has not been as widely used as originally defined, and its meaning has been redefined for use by a technique called <i>Differentiated Services (DS)</i> , as discussed in the "IP Datagram Type of Service (TOS) Field" section of this chapter.
TTL	2	Specifies the total length of the IP datagram, in bytes. Since this field is 16 bits wide, the maximum length of an IP datagram is 65,535 bytes, though most are much smaller.
Identification	2	This field contains a 16-bit value that is common to each of the fragments belonging to a particular message; for datagrams originally sent unfragmented, it is still filled in so it can be used if the datagram must be fragmented by a router during delivery. The recipient uses this field to reassemble messages without accidentally mixing fragments from different messages. This is needed because fragments may arrive from multiple messages mixed together, and datagrams can be received out of order from any device. (See the discussion of IP message fragmentation in Chapter 22.)
Flags	3/8 (3 bits)	Three control flags, two of which are used to manage fragmentation (as described in the section on fragmentation), and one that is reserved. See Table 21-2.
Fragment Offset	1 5/8 (13 bits)	When fragmentation of a message occurs, this field specifies the offset, or position, in the message where the data in this fragment goes in units of eight bytes (64 bits). The first fragment has an offset of 0. (See the discussion of fragmentation in Chapter 27 for a description of how the field is used.)
TTL	1	This specifies how long the datagram is allowed to live on the network, in router hops. Each router decrements the value of the TTL field (reduces it by one) prior to transmitting it. If the field drops to zero, the datagram is assumed to have taken too long a route and is discarded. (See the "IP Datagram Time to Live (TTL) Field" section later in this chapter for more information.)

Table 21-1: Internet Protocol Version 4 (IPv4) Datagram Format (continued)

Field Name	Size (Bytes)	Description
Protocol	1	Identifies the higher-layer protocol (generally either a transport layer protocol or encapsulated network layer protocol) carried in the datagram. Table 21-3 shows the protocol values of this field, which were originally defined by the IETF "Assigned Numbers" standard, RFC 1700, and are now maintained by the Internet Assigned Numbers Authority (IANA).
Header Checksum	2	A checksum is computed over the header to provide basic protection against corruption in transmission. This is not the more complex cyclic redundancy check (CRC) code that's typically used by data link layer technologies such as Ethernet; it's just a 16-bit checksum. It is calculated by dividing the header bytes into words (a word is two bytes) and then adding them together. Only the header is checksummed; not the data. At each hop, the device receiving the datagram does the same checksum calculation, and if there is a mismatch, it discards the datagram as damaged.
Source Address	4	This is the 32-bit IP address of the originator of the datagram. Note that even though intermediate devices such as routers may handle the datagram, they do not normally put their address into this field—the address is always that of the device that originally sent the datagram.
Destination Address	4	This is the 32-bit IP address of the intended recipient of the datagram. Again, even though devices such as routers may be the intermediate targets of the datagram, this field is always used to specify the ultimate destination.
Options	Variable	One or more of several types of options may be included after the standard headers in certain IP datagrams, as discussed later in this chapter, in the "IP Datagram Options and Option Format" section.
Padding	Variable	If one or more options are included, and the number of bits used for them is not a multiple of 32, enough 0 bits are added to pad out the header to a multiple of 32 bits (four bytes).
Data	Variable	This is the data that will be transmitted in the datagram. It is either an entire higher-layer message or a fragment of one.

Table 21-2: IPv4 Flags Subfields

Subfield Name	Size (Bytes)	Description
Reserved	1/8 (1 bit)	Not used.
DF	1/8 (1 bit)	When set to 1, this says that the datagram should not be fragmented. Since the fragmentation process is generally invisible to higher layers, most protocols don't care about this and don't set this flag. It is, however, used for testing the maximum transmission unit (MTU) of a link.
MF	1/8 (1 bit)	When set to 0, this indicates the last fragment in a message; when set to 1, it indicates that more fragments are yet to come in the fragmented message. If no fragmentation is used for a message, there is only one fragment (the whole message), and this flag is 0. If fragmentation is used, all fragments but the last set this flag to 1 so that the recipient knows when all fragments have been sent.

NOTE IP datagrams are sometimes called IP packets. Whether datagram or packet is the preferred term seems to depend on whom you ask; even the standards don't use one term exclusively. On the other hand, I have seen IP datagrams called IP frames, and that's definitely not correct. Chapter 1 describes these terms more completely.

IP Datagram Encapsulation

In Chapter 5, which described OSI Reference Model concepts, I looked at several ways that protocols at various layers in a networking protocol stack interact with each other. One of the most important concepts in interprotocol operation is the concept of *encapsulation*. Most data originates within the higher layers of the OSI model. The protocols at these layers pass the data down to lower layers for transmission, usually in the form of discrete messages. Upon receipt, each lower-level protocol takes the entire contents of the message received and encapsulates it into its own message format, adding a header and possibly a footer that contain important control information.

You might think of encapsulation as similar to sending a letter enclosed in an envelope. You write a letter and put it in an envelope with a name and address; if you give it to a courier for overnight delivery, the courier takes that envelope and puts it in a larger delivery envelope. In a similar way, messages at higher network layers are encapsulated in lower-layer messages, which can then in turn be further encapsulated.

Due to the prominence of TCP/IP, IP is one of the most important places where data encapsulation occurs on a modern network. Data is passed to IP typically from one of the two main transport layer protocols: the Transmission Control Protocol (TCP) or User Datagram Protocol (UDP). This data is already in the form of a TCP or UDP message with TCP or UDP headers. This is then encapsulated into the body of an IP message, usually called an *IP datagram* or *packet*. Encapsulation and formatting of an IP datagram is also sometimes called *packaging*—again, the envelope is an obvious comparison.

Figure 21-1 displays this entire process, which looks very similar to the diagram of the OSI Reference Model as a whole, as shown in Figure 5-5 in Chapter 5. As you can see, an upper-layer message is packaged into a TCP or UDP message. This becomes the payload of an IP datagram, shown here with only one header (this can get a bit more complex than this). The IP datagram is then passed down to layer 2, where it is encapsulated into some sort of local area network (LAN), wide area network (WAN), or wireless LAN (WLAN) frame, and then converted to bits and transmitted at the physical layer.

If the message to be transmitted is too large to pass through the underlying network, it may first be fragmented. This is analogous to splitting up a large delivery into multiple smaller envelopes or boxes. In this case, each IP datagram carries only part of the higher-layer message. The receiving device must reassemble the message from the IP datagrams.

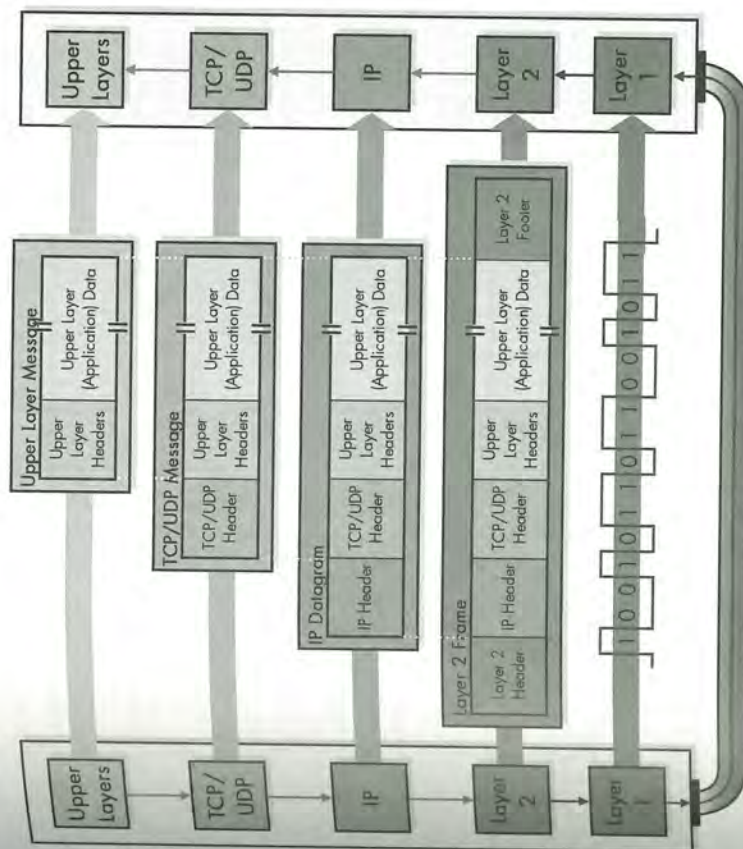


Figure 21-1: IP datagram encapsulation The upper-layer message is packaged into a TCP or UDP message, which becomes the payload of an IP datagram. The IP datagram is then passed down to layer 2, where it is encapsulated in a LAN, WAN, or WLAN frame. It is then converted to bits and transmitted at the physical layer.

