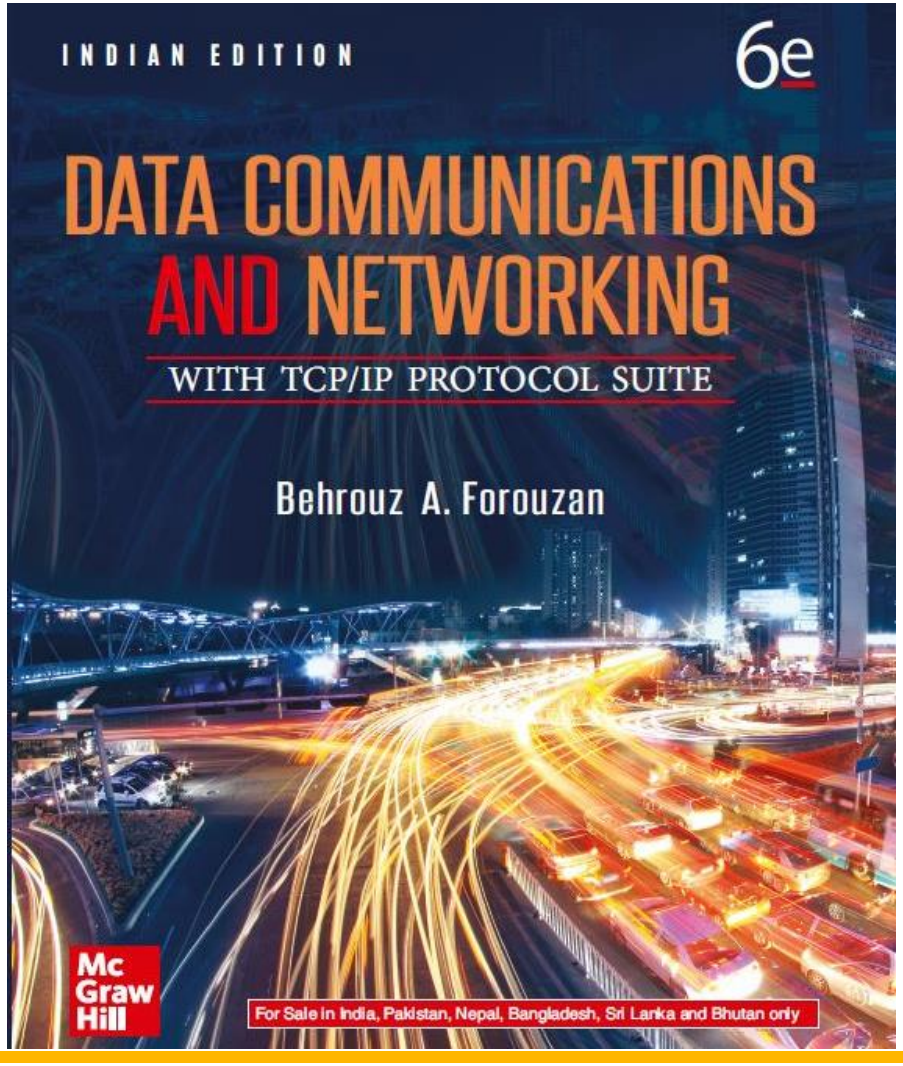
Because learning changes everything.®

Chapter 10

Application Layer

Data Communications and Networking, With TCP/IP protocol suite
Sixth Edition
Behrouz A. Forouzan

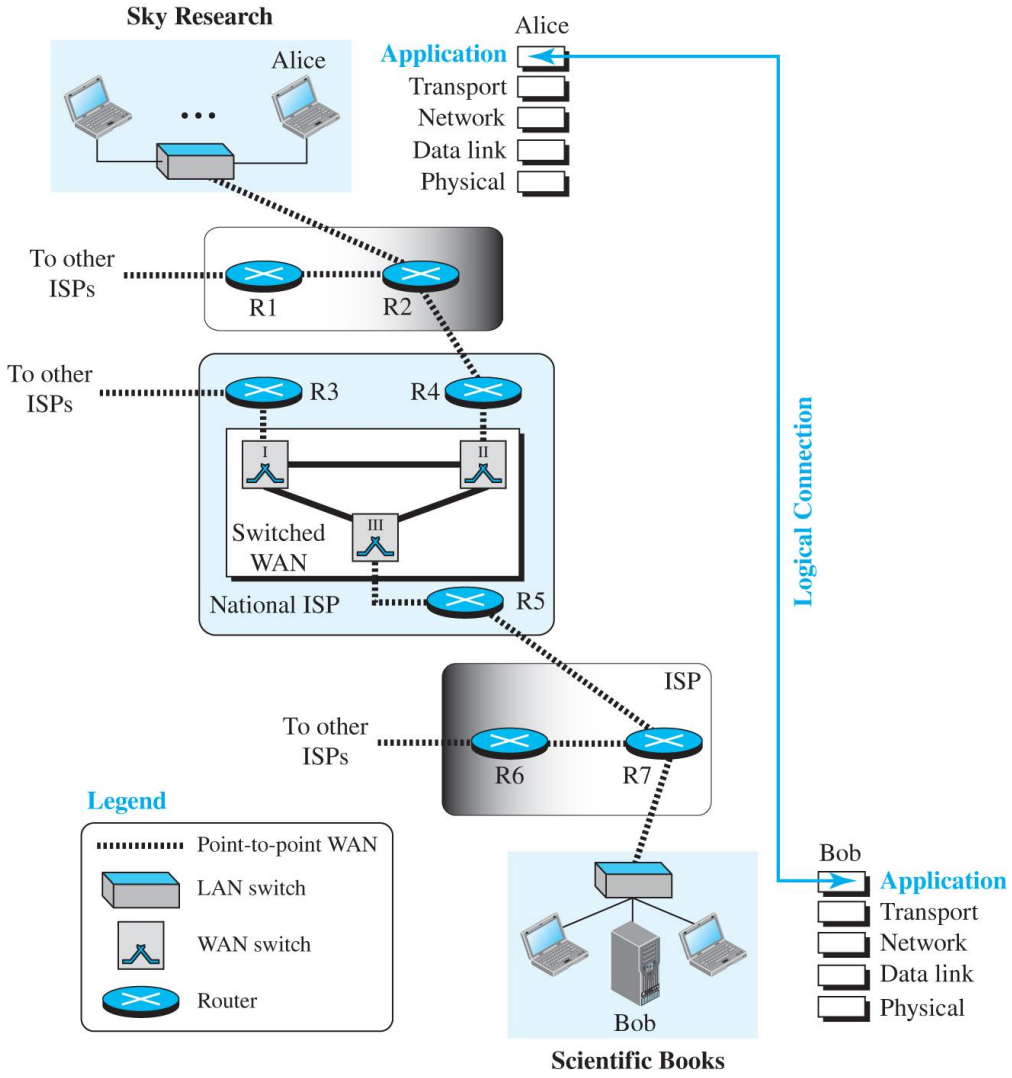


Chapter 10: Outline

10.1 INTRODUCTION

The application layer provides services to the user. Communication is provided using a logical connection, which means that the two application layers assume that there is an imaginary direct connection through which they can send and receive messages. Figure 10.1 shows the idea behind this logical connection.

Figure 10.1 Logical connection at the application layer



Access the text alternative for slide images.

10.1.1 Providing Services

All communication networks that started before the Internet were designed to provide services to network users. Most of these networks, however, were originally designed to provide one specific service. For example, the telephone network was originally designed to provide voice service: to allow people all over the world to talk to each other. This network, however, was later used for some other services, such as facsimile (fax), enabled by users adding some extra hardware at both ends.

Standard and Nonstandard Protocols

To provide a smooth operation of the Internet, the protocols used in the first four layers of the TCP/IP suite need to be standardized and documented. They normally become part of the package that is included in operating systems such as Windows or UNIX. To be flexible, however, the application-layer protocols can be both standard and nonstandard.