The IP datagram is somewhat similar in concept to a frame used in Ethernet or another data link layer, except that IP datagrams are designed to facilitate transmission across an internetwork, while data link layer frames are used only for direct delivery within a physical network. The fields included in the IP header are used to manage internetwork datagram delivery. This includes key information for delivery, such as the address of the destination device, identification of the type of frame, and control bits. The header follows a format that you will examine shortly.

Once data is encapsulated into an IP datagram, it is passed down to the data link layer for transmission across the current "hop" of the internetwork. There it is further encapsulated, IP header and all, into a data link layer frame such as an Ethernet frame. An IP datagram may be encapsulated into many such data link layer frames as it is routed across the internetwork; on each hop, the IP datagram is removed from the data link layer frame and then repackaged into a new one for the next hop. The IP datagram, however, is not changed (except for some control fields) until it reaches its final destination.

IPsec Modes: Transport and Tunnel

You just saw that three different basic implementation architectures could be used to provide IPsec facilities to TCP/IP networks. The choice of which implementation you use, as well as whether you implement in end hosts or routers, impacts the specific way that IPsec functions. Two specific modes of operation that are related to these architectures are defined for IPsec. They are called *transport mode* and *tunnel mode*.

IPsec modes are closely related to the function of the two core protocols, AH and ESP. Both of these protocols provide protection by adding a header (and possibly other fields) containing security information to a datagram. The choice of mode does not affect the method by which each generates its header, but rather, changes what specific parts of the IP datagram are protected and how the headers are arranged to accomplish this. In essence, the mode really describes, not prescribes, how AH or ESP do their thing. It is used as the basis for defining other constructs, such as security associations (SAs).

Transport Mode

As its name suggests, in transport mode, the protocol protects the message passed down to IP from the transport layer. The message is processed by AH and/or ESP, and the appropriate header(s) are added in front of the transport (UDP or TCP) header. The IP header is then added in front of that by IP.

Another way of looking at this is as follows: Normally, the transport layer packages data for transmission and sends it to IP. From IP's perspective, this transport layer message is the payload of the IP datagram. When IPsec is used in transport mode, the IPsec header is applied only over this IP payload, not the IP header. The AH and ESP headers appear between the original, single IP header and the IP payload. This is illustrated in Figure 29-4.

Tunnel Mode

In tunnel mode, IPsec is used to protect a completely encapsulated IP datagram after the IP header has already been applied to it. The IPsec headers appear in front of the original IP header, and then a new IP header is added in front of the IPsec header. That is to say, the entire original IP datagram is secured and then encapsulated within another IP datagram. This is shown in Figure 29-5.

Comparing Transport and Tunnel Modes

The bottom line in understanding the difference between the two IPsec modes is this: Tunnel mode protects the original IP datagram as a whole, header and all, while transport mode does not. Thus, in general terms, the order of the headers is as follows:

Transport Mode IP header, IPsec headers (AH and/or ESP), IP payload (including transport header)

Tunnel Mode New IP header, IPsec headers (AH and/or ESP), old IP header, IP payload

The advantage of this technique is that IPsec can be retrofitted to any IP device, since the IPsec functionality is separate from IP. The disadvantage is that there is a duplication of effort compared to the integrated architecture. BITS is generally used for IPv4 hosts.

Bump in the Wire (BITW) Architecture

In the bump in the wire (BITW) method, we add a hardware device that provides IPsec services. For example, suppose we have a company with two sites. Each has a network that connects to the Internet using a router that is not capable of IPsec functions. We can interpose a special IPsec device between the router and the Internet at both sites, as shown in Figure 29-3. These devices will then intercept outgoing datagrams, add IPsec protection to them, and strip it off incoming datagrams.

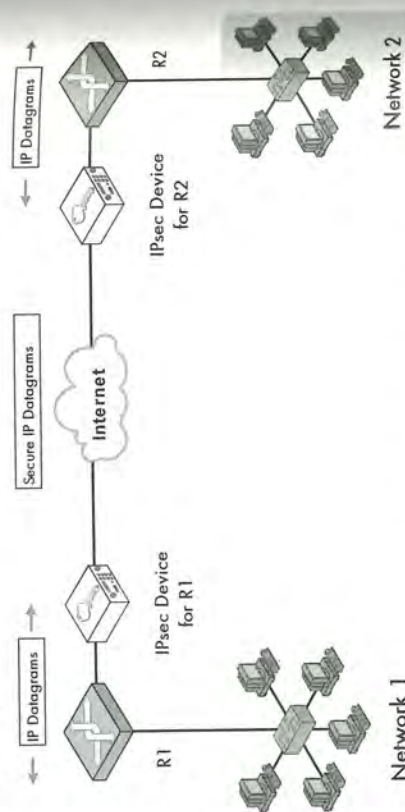


Figure 29-3: IPsec architecture In this IPsec architecture, IPsec is actually implemented in separate devices that sit between the devices that wish to communicate securely. These devices add IPsec protection to outgoing datagrams for transport over the public Internet.

Just as BITS lets you add IPsec to legacy hosts, BITW can retrofit non-IPsec routers to provide security benefits. The disadvantages are complexity and cost.

KEY CONCEPT Three different architectures or implementation models are defined for IPsec. The best is integrated architecture, in which IPsec is built into the IP layer of devices directly. Other two are bump in the stack (BITS) and bump in the wire (BITW), which are ways of layering IPsec underneath regular IP, using software and hardware solutions, respectively.

As you will see in the next section, the choice of architecture has an important impact on which of the two IPsec modes can be used. Incidentally, even though BITS and BITW seem quite different, they are actually do the same thing. In the case of BITS, we have an extra software layer that adds security to existing IP datagrams; in BITW, distinct hardware devices do this same job. In both cases, the result is the same, and the implications on the choice of IPsec mode is likewise the same.

IPsec Modes: Transport and Tunnel

You just saw that three different basic implementation architectures could be used to provide IPsec facilities to TCP/IP networks. The choice of which implementation you use, as well as whether you implement in end hosts or routers, impacts the specific way that IPsec functions. Two specific modes of operation that are related to these architectures are defined for IPsec. They are called *transport mode* and *tunnel mode*.

IPsec modes are closely related to the function of the two core protocols, AH and ESP. Both of these protocols provide protection by adding a header (and possibly other fields) containing security information to a datagram. The choice of mode does not affect the method by which each generates its header, but rather, changes what specific parts of the IP datagram are protected and how the headers are arranged to accomplish this. In essence, the mode really describes, not prescribes, how AH or ESP do their thing. It is used as the basis for defining other constructs, such as security associations (SAs).

Transport Mode

As its name suggests, in transport mode, the protocol protects the message passed down to IP from the transport layer. The message is processed by AH and/or ESP, and the appropriate header(s) are added in front of the transport (UDP or TCP) header. The IP header is then added in front of that by IP.

Another way of looking at this is as follows: Normally, the transport layer packages data for transmission and sends it to IP. From IP's perspective, this transport layer message is the payload of the IP datagram. When IPsec is used in transport mode, the IPsec header is applied only over this IP payload, not the IP header. The AH and ESP headers appear between the original, single IP header and the IP payload. This is illustrated in Figure 29-4.

Tunnel Mode

In tunnel mode, IPsec is used to protect a completely encapsulated IP datagram after the IP header has already been applied to it. The IPsec headers appear in front of the original IP header, and then a new IP header is added in front of the IPsec header. That is to say, the entire original IP datagram is secured and then encapsulated within another IP datagram. This is shown in Figure 29-5.

Comparing Transport and Tunnel Modes

The bottom line in understanding the difference between the two IPsec modes is this: Tunnel mode protects the original IP datagram as a whole, header and all, while transport mode does not. Thus, in general terms, the order of the headers is as follows:

Transport Mode IP header, IPsec headers (AH and/or ESP), IP payload (including transport header)

Tunnel Mode New IP header, IPsec headers (AH and/or ESP), old IP header, IP payload

The size of the Authentication Data field is variable to support different datagram lengths and hashing algorithms. Its total length must be a multiple of 4 bits. Also, the entire header must be a multiple of either 32 bits (for IPv4) or 64 bits (for IPv6), so additional padding may be added to the Authentication Data field if necessary.