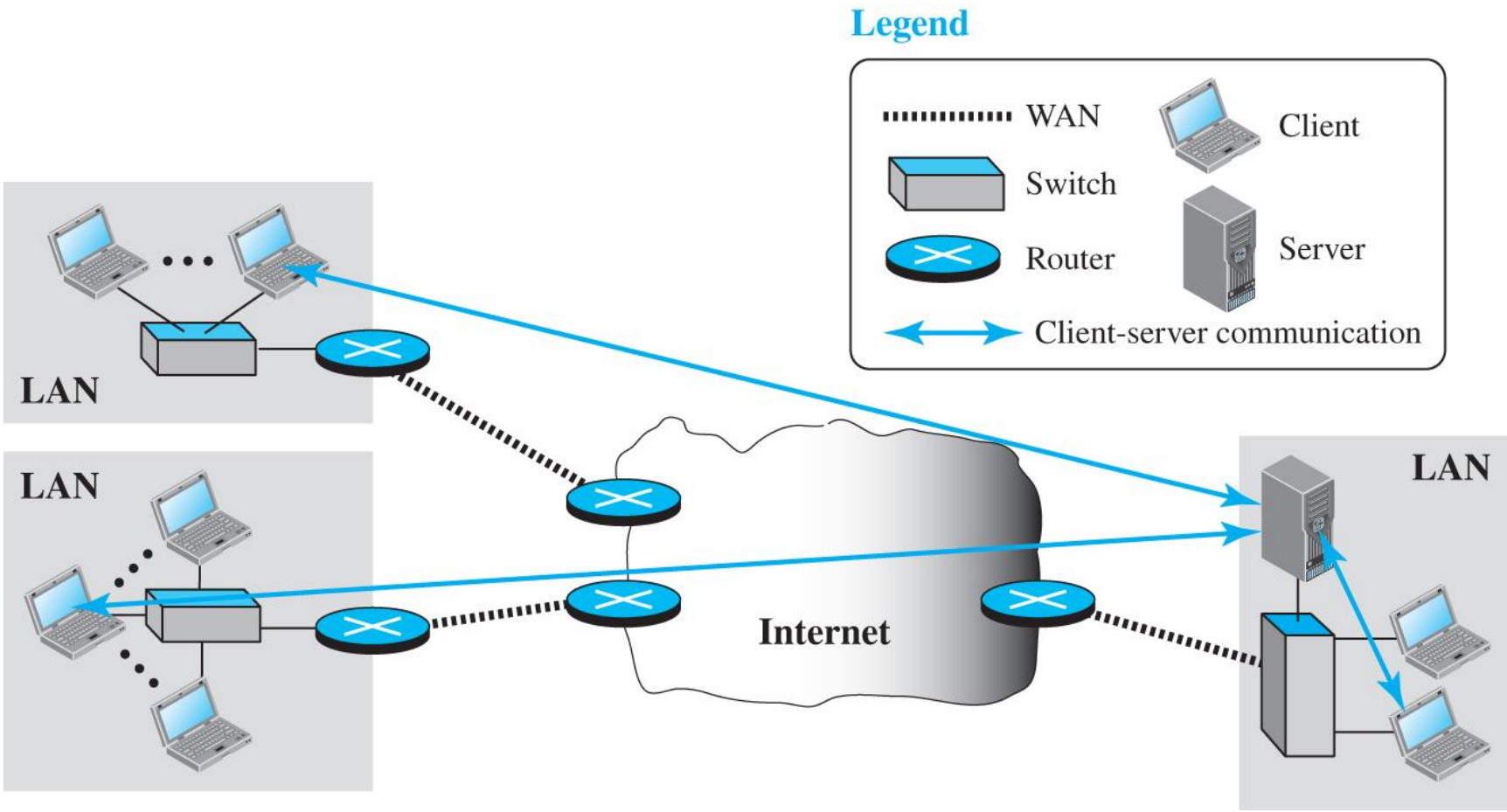
10.1.2 Application-Layer Paradigms

It should be clear that to use the Internet we need two application programs to interact with each other: one running on a computer somewhere in the world, the other running on another computer somewhere else in the world. The two programs need to send messages to each other through the Internet infrastructure. However, we have not discussed what the relationship should be between these programs. Two paradigms have been developed: the client-server paradigm and the peer-to-peer paradigm. We briefly introduce these two paradigms here.

Traditional Paradigm: Client-Server

The traditional paradigm is called the client-server paradigm. It was the most popular paradigm until a few years ago. In this paradigm, the service provider is an application program, called the server process; it runs continuously, waiting for another application program, called the client process, to make a connection through the Internet and ask for service. There are normally some server processes that can provide a specific type of service, but there are many clients that request service from any of these server processes. The server process must be running all the time; the client process is started when the client needs to receive service.

Figure 10.2 Example of a client-server paradigm

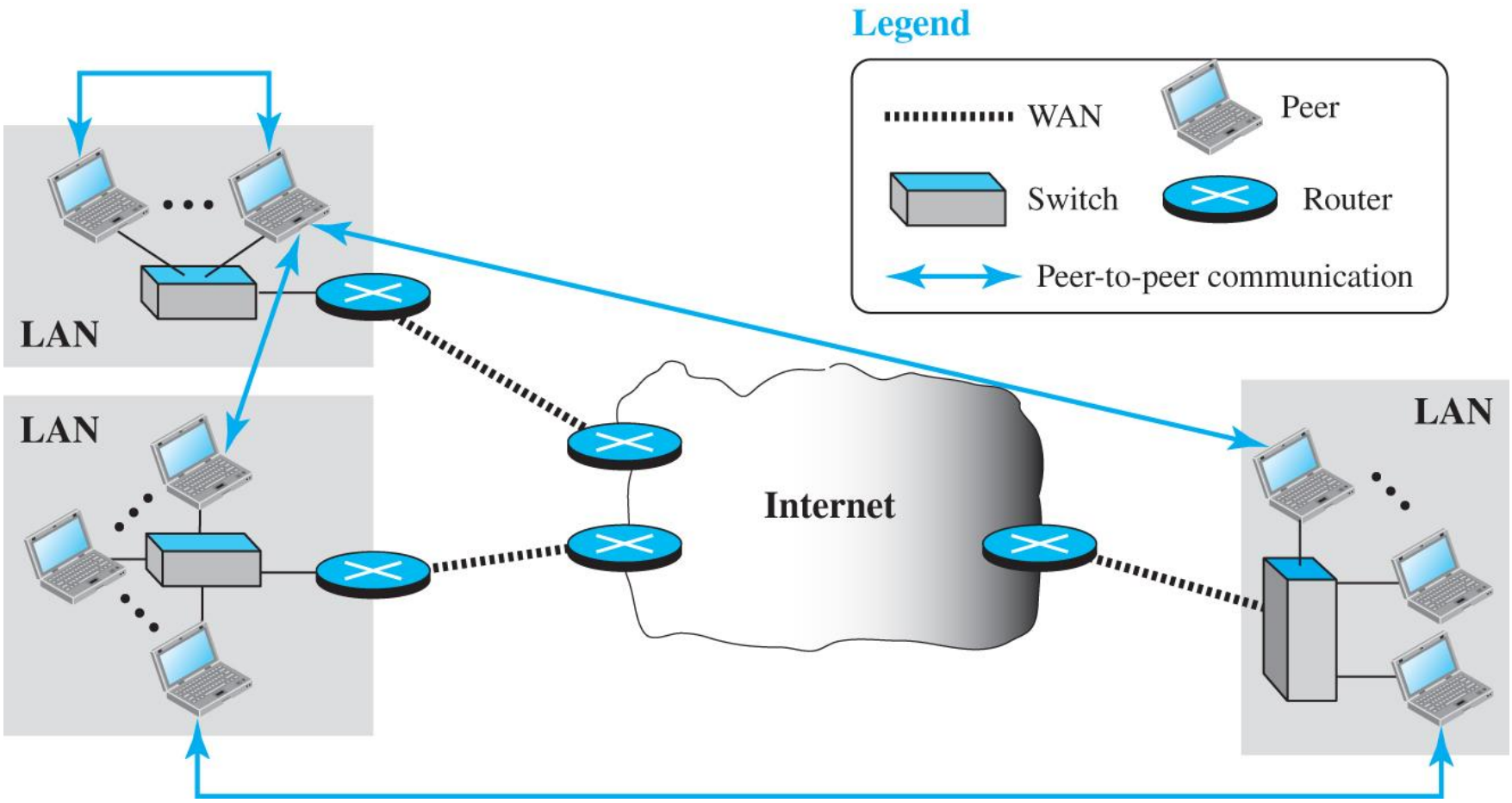


Access the text alternative for slide images.

New Paradigm: Peer-to-Peer

A new paradigm, called the peer-to-peer paradigm (often abbreviated P2P paradigm) has emerged to respond to the needs of some new applications. In this paradigm, there is no need for a server process to be running all the time and waiting for the client processes to connect. The responsibility is shared between peers. A computer connected to the Internet can provide service at one time and receive service at another time. A computer can even provide and receive services at the same time. Figure 10.3 shows an example of communication in this paradigm.

Figure 10.3 Example of a peer-to-peer paradigm



Access the text alternative for slide images.

Mixed Paradigm

An application may choose to use a mixture of the two paradigms by combining the advantages of both. For example, a light-load client-server communication can be used to find the address of the peer that can offer a service. When the address of the peer is found, the actual service can be received from the peer by using the peer-to-peer paradigm.

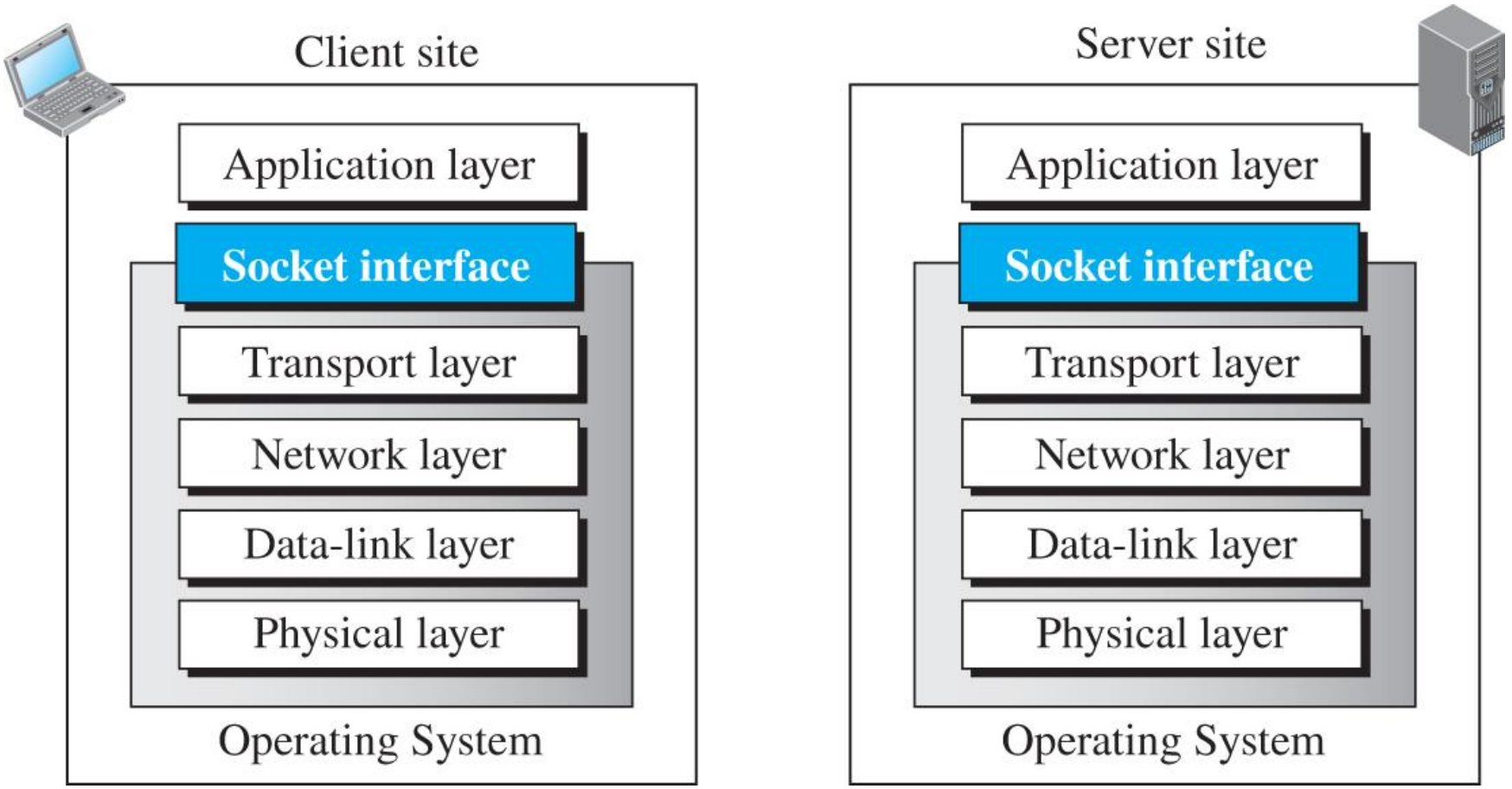
10.2 CLIENT-SERVER PROGRAMMING

In a client-server paradigm, communication at the application layer is between two running application programs called processes: a client and a server. A client is a running program that initializes the communication by sending a request; a server is another application program that waits for a request from a client. The server handles the request received from a client, prepares a result, and sends the result back to the client.

10.2.1 Application Programming Interface

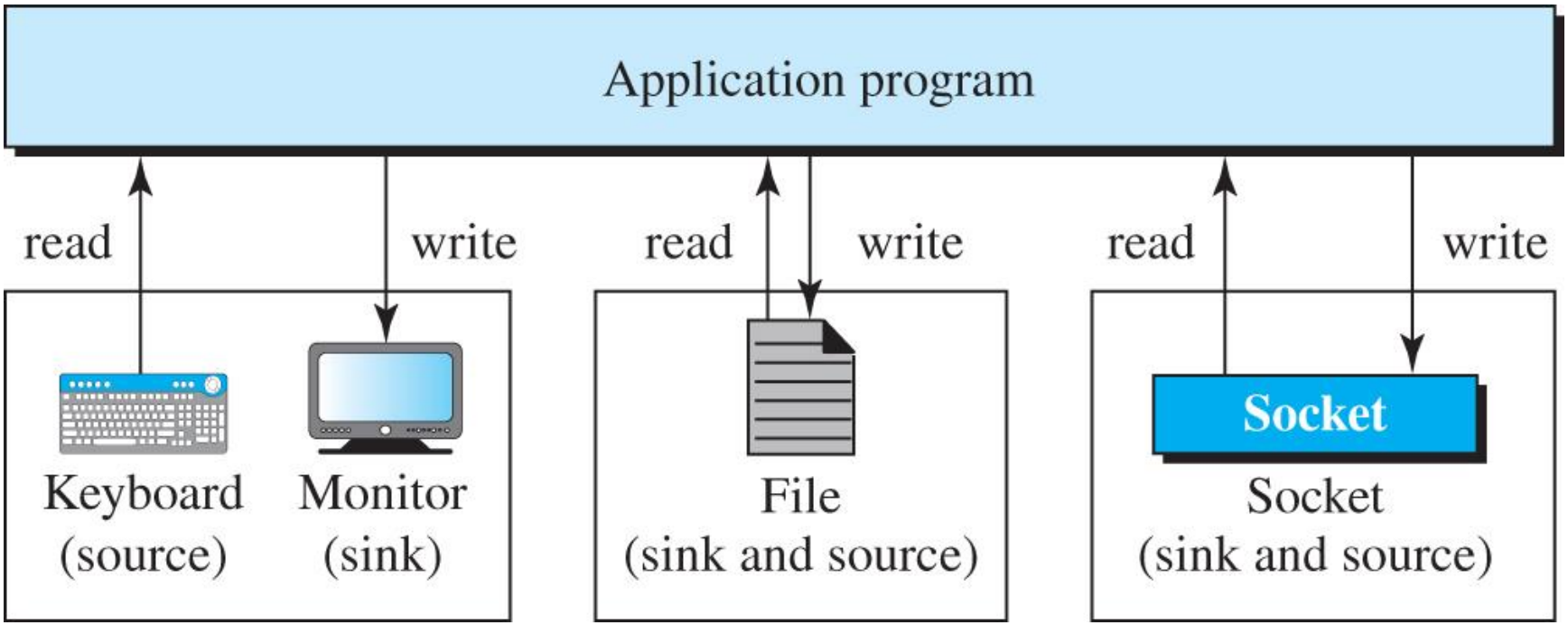
How can a client process communicate with a server process? A computer program is normally written in a computer language with a predefined set of instructions that tells the computer what to do. If we need a process to be able to communicate with another process, we need a new set of instructions to tell the lowest four layers of the TCP/IP suite to open the connection, send and receive data from the other end, and close the connection. A set of instructions of this kind is normally referred to as an application programming interface (API).

Figure 10.4 Position of the socket interface



Access the text alternative for slide images.

Figure 10.5 A sockets used like other sources and sinks

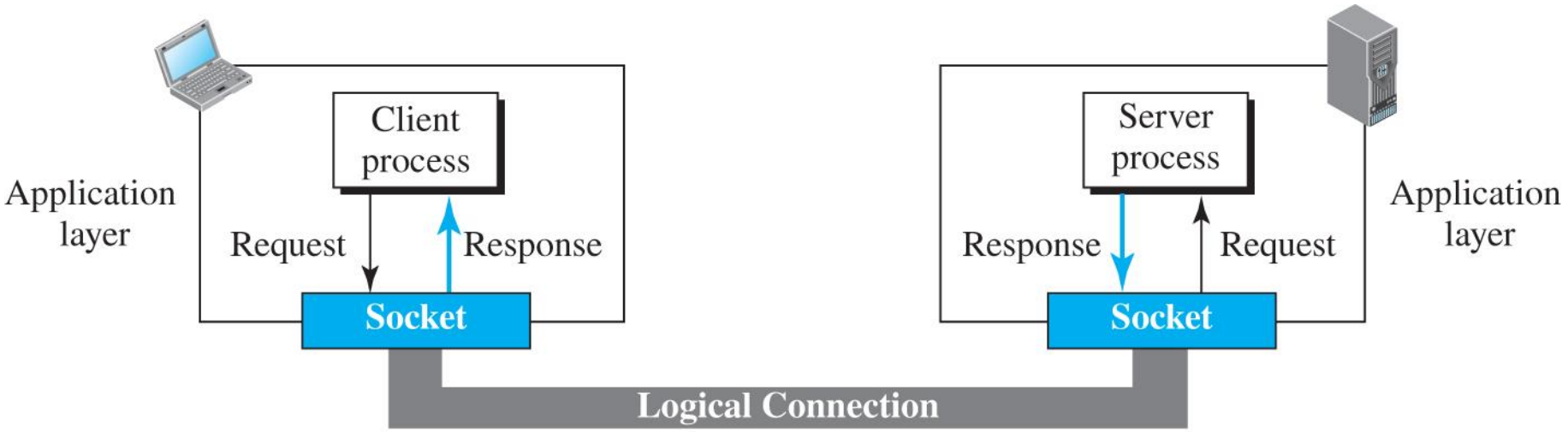


Access the text alternative for slide images.

Sockets

Although a socket is supposed to behave like a terminal or a file, it is not a physical entity like them; it is an abstraction. It is a data structure that is created and used by the application program.

Figure 10.6 Use of sockets in process-to-process communication



Access the text alternative for slide images.

Socket Addresses

The interaction between a client and a server is two-way communication. In a two-way communication, we need a pair of addresses: local (sender) and remote (receiver). The local address in one direction is the remote address in the other direction and vice versa. Since communication in the client-server paradigm is between two sockets, we need a pair of socket addresses for communication: a local socket address and a remote socket address. However, we need to define a socket address in terms of identifiers used in the TCP/IP protocol suite.

Figure 10.7 A socket address

Example 10.1

We can find a two-level address in telephone communication. A telephone number can define an organization, and an extension can define a specific connection in that organization. The telephone number in this case is similar to the IP address, which defines the whole organization; the extension is similar to the port number, which defines the particular connection.

Finding Socket Addresses

How can a client or a server find a pair of socket addresses for communication? The situation is different for each site.

10.2.2 Using Services of Transport Layer

A pair of processes provide services to the users of the Internet, human or programs. A pair of processes, however, need to use the services provided by the transport layer for communication because there is no physical communication at the application layer. As we discussed before, there are three common transport-layer protocols in the TCP/IP suite: UDP, TCP, and SCTP. Most standard applications have been designed to use the services of one of these protocols.

UDP Protocol

UDP provides connectionless, unreliable, datagram service. Connectionless service means that there is no logical connection between the two ends exchanging messages. Each message is an independent entity encapsulated in a packet called a datagram. UDP does not see any relation (connection) between consequent datagrams coming from the same source and going to the same destination.

TCP Protocol

TCP provides connection-oriented, reliable, byte-stream service. TCP requires that two ends first create a logical connection between themselves by exchanging some connection-establishment packets. This phase, which is sometimes called handshaking, establishes some parameters between the two ends including the size of the data packets to be exchanged, the size of buffers to be used for holding the chunks of data until the whole message arrives, and so on.