You may also notice that no IP addresses appear in the header, which is a prerequisite for it being the same for both IPv4 and IPv6.

IPsec Encapsulating Security Payload (ESP)

The IPsec AH provides integrity authentication services to IPsec-capable devices so that they can verify that messages are received intact from other devices. For many applications, however, this is only one piece of the puzzle. We want to not only protect against intermediate devices changing the datagrams, but also to protect against them examining their contents as well. For this level of private communication, AH is not enough; we need to use the ESP protocol.

The main job of ESP is to provide the privacy we seek for IP datagrams by encrypting them. An encryption algorithm combines the data in the datagram with a key to transform it into an encrypted form. This is then repackaged using a special format that you will see shortly, and then transmitted to the destination, which decrypts it using the same algorithm. ESP also sports its own authentication scheme like the one used in AH, or it can be used in conjunction with AH.

ESP Fields

ESP has several fields that are the same as those used in AH, but it packages its fields in a very different way. Instead of having just a header, it divides its fields into three components:

ESP Header This contains two fields, SPI and Sequence Number, and comes before the encrypted data. Its placement depends on whether ESP is used in transport mode or tunnel mode, as explained earlier in this chapter.

ESP Trailer This section is placed after the encrypted data. It contains padding that is used to align the encrypted data through a Padding and Pad Length field. Interestingly, it also contains the Next Header field for ESP.

ESP Authentication Data This field contains an ICV that's computed in a manner that's similar to how the AH protocol works. The field is used when ESP's optional authentication feature is employed.

There are two reasons why these fields are broken into pieces like this. The first is that some encryption algorithms require the data to be encrypted to have a certain block size, and so padding must appear after the data and not before it. That's why padding appears in the ESP Trailer field. The second is that the ESP Authentication Data appears separately because it is used to authenticate the rest of the encrypted datagram after encryption. This means that it cannot appear in the ESP Header or ESP Trailer.

ESP Operations and Field Use

This is still a bit boggling so I'm going to try to explain this procedurally by considering three basic steps performed by ESP: calculating the header, then the trailer, and then the Authentication field.

Header Calculation and Placement

The first thing to consider is how the ESP header is placed. This is similar to how AH works and depends on the IP version, as follows:

IPv6 The ESP Header field is inserted into the IP datagram as an extension header, following the normal IPv6 rules for extension-header linking. In transport mode, it appears before a Destination Options header that contains options intended for the final destination, but after any other extension headers, if present. In tunnel mode, it appears as an extension header of the new IP datagram that encapsulates the original one being tunneled. This is shown in Figure 29-9.

IPv4 As with AH, the ESP Header field is placed after the normal IPv4 header. In transport mode, it appears after the IP header of the original datagram; in tunnel mode, it appears after the IP header of the new IP datagram that's encapsulating the original one. You can see this in Figure 29-10.

Trailer Calculation and Placement

The ESP Trailer field is appended to the data that will be encrypted. ESP then performs the encryption. The payload (TCP/UDP message or encapsulated IP datagram) and the ESP trailer are both encrypted, but the ESP header is not. Note again that any other IP headers that appear between the ESP header and the payload are also encrypted. In IPv6, this can include a Destination Options extension header.

Normally, the Next Header field would appear in the ESP Header and would be used to link the ESP Header to the header that comes after it. However, the Next Header field in ESP appears in the trailer and not the header, which makes the linking seem a bit strange in ESP. The method is basically the same as what's used in AH and in IPv6 in general, with the Next Header and Protocol fields being used to tie everything together. However, in ESP the Next Header field appears after the encrypted data, and so it points back to one of the following: a Destination Options extension header (if present), a TCP/UDP header (in transport mode), or an IPv4/IPv6 header (in tunnel mode). This is also shown in Figures 29-9 and 29-10.

ESP Authentication Field Calculation and Placement

If the optional ESP authentication feature is being used, it is computed over the entire ESP datagram (except the Authentication Data field itself, of course). This includes the ESP header, payload, and trailer.

KEY CONCEPT The IPsec ESP protocol allows the contents of a datagram to be encrypted, which ensures that only the intended recipient is able to see the data. ESP is implemented using three components: an ESP Header that's added to the front of a protected datagram, an ESP Trailer that follows the protected data, and an optional ESP Authentication Data field that provides authentication services similar to those provided by AH.

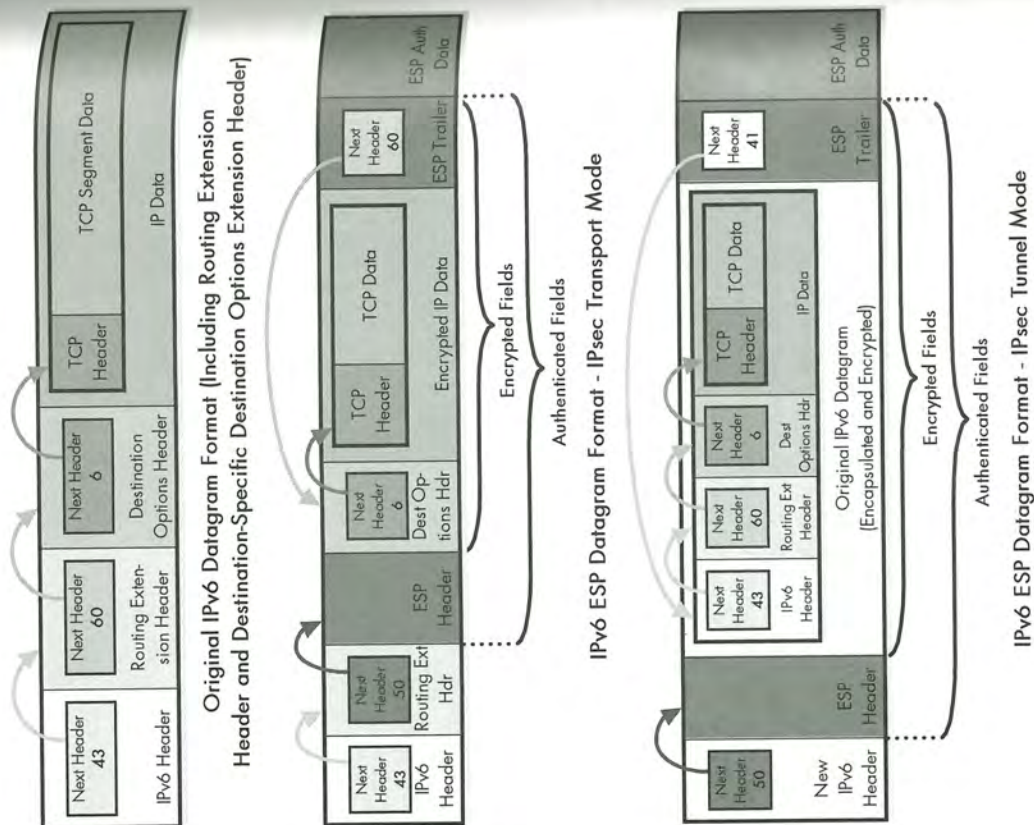


Figure 29-9: IPv6 datagram format with IPsec ESP Here is the same example of an IPv6 datagram with two extension headers that you saw in Figure 29-6. When ESP is applied in transport mode, the ESP Header field is added to the existing datagram as in AH, and the ESP Trailer and ESP Authentication Data fields are placed at the end. In tunnel mode, the ESP Header and Trailer fields bracket the entire encapsulated IPv6 datagram. Note the encryption and authentication coverage in each case, and also how the Next Header field points back into the datagram since it appears in the ESP Trailer.