SCTP Protocol

SCTP provides a service that is a combination of the two other protocols. Like TCP, SCTP provides a connection-oriented, reliable service, but it is not byte-stream oriented. It is a message-oriented protocol like UDP. In addition, SCTP can provide multi-stream service by providing multiple network-layer connections.

10-3 STANDARD APPLICATIONS

During the lifetime of the Internet, several client-server application programs have been developed. We do not have to redefine them, but we need to understand what they do. For each application, we also need to know the options available to us. The study of these applications and the ways they provide different services can help us to create customized applications in the future.

10.3.1 World Wide Web and HTTP

In this section, we first introduce the World Wide Web (abbreviated WWW or Web). We then discuss the HyperText Transfer Protocol (HTTP), the most common client-server application program used in relation to the Web.

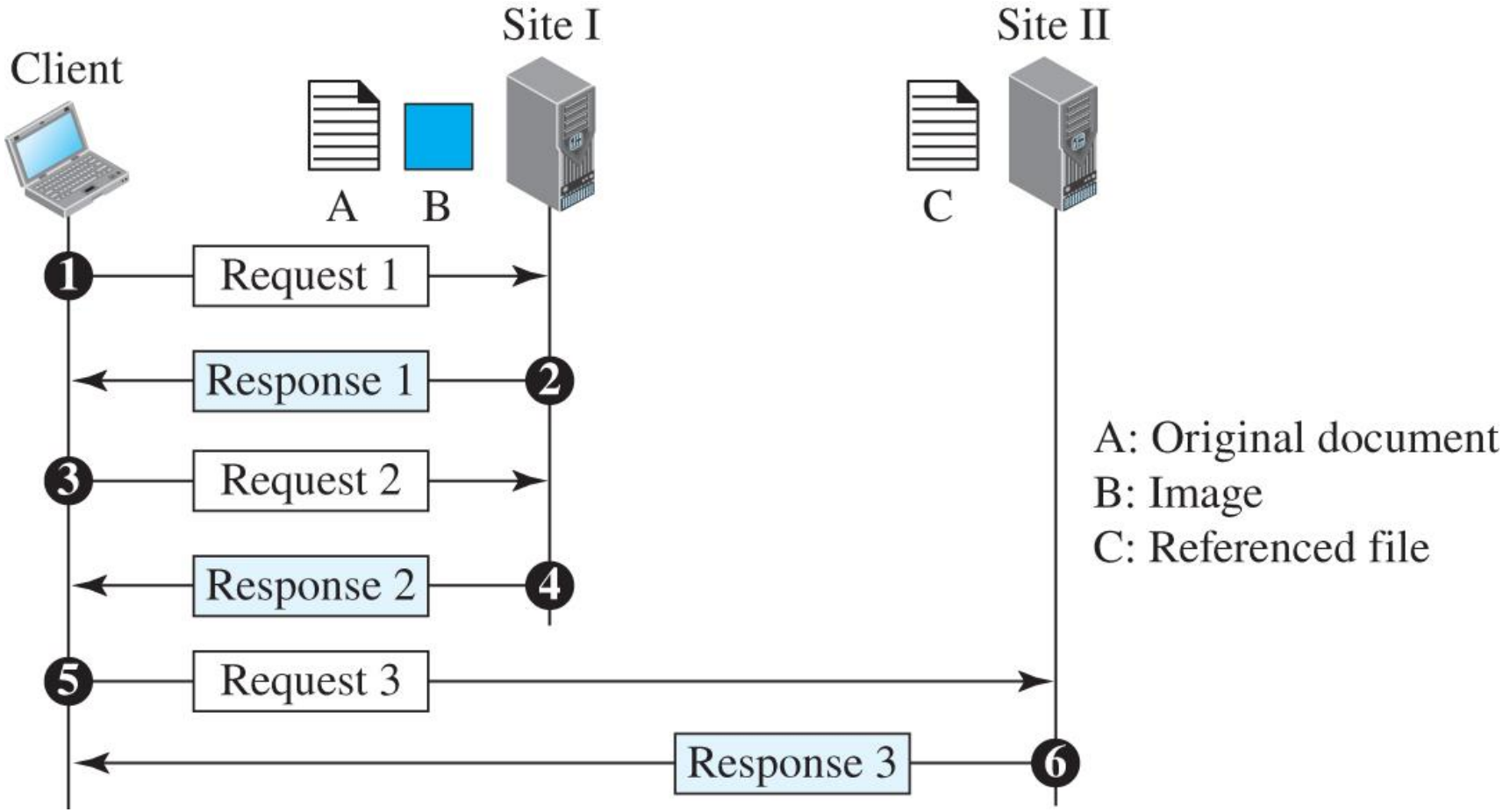
World Wide Web

The idea of the Web was first proposed by Tim Berners-Lee in 1989 at CERN, the European Organization for Nuclear Research, to allow several researchers at different locations throughout Europe to access each others' researches. The commercial Web started in the early 1990s.

Example 10.2

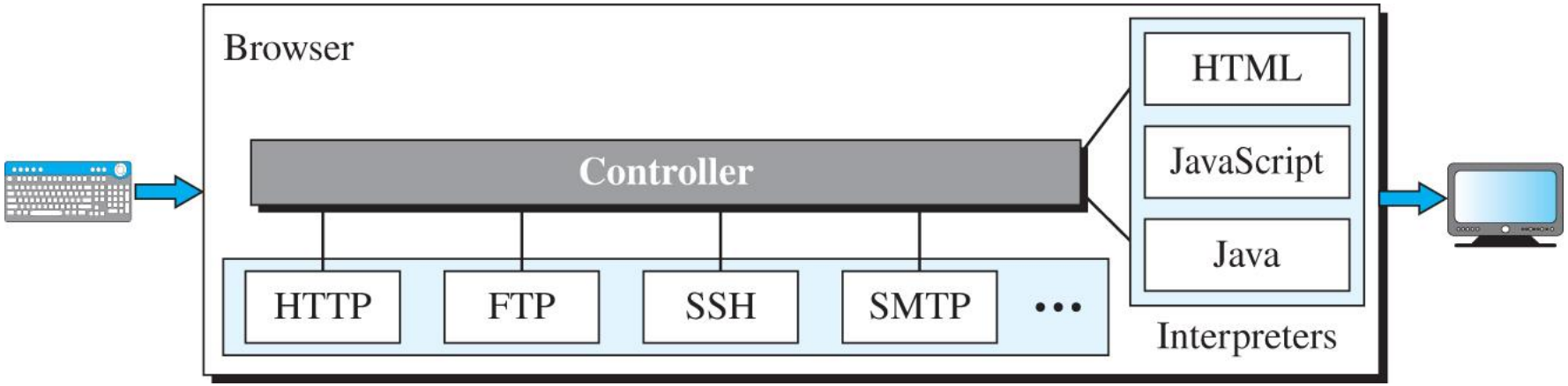
Assume we need to retrieve a scientific document that contains one reference to another text file and one reference to a large image. Figure 10.8 shows the situation.

Figure 10.8 Example 10.2



Access the text alternative for slide images.

Figure 10.9 Browser



Access the text alternative for slide images.

Example 10.3

The URL http://www.mhhe.com/compsci/forouzan/ defines the web page related to one of the computers in the McGraw-Hill company (the three letters www are part of the host name and are added to the commercial host). The path is *compsci/forouzan/*, which defines Forouzan's web page under the directory *compsci* (computer science).

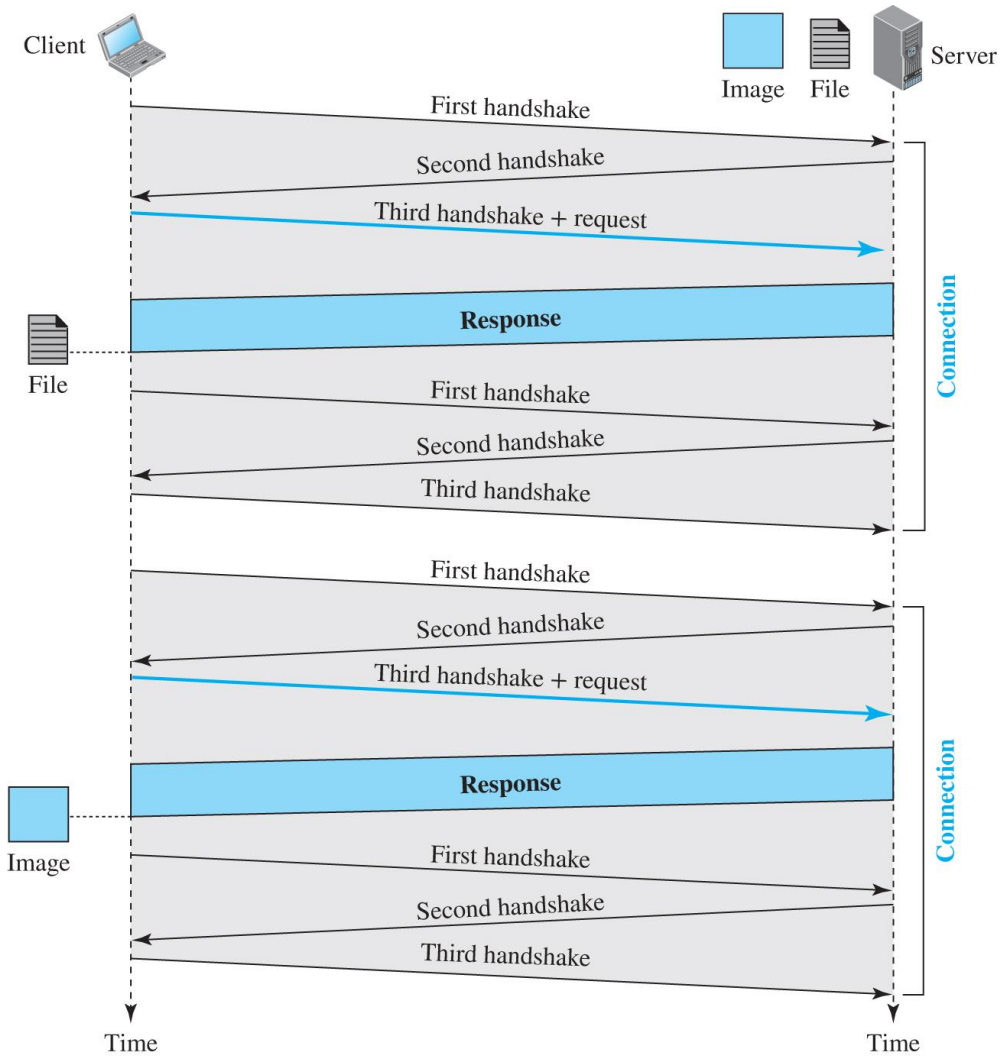
HyperText Transfer Protocol (HTTP)

The HyperText Transfer Protocol (HTTP) is a protocol that is used to define how the client-server programs can be written to retrieve web pages from the Web. An HTTP client sends a request; an HTTP server returns a response. The server uses the port number 80; the client uses a temporary port number.

Example 10.4

Figure 10.10 shows an example of a *nonpersistent* connection. The client needs to access a file that contains one link to an image. The text file and image are located on the same server. Here we need two connections. For each connection, TCP requires at least three handshake messages to establish the connection, but the request can be sent with the third one. After the connection is established, the object can be transferred. After receiving an object, another three handshake messages are needed to terminate the connection.

Figure 10.10 Example 10.4

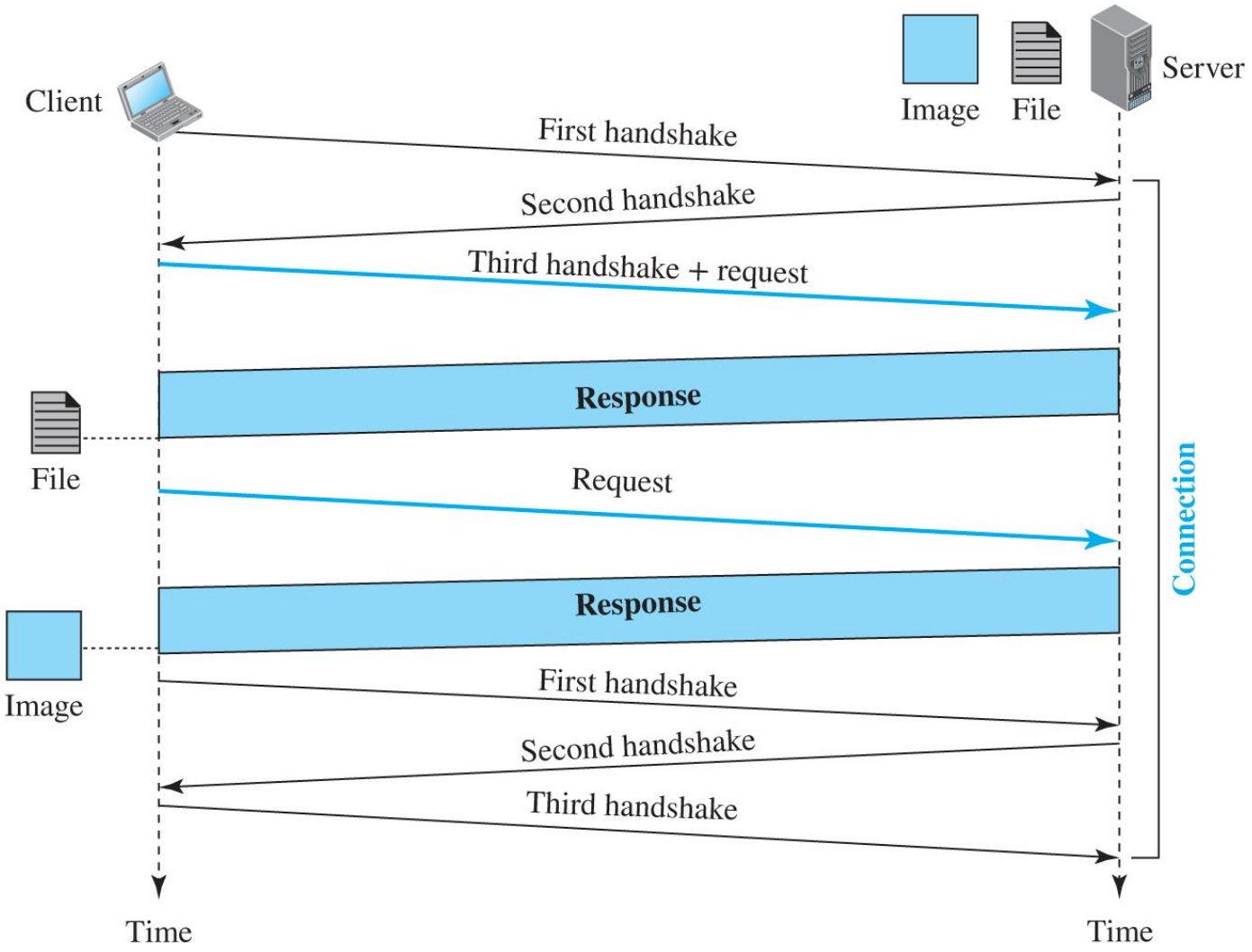


Access the text alternative for slide images.

Example 10.5

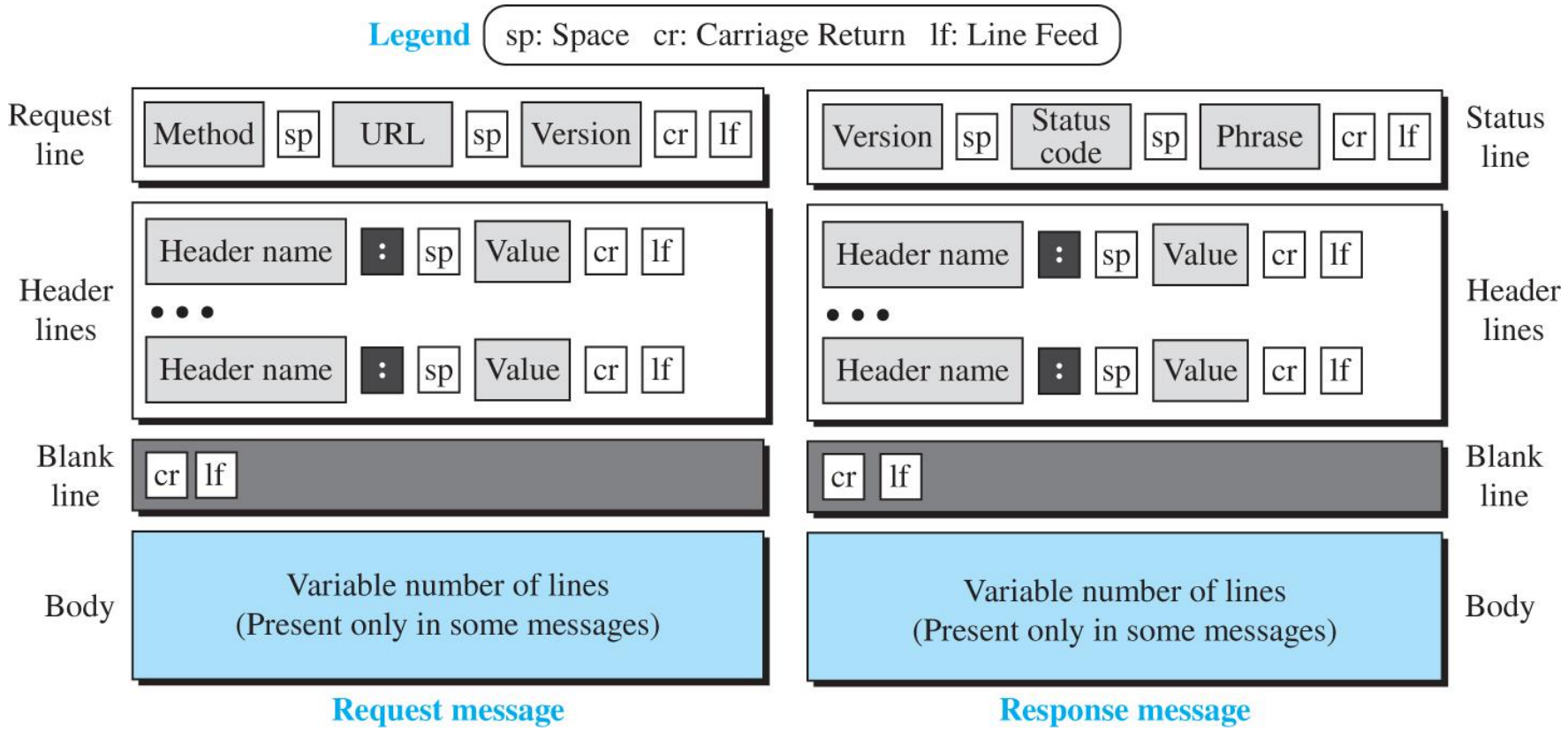
Figure 10.11 shows the same scenario as in Example 10.4, but using a persistent connection. Only one connection establishment and connection termination is used, but the request for the image is sent separately.

Figure 10.11 Example 10.5



Access the text alternative for slide images.

Figure 10.12 Formats of the request and response messages



Access the text alternative for slide images.