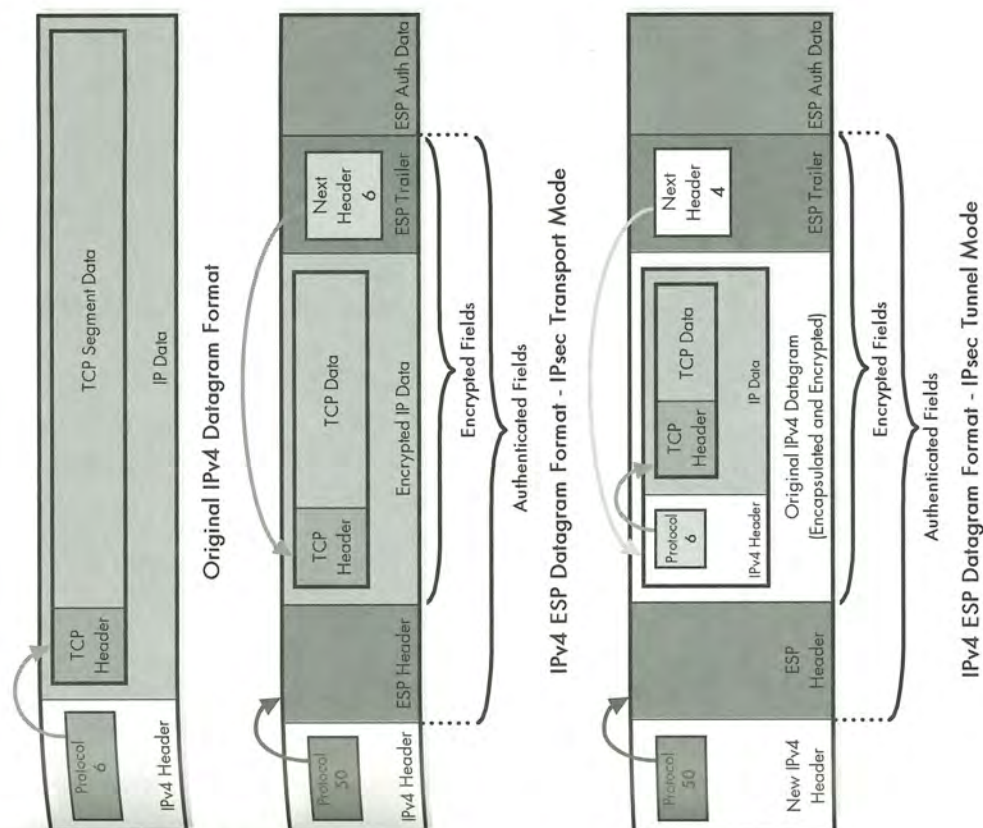


Figure 29-10: IPv4 datagram format with IPsec ESP Here is the same sample IPv4 datagram that you saw in Figure 29-7. When ESP processes this datagram in transport mode, the ESP Header field is placed between the IPv4 header and data, with the ESP Trailer and ESP Authentication Data fields following. In tunnel mode, the entire original IPv4 datagram is surrounded by these ESP components, rather than just the IPv4 data. Again, as in Figure 29-9, note the encryption and authentication coverage, and how the Next Header field points back to specify the identity of the encrypted data or datagram.

ESP Format

The format of the ESP sections and fields is described in Table 29-3 and illustrated in Figure 29-11. In both the figure and the table, I have shown the encryption and authentication coverage of the fields explicitly, to clarify how it all works.

Table 29-3: IPsec Encapsulating Security Payload (ESP) Format

Section	Field Name	Size (Bytes)	Description	Encryption Coverage	Authentication Coverage
ESP Header	SPI	4	A 32-bit value that is combined with the destination address and security protocol type to identify the SA that will be used for this datagram. (SAs are discussed earlier in this chapter.)		
	Sequence Number	4	A counter field initialized to zero when an SA is formed between two devices, and then incremented for each datagram that's sent using that SA. This is used to provide protection against replay attacks.		
Payload	Payload Data	Variable	The encrypted payload data, which consists of a higher-layer message or encapsulated IP datagram. It may also include support information such as an initialization vector that's required by certain encryption methods.		
ESP Trailer	Padding	Variable (0 to 255)	Additional padding bytes are included as needed for encryption or for alignment.		
	Pad Length	1	The number of bytes in the preceding Padding field.		
	Next Header	1	Contains the protocol number of the next header in the datagram. Used to chain together headers.		
ESP Authentication Data		Variable	Contains the ICV resulting from the application of the optional ESP authentication algorithm.		

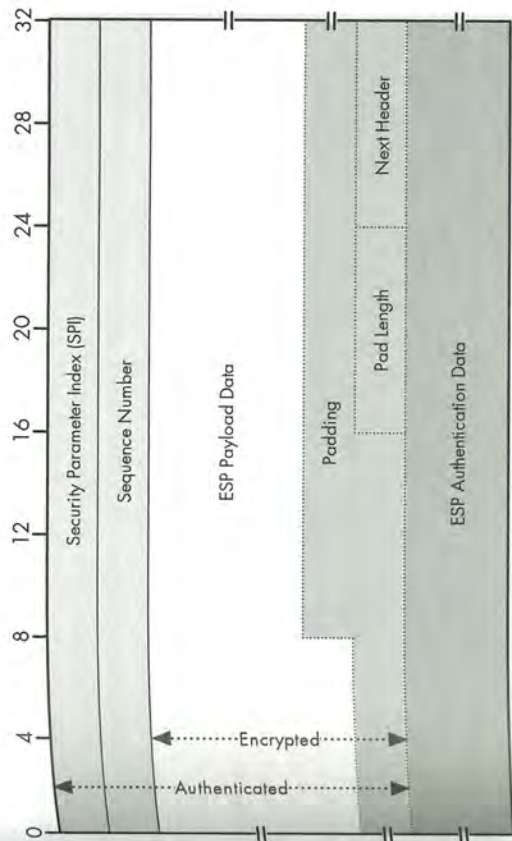


Figure 29-11: IPsec ESP format. Note that most of the fields and sections in this format are variable length. The exceptions are the SPI and Sequence Number fields, which are four bytes long, and the Pad Length and Next Header fields, which are one byte each.

The Padding field is used when encryption algorithms require it. Padding is also used to make sure that the ESP Trailer field ends on a 32-bit boundary. That is, the size of the ESP Header field plus the Payload field, plus the ESP Trailer field must be a multiple of 32 bits. The ESP Authentication Data field must also be a multiple of 32 bits.

IPsec Internet Key Exchange (IKE)

IPsec, like many secure networking protocol sets, is based on the concept of a shared secret. Two devices that want to send information securely encode and decode it using a piece of information that only the devices know. Anyone who isn't in on the secret is able to intercept the information but is prevented either from reading it (if ESP is used to encrypt the payload) or from tampering with it undetected (if AH is used). Before either AH or ESP can be used, however, it is necessary for the two devices to exchange the secret that the security protocols themselves will use. The primary support protocol used for this purpose in IPsec is called *Internet Key Exchange (IKE)*.

IKE is defined in RFC 2409, and it is one of the more complicated of the IPsec protocols to comprehend. In fact, it is simply impossible to truly understand more than a real simplification of its operation without significant background in cryptography. I don't have a background in cryptography, and I must assume that you, my reader, do not either. So rather than fill this topic with baffling acronyms and unexplained concepts, I will just provide a brief outline of IKE and how it is used.

Table 43-1: Common TCP/IP Well-Known Port Numbers and Applications (continued)

Port #	TCP/UDP	Keyword	Protocol Abbreviation	Application or Protocol Name/Comments
69	UDP	lftp	TFTP	Trivial File Transfer Protocol
70	TCP	gopher	—	Gopher Protocol
79	TCP	finger	—	Finger User Information Protocol
80	TCP	http	HTTP	Hypertext Transfer Protocol (World Wide Web)
110	TCP	pop3	POP	Post Office Protocol (version 3)
119	TCP	nntp	NNTP	Network News Transfer Protocol
123	UDP	ntp	NTP	Network Time Protocol
137	TCP + UDP	netbios-ns	—	NetBIOS (Name Service)
138	UDP	netbios-dgm	—	NetBIOS (Datagram Service)
139	TCP	netbios-ssn	—	NetBIOS (Session Service)
143	TCP	imap	IMAP	Internet Message Access Protocol
161	UDP	snmp	SNMP	Simple Network Management Protocol
162	UDP	snmptrap	SNMP	Simple Network Management Protocol (Trap)
179	TCP	bgp	BGP	Border Gateway Protocol
194	TCP	irc	IRC	Internet Relay Chat
443	TCP	https	HTTP over SSL	Hypertext Transfer Protocol over Secure Sockets Layer
500	UDP	isakmp	IKE	IPsec Internet Key Exchange
520	UDP	router	RIP	Routing Information Protocol (RIP-1 and RIP-2)
521	UDP	ripng	RIPng	Routing Information Protocol - Next Generation

The registered port numbers are by definition for protocols that are not standardized using the RFC process, so they are mostly esoteric applications, and I don't think it's necessary to list all of them. Table 43-2 shows a few that I feel are of particular interest.