Table 10.1 Methods

Method	*Action*
GET	Requests a document from the server
HEAD	Requests information about a document but not the document itself
PUT	Sends a document from the client to the server
POST	Sends some information from the client to the server
TRACE	Echoes the incoming request
DELETE	Removes the web page
CONNECT	Reserved
OPTIONS	Inquires about available options

Table 10.2 Request header names

Header	*Description*
User-agent	Identifies the client program
Accept	Shows the media format the client can accept
Accept-charset	Shows the character set the client can handle
Accept-encoding	Shows the encoding scheme the client can handle
Accept-language	Shows the language the client can accept
Authorization	Shows what permissions the client has
Host	Shows the host and port number of the client
Date	Shows the current date
Upgrade	Specifies the preferred communication protocol
Cookie	Returns the cookie to the server (explained later in this section)
If-Modified-Since	Specifies if the file has been modified since a specific date

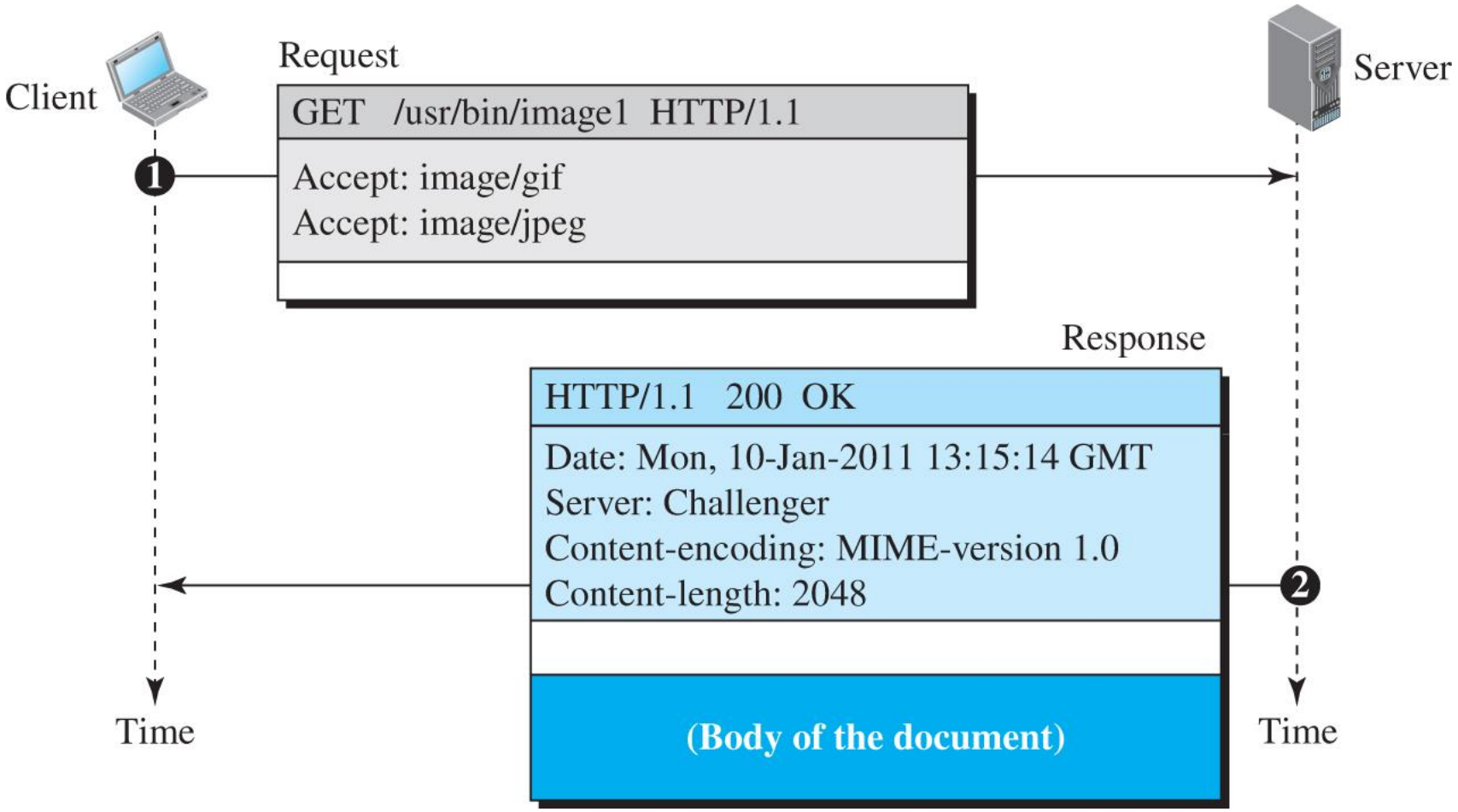
Table 10.3 Response header names

Header	*Description*
Date	Shows the current date
Upgrade	Specifies the preferred communication protocol
Server	Gives information about the server
Set-Cookie	The server asks the client to save a cookie
Content-Encoding	Specifies the encoding scheme
Content-Language	Specifies the language
Content-Length	Shows the length of the document
Content-Type	Specifies the media type
Location	To ask the client to send the request to another site
Accept-Ranges	The server will accept the requested byte-ranges
Last-modified	Gives the date and time of the last change

Example 10.6

This example retrieves a document (see Figure 10.13). We use the GET method to retrieve an image with the path /usr/bin/image1. The request line shows the method (GET), the URL, and the HTTP version (1.1). The header has two lines that show that the client can accept images in the GIF or JPEG format. The request does not have a body. The response message contains the status line and four lines of header. The header lines define the date, server, content encoding (MIME version, which will be described in electronic mail), and length of the document. The body of the document follows the header.

Figure 10.13 Example 10.6

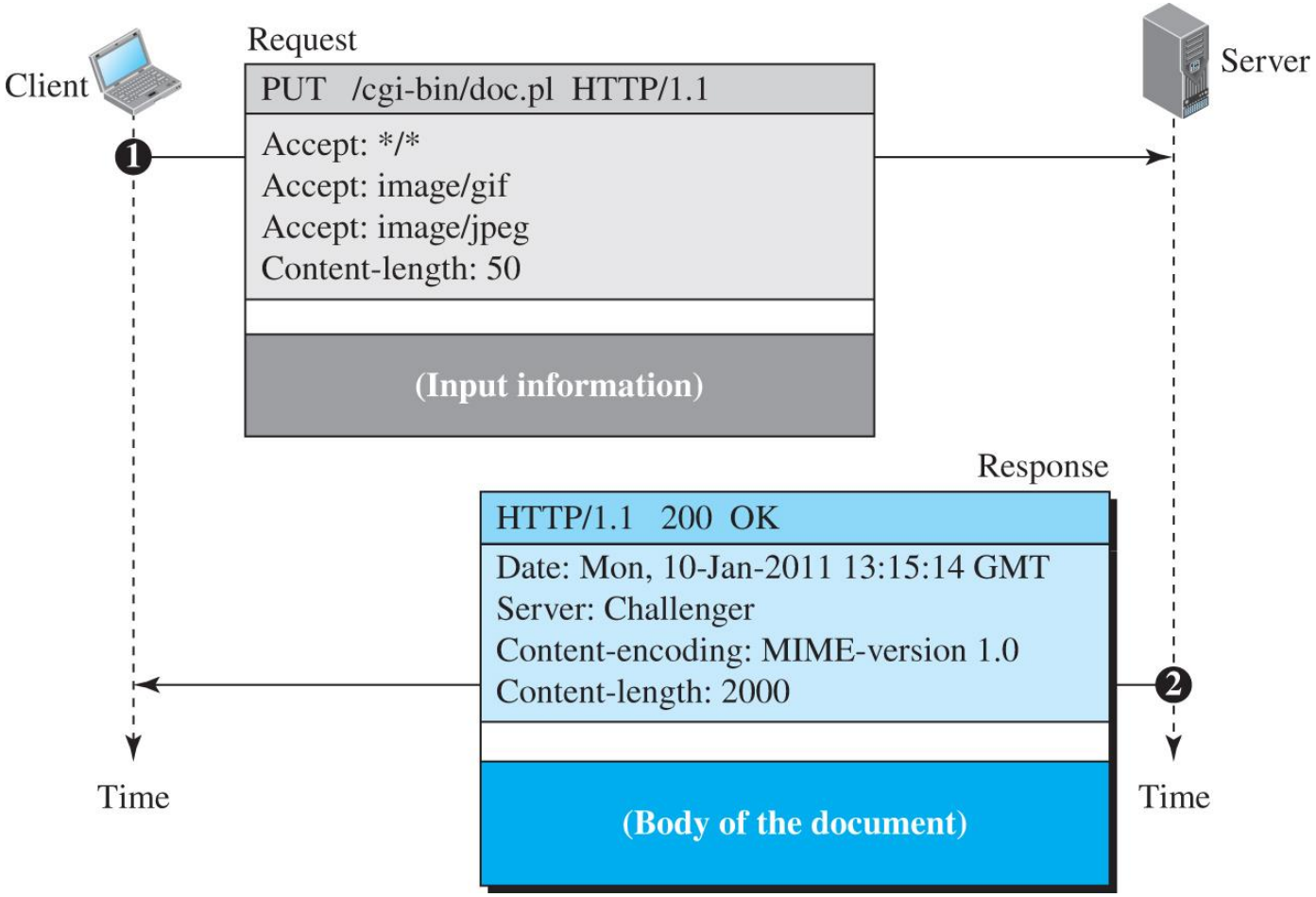


Access the text alternative for slide images.

Example 10.7

In this example, the client wants to send a web page to be posted on the server. We use the PUT method. The request line shows the method (PUT), URL, and HTTP version (1.1). There are four lines of headers. The request body contains the web page to be posted. The response message contains the status line and four lines of headers. The created document, which is a CGI document, is included as the body (see Figure 10.14).

Figure 10.14 Example 10.7



Access the text alternative for slide images.

Example 10.8

The following shows how a client imposes the modification data and time condition on a request.