Table 43-2: Common TCP/IP Registered Port Numbers and Applications

Port #	TCP/UDP	Keyword	Protocol Abbreviation	Application or Protocol Name/Comments
1512	TCP + UDP	wins	WINS	Microsoft Windows Internet Naming Service
1701	UDP	l2tp	L2TP	Layer 2 Tunneling Protocol
1723	TCP	pptp	PPTP	Point-to-Point Tunneling Protocol
2049	TCP + UDP	nfs	NFS	Network File System
6000-6063	TCP	x11	X11	X Window System

This document is the definitive list of all well-known and registered TCP and UDP port assignments. Each port number is assigned a short *keyword* with a brief description of the protocol that uses it. There are two problems with this document. First, it is incredibly long; it contains over 10,000 lines of text. Most of the protocols mentioned in those thousands of lines are for obscure applications that you have probably never heard of before (I certainly have never heard of most of them!). This makes it hard to easily see the port assignments for the protocols that are most commonly used.

The other problem with this document is that it shows the same port number as reserved for both TCP and UDP for an application. As I mentioned earlier, TCP and UDP port numbers are actually independent, so, in theory, one port number could assign TCP port 80 to one server application type and UDP port 80 to another. It was believed that this would lead to confusion, so with very few exceptions, the same port number is shown in the list for the same application for both TCP and UDP. Nevertheless, showing this in the list has a drawback: You can't tell which protocol the application actually uses, and which has just been reserved for consistency.

Given all that, I have decided to include a couple of summary tables that show the well-known and registered port numbers for the most common TCP/IP applications. I have indicated whether or not the protocol uses TCP, UDP, or both. Table 43-1 lists the well-known port numbers for the most common TCP/IP application protocols.

Table 43-1: Common TCP/IP Well-Known Port Numbers and Applications

Port #	TCP/UDP	Keyword	Protocol Abbreviation	Application or Protocol Name/Comments
7	TCP + UDP	echo	—	Echo Protocol
9	TCP + UDP	discard	—	Discard Protocol
11	TCP + UDP	sysstat	—	Active Users Protocol
13	TCP + UDP	daytime	—	Daytime Protocol
17	TCP + UDP	qotd	QOTD	Quote of the Day Protocol
19	TCP + UDP	chargen	—	Character Generator Protocol
20	TCP	ftp-data	FTP (data)	File Transfer Protocol (default data port)
21	TCP	ftp	FTP (control)	File Transfer Protocol (control/command)
23	TCP	telnet	—	Telnet Protocol
25	TCP	smtp	SMTP	Simple Mail Transfer Protocol
37	TCP + UDP	time	—	Time Protocol
43	TCP	nickname	—	Whois Protocol (also called Nickname)
53	TCP + UDP	domain	DNS	Domain Name Server (Domain Name System)
67	UDP	bootps	BOOTP/DHCP	Bootstrap Protocol/Dynamic Host Configuration Protocol (server)
68	UDP	bootpc	BOOTP/DHCP	Bootstrap Protocol/Dynamic Host Configuration Protocol (client)

(continued)

age. A precocious protocol, you might say. (To be fair, other standards have described many additional features and modifications to TCP, rather than upgrading the main document.)

Overview of TCP Operation

TCP is a full-featured transport layer protocol that provides all the functions that a typical application needs for the reliable transportation of data across an arbitrary internetwork. It provides transport layer addressing for application processes in the form of TCP ports and allows machines to use these ports in order to establishing connections between them. Once the devices have connected to each other, they can pass data bidirectionally between them. Applications can send data to TCP as a simple stream of bytes, and TCP takes care of packaging and sending the data as segments that TCP packages into IP datagrams. The receiving device's TCP implementation reverses the process, passing up to the application the stream of data that the device originally sent.

TCP includes an extensive set of mechanisms. These mechanisms ensure that data gets from source to destination reliably, consistently, and in a timely fashion. The key to its operation in this regard is the *sliding window acknowledgment system*, which allows each device to keep track of the bytes of data it has sent and to confirm the receipt of data received from the other device in the connection. Unacknowledged data is eventually retransmitted automatically, and the parameters of the system can be adjusted to the needs of the devices and the connection. This same system also provides buffering and flow control capabilities between devices. These capabilities handle uneven data delivery rates and other problems.

KEY CONCEPT The primary transport layer protocol in the TCP/IP protocol suite is the *Transmission Control Protocol (TCP)*. TCP is a connection-oriented, acknowledged, reliable, full-featured protocol designed to provide applications with a reliable way to send data using the unreliable Internet Protocol (IP). It allows applications to send bytes of data as a stream of bytes and automatically packages them into appropriately sized segments for transmission. It uses a special *sliding window acknowledgment system* to ensure that its recipient receives all data, handles necessary retransmissions, and provides flow control so that each device in a connection can manage the rate at which other devices send data to it.

Because of TCP's many capabilities, it's likely that the protocol will satisfy just about any application that requires reliable, connection-oriented data delivery. A primary goal of TCP, reliable data delivery means that higher-layer applications don't need to provide TCP's common functions. Because the majority of conventional message-passing applications employ it, the TCP/IP transport protocol is the most widely used transport protocol.

TCP Standards

RFC 793 is the defining standard for TCP, but it doesn't include all the details about how modern TCP operates. Several other standards include additional information about how the protocol works and describe enhancements to the basic

TCP mechanisms that were developed over the years. Some of these are fairly esoteric, but they are useful for gaining a more complete understanding of TCP. I have listed some of them in Table 45-1.

Table 45-1: Supplementary TCP Standards

RFC #	Name	Description
813	Window and Acknowledgment Strategy in TCP	Discusses the TCP sliding window acknowledgment system, describes certain problems that can occur with it, and offers methods to correct them.
879	The TCP Maximum Segment Size and Related Topics	Discusses the important maximum segment size (MSS) parameter that controls the size of TCP messages, and then relates this parameter to the datagram size.
896	Congestion Control in IP/TCP Internetworks	Talks about congestion problems and how you can use TCP to handle them. Note the interesting inversion of the normal protocol suite name IP/TCP.
1122	Requirements for Internet Hosts — Communication Layers	Describes important details of how TCP should be implemented on hosts.
1146	TCP Alternate Checksum Options	Specifies a mechanism for having TCP devices use an alternative method of checksum generation.
1323	TCP Extensions for High Performance	Defines extensions to TCP for high-speed links and new TCP options.
2018	TCP Selective Acknowledgment Options	An enhancement to basic TCP functionality that allows TCP devices to selectively specify specific segments for retransmission.
2581	TCP Congestion Control	Describes four algorithms used for congestion control in TCP networks: slow start, congestion avoidance, fast retransmit, and fast recovery.
2988	Computing TCP's Retransmission Timer	Discusses issues related to setting the TCP retransmission timer, which controls how long a device waits for acknowledgment of sent data before retransmitting it.

There are hundreds of higher-layer application protocols that use TCP and whose defining standards therefore make at least glancing reference to it.

TCP is designed to use IP, since they were developed together and as you have seen, were even once part of the same specification. They were later split up in order to respect the principles of architectural layering. For this reason, TCP tries to make as few assumptions as possible regarding the underlying protocol over which it runs. It is not as strictly tied to the use of IP as you might imagine, and you can even adapt it for use over other network layer protocols. For our purposes, however, this should be considered mainly an interesting aside.

TCP Functions

You have now seen where TCP comes from and the standards that describe it. As I said in the introduction to this chapter, TCP is a complicated protocol, so it will take some time to explain how it works. Here, I'll describe what TCP does and what it doesn't do.

Preparation for Connection

Both the client and the server create the TCB for the connection at the time that they perform the Open. The client already knows the IP addresses and port numbers for both the client process and the server process it is trying to reach, so it can use these to uniquely identify the connection and the TCB that goes with it.

For the server, the concept of a TCB at this stage of the game is a bit more complex. If the server is waiting for a particular client, it can identify the connection using its own socket and the socket of the client for which it is waiting. Normally, however, the server doesn't know which client is trying to reach it. In fact, more than one client could contact it at nearly the same time.