GET http://www.commonServer.com/information/file1 HTTP/1.1	**Request line**
If-Modified-Since: Thu, Sept 04 00:00:00 GMT	**Header line**
	Blank line

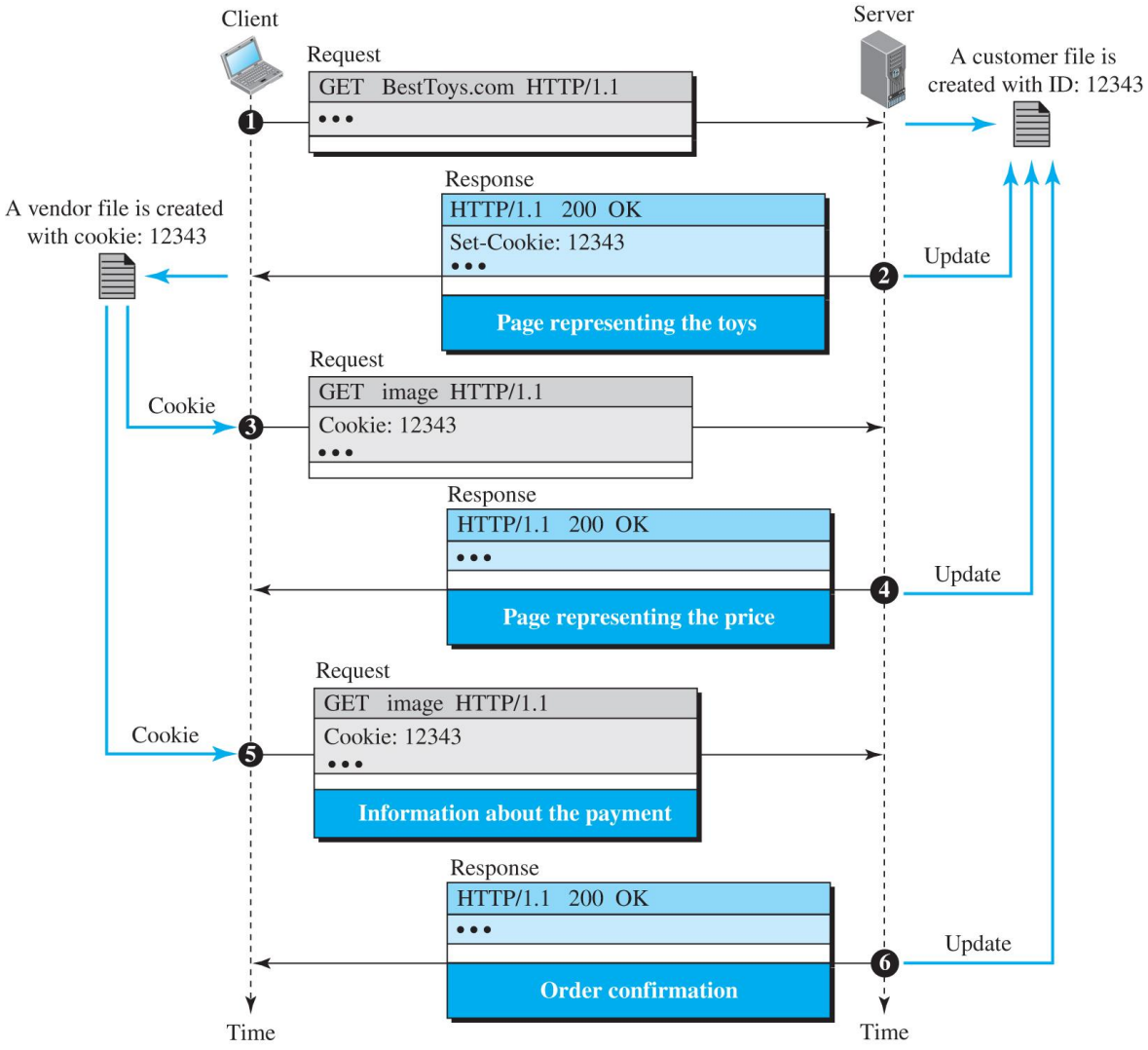
The status line in the response shows the file was not modified after the defined point in time. The body of the response message is also empty.

HTTP/1.1 304 Not Modified	**Status line**
Date: Sat, Sept 06 08 16:22:46 GMT	**First header line**
Server: commonServer.com	**Second header line**
	Blank line
(Empty Body)	**Empty body**

Example 10.9

Figure 10.15 shows a scenario in which an electronic store can benefit from the use of cookies. Assume a shopper wants to buy a toy from an electronic store named BestToys. The shopper browser (client) sends a request to the BestToys server. The server creates an empty shopping cart (a list) for the client and assigns an ID to the cart (for example, 12343). The server then sends a response message, which contains the images of all toys available, with a link under each toy that selects the toy if it is being clicked. This response message also includes the Set-Cookie header line whose value is 12343. The client displays the images and stores the cookie value in a file named BestToys.

Figure 10.15 Example 10.9

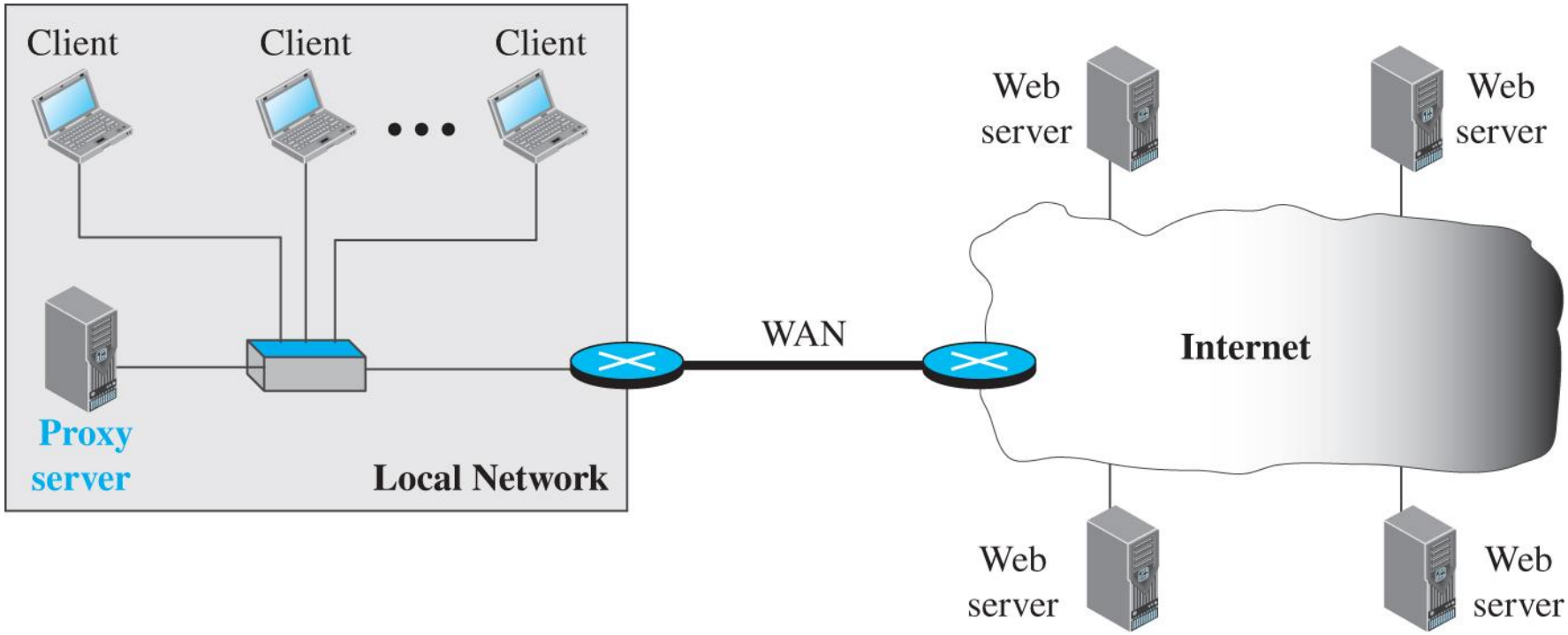


Access the text alternative for slide images.

Example 10.10

Figure 10.16 shows an example of a use of a proxy server in a local network, such as the network on a campus or in a company. The proxy server is installed in the local network. When an HTTP request is created by any of the clients (browsers), the request is first directed to the proxy server If the proxy server already has the corresponding web page, it sends the response to the client. Otherwise, the proxy server acts as a client and sends the request to the web server in the Internet. When the response is returned, the proxy server makes a copy and stores it in its cache before sending it to the requesting client.

Figure 10.16 Example of a proxy server



Access the text alternative for slide images.