In this case, the server creates a TCB with an unspecified (zero) client socket number and waits to receive an active Open. It then *binds* the socket number of the client to the TCB for the passive Open as part of the connection process. To allow the server to handle multiple incoming connections, the server process may perform several unspecified passive Opens simultaneously.

The TCB for a connection is maintained throughout the connection and destroyed when the connection is completely terminated, and the device returns to the CLOSED state. TCP does include a procedure that handles the situation in which both devices perform an active Open simultaneously, as I discuss in the next section.

TCP Connection Establishment Process: The Three-Way Handshake

Before TCP can be employed for any actually useful purpose—that is, sending data—a connection must be set up between the two devices that wish to communicate. This process, usually called *connection establishment*, involves an exchange of messages that transitions both devices from their initial connection state (CLOSED) to the normal operating state (ESTABLISHED).

Connection Establishment Functions

The connection establishment process actually accomplishes the following tasks as it creates a connection suitable for data exchange:

Contact and Communication The client and server make contact with each other and establish communication by sending each other messages. The server usually doesn't even know which client it will be talking to before this point, so it discovers this during connection establishment.

Sequence Number Synchronization Each device lets the other know what initial sequence number it wants to use for its first transmission.

Parameter Exchange The two devices exchange certain parameters that control the operation of the TCP connection.

I'll discuss the sequence number synchronization and parameter exchange tasks in the "TCP Connection Establishment Sequence Number Synchronization and Parameter Exchange" section later in this chapter.

TCP Connection Management and Problem Handling

Once both of the devices in a TCP connection have completed connection setup and have entered the ESTABLISHED state, the TCP software is in its normal operating mode. The TCP software will package bytes of data into segments for transmission using the mechanisms described in Chapter 48. TCP will use the sliding windows scheme to control segment size and to provide flow control, congestion handling, and retransmissions as needed.

Once in the sliding windows mode, both devices can remain there indefinitely. Some TCP connections can be very long-lived—in fact, some users maintain certain connections like Telnet sessions for hours or even days at a time. The following circumstances can cause a connection to move out of the ESTABLISHED state:

Connection Termination Either of the devices decides to terminate the connection. This involves a specific procedure that I cover in the “TCP Connection Termination” section later in this chapter.

Connection Disruption A problem of some sort occurs and interrupts the connection.

The TCP Reset Function

In order for it to live up to its job of being a reliable and robust protocol, TCP includes intelligence that allows it to detect and respond to various problems that can occur during an established connection. One of the most common is the *half-open connection*. This situation occurs when, due to some sort of problem, one device closes or aborts the connection without the other one knowing about it. This means one device is in the ESTABLISHED state, while the other may be in the CLOSED state (no connection) or some other transient state. This could happen, for example, one device had a software crash and someone restarted it in the middle of a connection, or if some sort of glitch caused the states of the two devices to become unsynchronized.

To handle half-open connections and other problem situations, TCP includes a special *reset function*. A *reset* is a TCP segment that TCP sends with the *RST* flag set to 1 in its header. Generally speaking, the TCP software generates a reset whenever something unexpected happens. The following are some of the most common cases in which the TCP software generates a reset:

- Receipt of any TCP segment from any device with which the device receiving the segment does not currently have a connection (other than a SYN request for a new connection)
- Receipt of a message with an invalid or incorrect Sequence Number or Acknowledgment Number field, indicating that the message may belong to a prior connection or is spurious in some other way
- Receipt of a SYN message on a port where there is no process listening for connections

Handling Reset Segments

When a device receives a segment with the RST bit, it tells the other device to reset the connection so that the device can reestablish the connection. Like all segments, the reset itself must be checked to ensure that it is valid (by looking at the value of its Sequence Number field). This check prevents a spurious reset from shutting down a connection. Assuming the reset is valid, the handling of the message depends on the state of the device that receives it, as follows:

- If the device is in the LISTEN state, it ignores the reset and remains in that state.
- If the device is in the SYN-RECEIVED state but was previously in the LISTEN state (which is the normal course of events for a server setting up a new connection), it returns to the LISTEN state.
- In any other situation, the reset causes the device to abort the connection and the device returns to the CLOSED state for that connection. The device will advise the higher-layer process that is using TCP that it has closed the connection.

KEY CONCEPT TCP includes a special *connection reset* feature that allows devices to deal with problem situations, such as *half-open connections* or the receipt of unexpected message types. To use the feature, the device detecting the problem sends a TCP segment with the RST (reset) flag set to 1. The receiving device either returns to the LISTEN state, if it was in the process of connection establishment, or closes the connection and returns to the CLOSED state pending a new session negotiation.

Idle Connection Management and Keepalive Messages

One final connection management issue in TCP is how to handle an idle connection; that is, a TCP session that is active but that has no data being transmitted by either device for a prolonged period of time. The TCP standard specifies that the appropriate action to take in this situation is nothing. The reason is that, strictly speaking, there is no need to do anything to maintain an idle connection in TCP. The protocol is perfectly happy to allow both devices to stop transmitting for a very long period of time. Then it simply allows both devices to resume transmissions of data and acknowledgment segments when each one has data to send.

However, in the same way that people become antsy when they are on a telephone call and don't hear anything for a long time, some TCP implementors were concerned that an idle TCP connection might mean that something had broken the connection.

Thus, TCP software often includes an unofficial feature that allows a device with a TCP link to periodically send a null segment, which contains no data, to its peer on the connection. If the connection is still valid, the other device responds with a segment that contains an acknowledgment; if it is not, the other device will reply with a connection reset segment as I described earlier. These segments are sometimes called *TCP keepalive messages*, or just *keepalives*. They are analogous to Border Gateway Protocol (BGP) Keepalive messages (described in Chapter 40).

The use of these messages is quite controversial, and therefore, not universal. Those who oppose using them argue that they are not really necessary, and that sending them represents a waste of internetwork bandwidth and a possible additional cost on metered links (those that charge for each datagram sent). Their key point is that if the connection is not presently being used, it doesn't matter if it is still valid or not; as soon as the connection is used again, if it has broken, in the meantime, TCP can handle that using the reset function mentioned earlier.

Sending a keepalive message can, in theory, break a good TCP session unnecessarily. This may happen if the keepalive is sent during a time when there is an intermittent failure between the client and server. The failure might otherwise have corrected itself by the time the next piece of real data must be sent. In addition, some TCP implementations may not properly deal with the receipt of these segments.

Those in favor of using keepalives point out that each TCP connection consumes a certain amount of resources, and this can be an issue, especially for busy servers. If many clients connect to such a server and don't terminate the TCP connection properly, the server may sit for a long time with an idle connection, using system memory and other resources that it could apply elsewhere.

Since there is no wide acceptance on the use of this feature, devices implementing it include a way to disable it if necessary. Devices are also programmed so that they will not terminate a connection simply because they did not receive a response to a single keepalive message. They may terminate the connection if they do not receive a reply after several such messages have been sent over a period of time.

TCP Connection Termination

As the saying goes, all good things must come to an end, and so it is with TCP connections. The link between a pair of devices can remain open for a considerable period of time, assuming that a problem doesn't force the device to abort the connection. Eventually, however, one or both of the processes in the connection will run out of data to send and will shut down the TCP session, or the user will instruct the device to shut down.

Requirements and Issues in Connection Termination

Just as TCP follows an ordered sequence of operations in order to establish a connection, it also includes a specific procedure for terminating a connection. As with connection establishment, each of the devices moves from one state to the next in order to terminate the connection. This process is more complicated than you might imagine. In fact, an examination of the TCP FSM shows that there are more distinct states involved in shutting down a connection than in setting one up.

The reason that connection termination is complex is that during normal operation, both devices are sending and receiving data simultaneously. Usually, connection termination begins with one device indicating to TCP that it wants to close the connection. The matching process on the other device may not be aware that its peer wants to end the connection at all. Several steps are required to ensure that both devices shut down the connection gracefully and that no data is lost in the process.

Ultimately, shutting down a TCP connection requires the application processes on both ends of the connection to recognize that "the end is nigh" for the connection and that they should stop sending data. For this reason, connection termination is implemented so that each device terminates its end of the connection separately. The act of closing the connection by one device means that device will no longer send data, but can continue to receive it until the other device has decided to stop sending. This allows all data that is pending to be sent by both sides of the communication to be flushed before the connection is ended.

Normal Connection Termination

In the normal case, each side terminates its end of the connection by sending a special message with the FIN (finish) bit set. The FIN message serves as a connection termination request to the other device, while also possibly carrying data like a regular segment. The device receiving the FIN responds with an acknowledgment to the FIN that indicates that it received the acknowledgment. Neither side considers the connection terminated until they both have sent a FIN and received an ACK, thereby finishing the shutdown procedure.

Thus, termination isn't a three-way handshake as with establishment. It is a pair of two-way handshakes. The states that the two devices in the connection move through during a normal connection shutdown are different because the device initiating the shutdown must behave differently than the one that receives the termination request. In particular, the TCP on the device receiving the initial termination request must inform its application process and wait for a signal that the process is ready to proceed. The initiating device doesn't need to do this, since the application started the ball rolling in the first place.

KEY CONCEPT A TCP connection is terminating using a special procedure by which each side independently closes its end of the link. The connection normally begins with one of the application processes signaling to its TCP layer that the session is no longer needed. That device sends a FIN message to tell the other device that it wants to end the connection, which the other device acknowledges. When the responding device is ready, it too sends a FIN that the other device acknowledges; after waiting a period of time for the device to receive the ACK, the device closes the session.

Table 47-4 describes in detail how the connection termination process works. You can also see the progression of states and messages exchanged in Figure 47-5. The table is adapted from Table 47-1, which describes the TCP FSM, but shows what happens for both the server and the client over time during connection shutdown. Either device can initiate connection termination; in this example, I am assuming the client does it. Each row shows the state each device begins in, what action it takes in that state, and what state it transitions to. I have also shown the send and receive stages of both of the steps for each of the client and server's close operations.

Table 47-4: TCP Connection Termination Procedure

Client			Server		
Start State	Action	Transitions to State	Start State	Action	Transitions to State
ESTABLISHED	Client Close Step 1 Transmit: The application using TCP signals that the connection is no longer needed. The client TCP sends a segment with the FIN bit set to request that the connection be closed.	FIN-WAIT-1	ESTABLISHED	At this stage the server is still in normal operating mode.	—
FIN-WAIT-1	The client, having sent a FIN, is waiting for a device to acknowledge it and for the server to send its own FIN. In this state, the client can still receive data from the server but will no longer accept data from its local application to be sent to the server.	—	ESTABLISHED	Client Close Step 1 Receive and Step 2 Transmit: The server receives the client's FIN. It sends an ACK to acknowledge the FIN. The server must wait for the application using it to be told that the other end is closing, so the application here can finish what it is doing.	—
FIN-WAIT-1	Client Close Step 2 Receive: The client receives the ACK for its FIN. It must now wait for the server to close.	FIN-WAIT-2	CLOSE-WAIT	The server waits for the application process on its end to signal that it is ready to close.	—
FIN-WAIT-2	The client is waiting for the server's FIN.	—	CLOSE-WAIT	Server Close Step 1 Transmit: The server's TCP receives a notice from the local application that it is done. The server sends its FIN to the client.	LAST-ACK
FIN-WAIT-2	Server Close Step 1 Receive and Step 2 Transmit: The client receives the server's FIN and sends back an ACK.	TIME-WAIT	LAST-ACK	The server is waiting for an ACK for the FIN that it sent.	—
TIME-WAIT	The client waits for a period of time equal to double the maximum segment life (MSL) time; this wait ensures that the ACK it sent was received.	—	LAST-ACK	Server Close Step 2 Receive: The server receives the ACK to its FIN and closes the connection.	CLOSE
TIME-WAIT	The timer expires after double the MSL time.	CLOSED	CLOSED	The connection is closed on the server's end.	—
CLOSED	The connection is closed.	—	CLOSED	The connection is closed.	—

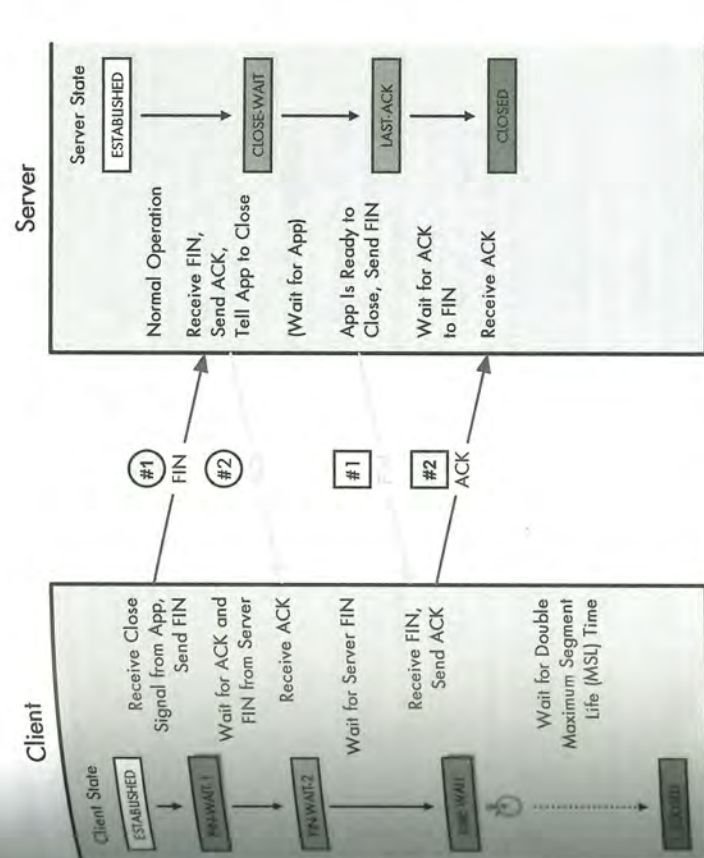


Figure 47-5: TCP connection termination procedure This diagram shows the conventional termination procedure for a TCP session, with one device initiating termination and the other responding. In this case, the client initiates; it sends a FIN, which the server acknowledges. The server waits for the server process to be ready to close and then sends its FIN, which the client acknowledges. The client waits for a period of time in order to ensure that the device receives its ACK, before proceeding to the CLOSED state.

The device receiving the initial FIN may have to wait a fairly long time (in networking terms) in the CLOSE-WAIT state for the application it is serving to indicate that it is ready to shut down. TCP cannot make any assumptions about how long this will take. During this period of time, the server in the previous example may continue sending data, and the client will receive it. However, the client will not send data to the server.

Eventually, the second device (the server in the example) will send a FIN to close its end of the connection. The device that originally initiated the close (the client) will send an ACK for this FIN. However, the client cannot immediately go to the CLOSED state right after sending that ACK because it must allow time for the ACK to travel to the server. Normally, this will be quick, but delays might slow it down somewhat.

The TIME-WAIT State

The TIME-WAIT state is required for two main reasons:

- To provide enough time to ensure that the other device receives the ACK, and to retransmit it if it is lost

World Wide Web System Concepts and Components

Hypertext is the main concept that makes the Web more than just another message transfer system. However, the idea behind hypertext had been around for decades before the Web was born, as had certain software products based on that idea. Obviously, more than just a concept is needed for an idea to be developed into a successful system.

The Web became a phenomenon because it combined the basic idea of hypertext with several other concepts and technologies to create a rich, comprehensive mechanism for interactive communication. This system today encompasses so many different concepts and software elements, and is so integrated with other technologies, that it's difficult to find any two people who agree on what exactly the Web comprises, and which parts are most critical.

For example, one of the keys to the success of the Web is undeniably the combination of the TCP/IP internetworking protocol suite and the Internet infrastructure that connects together the computers of the world. Is the Internet then an essential component of the Web? In many ways, it is; and, in fact, due to how popular the Web is today, it is common to hear people refer to the Web as *the Internet*. We know that this is not a precise use of terms, of course, but it shows how important the Web has become and how closely it is tied to the Internet.

Major Functional Components of the Web

While the Internet and TCP/IP are obviously important parts of the Web's success, they are generic in nature. When it comes to defining the Web system itself more specifically, three particular components are usually considered most essential (see Figure 79-1):

Hypertext Markup Language HTML is a text language used to define hypertext documents. The idea behind HTML was to add simple constructs, called *tags*, to regular text documents, to enable the linking of one document to another, as well as to allow special data formatting and the combining of different types of media. HTML has become the standard language for implementing information in hypertext and has spawned the creation of numerous related languages.

Hypertext Transfer Protocol HTTP is the TCP/IP application layer protocol that implements the Web, by enabling the transfer of hypertext documents and other files between a client and server. HTTP began as a very crude protocol for transferring HTML documents between computers, and it has evolved to a full-featured and sophisticated messaging protocol. It supports transfers of many different kinds of documents, streaming of multiple files on a connection, and various advanced features including caching, proxying, and authentication.

Uniform Resource Identifiers URIs are used to define labels that identify resources on an internetwork so that they can be easily found and referenced. URIs were originally developed to provide a means by which the users of the Web could locate hypertext documents so they could be retrieved. URIs are actually not specific to the Web, though they are most often associated with the Web and HTTP.

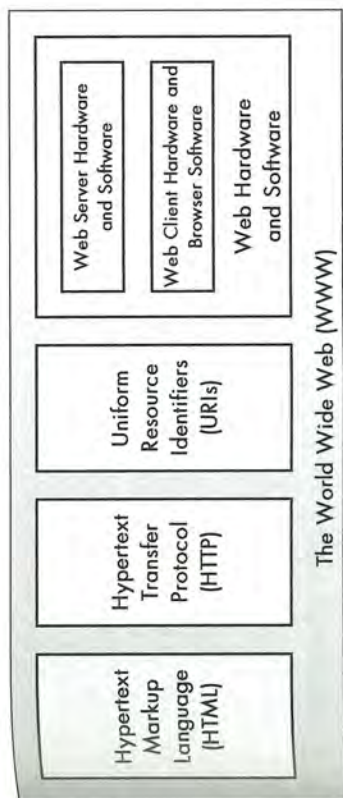


Figure 79-1: Major functional components of the World Wide Web

NOTE Uniform Resource Locators (URLs) are actually a subset of Uniform Resource Identifiers (URIs). The terms are often used interchangeably in World Wide Web discussions.

All three of these components were created and developed at around the same time, and taken together they represent the key technologies that define the Web. In this chapter, I'll describe HTML and the use of URIs in the context of the Web. HTTP is really the heart of the Web and is covered in the remaining five chapters of this part of the book.

Web Servers and Web Browsers

These three main Web components are supplemented by a number of other elements that play supporting roles in rounding out the system as a whole. Chief among these are the hardware and software used to implement client/server communication that makes the Web work, also illustrated in Figure 79-1: *web servers* and *web browsers*.

Web servers are computers that run special server software that allows them to provide hypertext documents and other files to clients who request them. Millions of such machines around the world now serve as a virtual distributed repository of the enormous wealth of information that the Web represents.

Web browsers are HTTP client software programs that run on TCP/IP client computers to access web documents on web servers. These browser programs retrieve hypertext documents and display them, and they also implement many of the Web's advanced features, such as caching. Today's browsers support a wide variety of media, allowing the Web to implement many different functions aside from hypertext document transfer. Examples include displaying images, playing sounds, and implementing interactive programs.

Last, but certainly not least, the *users* of the Web are perhaps its most important component. User involvement has had more of a role in shaping the development of Web technology than any other networking application. The Web began as a simple means of exchanging documents; today, it has grown to encompass thousands of different applications and services, largely as a result of the creativity of its users. Content providers have pushed the boundaries of what the Web can do by creating new ideas for information and services to satisfy the insatiable demands of the end-user community.

rather than a persistent one (as explained in the previous section), and the server will avoid using HTTP/1.1 headers in its reply. The version token is sent in uppercase letters, as HTTP/0.9, HTTP/1.0, or HTTP/1.1—just the way I’ve been doing throughout my discussion of the protocol.

Headers

After the request line come any of the headers that the client wants to include in the message. In these headers, details are provided to the server about the request. The headers all use the same structure, but are organized into the following categories based on the functions they serve and whether they are specific to one kind of message:

General Headers General headers refer mainly to the message itself, as opposed to its contents, and they are used to control its processing or provide the recipient with extra information. They are not particular to either Request or Response messages, so they can appear in either. Also, they are not specifically relevant to any entity the message may be carrying.

Request Headers These headers convey to the server more details about the nature of the client’s request, and they give the client more control over how the request is handled. For example, special request headers can be used by the client to specify a conditional request—one that is filled only if certain criteria are met. Others can tell the server which formats or encodings the client is able to process in a Response message.

Entity Headers These are headers that describe the entity contained in the body of the request, if any.

KEY CONCEPT HTTP Request messages are the means by which HTTP clients ask servers to take a particular type of action, such as sending a file or processing user input. Each Request message begins with a *request line*, which contains three critical pieces of information: the *method* (type of action) the client is requesting, the *URI* of the resource on which the client wishes the action to be performed, and the version of HTTP that the client is using. After the request line comes a set of message headers related to the request, followed by a blank line, and then optionally, the message body of the request.

Request headers are obviously used only in Request messages, but both general headers and entity headers can appear in either a Request or a Response message. Since there are so many headers and most are not particular to one message type, I describe them in detail in the next chapter.

HTTP Response Message Format

Each Request message sent by an HTTP client to a server prompts the server to send back a *Response message*. Actually, in certain cases, the server may send two responses: a preliminary response, followed by the real one. Usually though, one

request yields one response, which indicates the results of the server’s processing of the request, and a response often also carries an entity (file or resource) in the message body.

Like Request messages, Response messages use their own specific format that is based on the HTTP generic message format described earlier in this chapter. The format Response message header is as follows (see Figure 81-2):

```
<status-line>
<general-headers>
<response-headers>
<entity-headers>
<empty-line>
[<message-body>]
[<message-trailers>]
```

HTTP/1.1 200 OK			Status Line
Date: Fri, 19 May 2006 21:12:58 GMT	Connection: close		General Headers
Server: Apache/1.3.3.7	Accept-Ranges: bytes		Response Headers
Content-Type: text/html	Content-Length: 170	Last-Modified: Wed, 17 May 2006 10:14:49 GMT	Entity Headers
<html>			Message Body
<head>			
<title>Welcome to the Amazing Site!</title>			
</head>			
<body>			
This site is under construction. Please come back later. Sorry!			
</body>			
</html>			

Figure 81-2: HTTP Response message format This figure illustrates the construction of an HTTP Response message and includes an example of both message headers and body. The status code 200 indicates that this is a successful response to a request; it contains a brief text HTML entity in the message body. See Figure 81-1 for the HTTP Request message format.

Status Line

The *status line* (note that this is not called the *response line*) is the start line used for Response messages. It has two functions: to tell the client what version of the protocol the server is using and to communicate a summary of the results of processing the client’s request. The formal syntax for the status line is as follows:

```
<HTTP-VERSION> <status-code> <reason-phrase>
```

Each of the status line components is discussed in the following sections.

HTTP Version

The *HTTP-VERSION* label in the status line serves the same purpose as it does in the request line of a Request message (described in the previous section). Here, it tells the client the version number that the server is using for its response. It uses the same format as in the request line, with the version in uppercase as HTTP/0.9, HTTP/1.0, or HTTP/1.1. The server is required to return an HTTP version number that is no greater than the number the client sent in its request.

Status Code and Reason Phrase

The *status code* and *reason phrase* provide information about the results of processing the client's request in two different forms. The status code is a three-digit number that indicates the formal result that the server is communicating to the client. It is intended for the client HTTP implementation to process so the software can take appropriate action. The reason phrase is an additional, descriptive text string, which can be displayed to the human users of the HTTP client so they can see how the server responded. I describe status codes and reason phrases later in this chapter, and also list all of the standard codes.

Headers

The Response message will always include a number of headers that provide extra information about it. Response message headers fall into three categories:

General Headers General headers that refer to the message itself and are not specific to Response messages or the entity in the message body. These are the same as the generic headers that can appear in Request messages (though certain headers appear more often in responses, and others are more common in requests).

Response Headers These headers provide additional data that expands on the summary result information in the status line. The server may also return extra result information in the body of the message, especially when an error occurs.

Entity Headers These are headers that describe the entity contained in the body of the response, if any. These are the same entity headers that can appear in a Request message, but they are seen more often in response messages.

Most Response messages contain an entity in the message body. In the case of a successful request to retrieve a resource, this is the resource itself. Responses indicating unsuccessful requests usually contain detailed error information, often in the form of an HTML-formatted error message.

NOTE *Entity headers may appear in a Response message to describe the resource that is the subject of the request, even if the entity itself is not sent in the message. This occurs when the HEAD method is used to request only the headers associated with an entity.*

Response headers are used only in Response messages, while the others are general with respect to message type. See Chapter 82 for more details about HTTP headers.