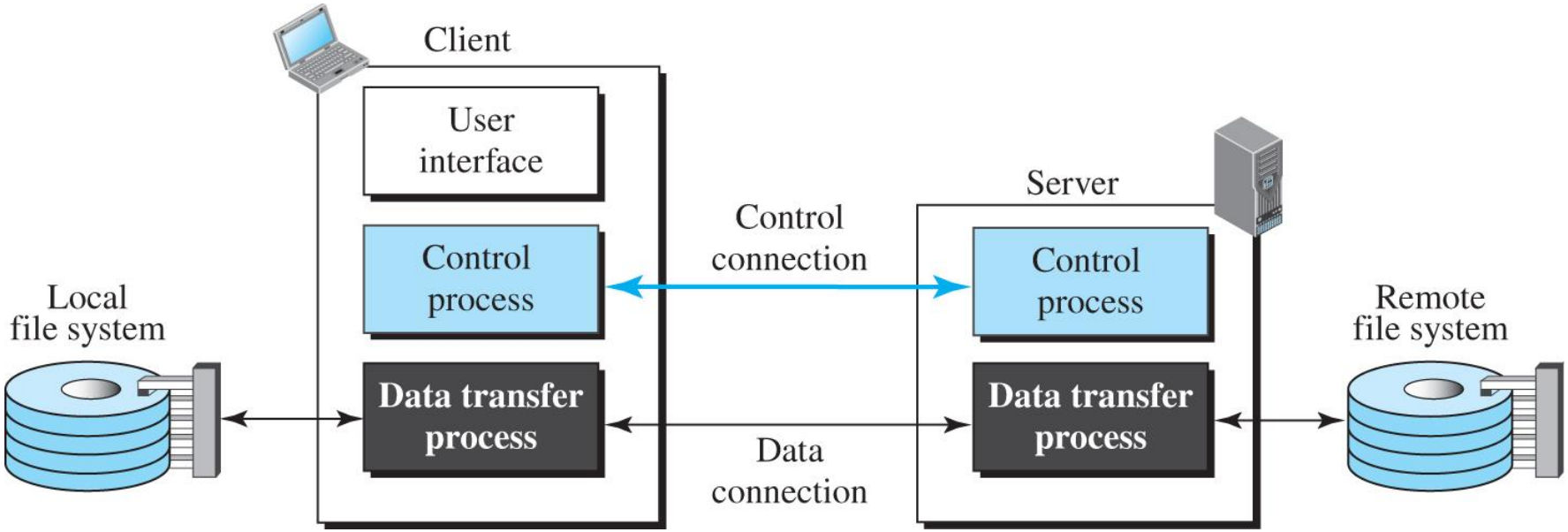
HTTP Security

HTTP per se does not provide security. However, HTTP can be run over the Secure Socket Layer (SSL). In this case, HTTP is referred to as HTTPS. HTTPS provides confidentiality, client and server authentication, and data integrity.

10.3.2 FTP

File Transfer Protocol (FTP) is the standard protocol provided by TCP/IP for copying a file from one host to another. Although transferring files from one system to another seems simple and straightforward, some problems must be dealt with first. For example, two systems may use different file name conventions. Two systems may have different ways to represent data. Two systems may have different directory structures. All of these problems have been solved by FTP in a very simple and elegant approach. Although we can transfer files using HTTP, FTP is a better choice to transfer large files or to transfer files using different formats.

Figure 10.17 FTP



Access the text alternative for slide images.

Lifetimes of Two Connections

The two connections in FTP have different lifetimes. The control connection remains connected during the entire interactive FTP session. The data connection is opened and then closed for each file transfer activity. It opens each time commands that involve transferring files are used, and it closes when the file is transferred. In other words, when a user starts an FTP session, the control connection opens. While the control connection is open, the data connection can be opened and closed multiple times if several files are transferred. FTP uses two well-known TCP ports: port 21 is used for the control connection, and port 20 is used for the data connection.

Control Connection

For control communication, FTP uses the same approach as TELNET (discussed later). It uses the NVT ASCII character set as used by TELNET. Communication is achieved through commands and responses. This simple method is adequate for the control connection because we send one command (or response) at a time. Each line is terminated with a two-character (carriage return and line feed) end-of-line token.

Table 10.4 Some FTP commands 1

Command	*Argument(s)*	*Description*
ABOR		Abort the previous command
CDUP		Change to parent directory
CWD	Directory name	Change to another directory
DELE	File name	Delete a file
LIST	Directory name	List subdirectories or files
MKD	Directory name	Create a new directory
PASS	User password	Password
PASV		Server chooses a port
PORT	port identifier	Client chooses a port
PWD		Display name of current directory
QUIT		Log out of the system
RETR	File name(s)	Retrieve files; files are transferred from server to client
RMD	Directory name	Delete a directory
RNFR	File name (old)	Identify a file to be renamed

Table 10.4 Some FTP commands 2

Command	*Argument(s)*	*Description*
RNTO	File name (new)	Rename the file
STOR	File name(s)	Store files; file(s) are transferred from client to server
STRU	**F**, **R**, or **P**	Define data organization (**F**: file, **R**: record, or **P**: page)
TYPE	**A**, **E**, **I**	Default file type (**A**: ASCII, **E**: EBCDIC, **I**: image)
USER	User ID	User information
MODE	**S**, **B**, or **C**	Define transmission mode (**S**: stream, **B**: block, or **C**: compressed)

Table 10.5 Some responses in FTP

Code	*Description*	*Code*	*Description*
125	Data connection open	**250**	Request file action OK
150	File status OK	**331**	User name OK; password is needed
200	Command OK	**425**	Cannot open data connection
220	Service ready	**450**	File action not taken; file not available
221	Service closing	**452**	Action aborted; insufficient storage
225	Data connection open	**500**	Syntax error; unrecognized command
226	Closing data connection	**501**	Syntax error in parameters or arguments
230	User login OK	**530**	User not logged in

Data Connection

The data connection uses the well-known port 20 at the server site. However, the creation of a data connection is different from the control connection. The following shows the steps:

1. *The client, not the server, issues a passive open using an ephemeral port.*

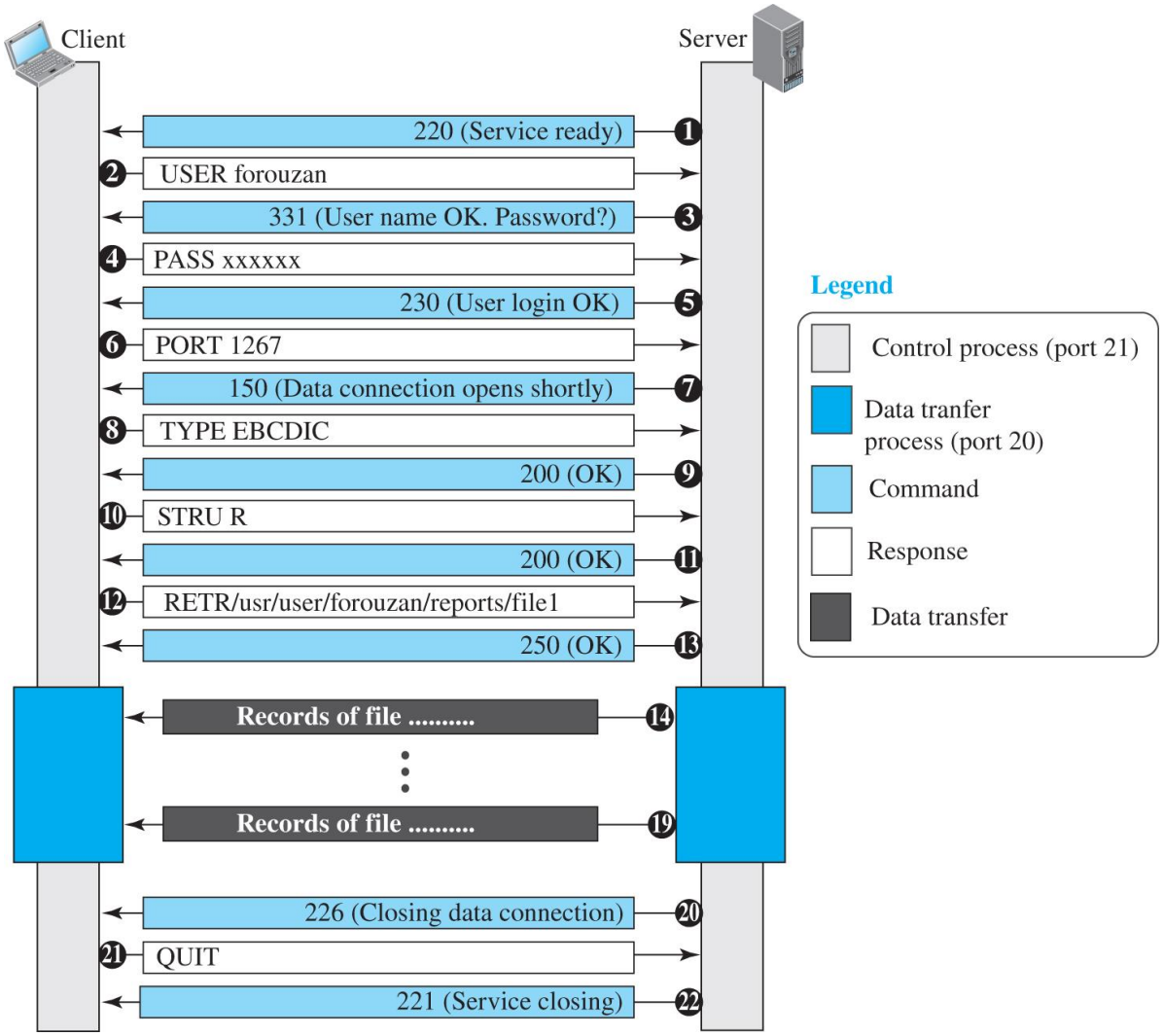
2. *Using the PORT command the client sends the port number to the server..*

3. *The server receives the port number and issues an active open using the well-known port 20 and the received ephemeral port number.*

Example 10.11

Figure 10.18 shows an example of using FTP for retrieving a file. The figure shows only one file to be transferred. The control connection remains open all the time, but the data connection is opened and closed repeatedly. We assume the file is transferred in six sections. After all records have been transferred, the server control process announces that the file transfer is done. Since the client control process has no file to retrieve, it issues the QUIT command, which causes the service connection to be closed.

Figure 10.18 Example 10.11



Access the text alternative for slide images.

Example 10.12

The following shows an actual FTP session that lists the directories.

```
$ ftp voyager.deanza.fhda.edu
Connected to voyager.deanza.fhda.edu.
220 (vsFTPd 1.2.1)
530 Please login with USER and PASS.
Name (voyager.deanza.fhda.edu:forouzan): forouzan
331 Please specify the password.
Password:*********
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
227 Entering Passive Mode (153,18,17,11,238,169)
150 Here comes the directory listing.
drwxr-xr-x 2   3027   411   4096   Sep 24   2002   business
drwxr-xr-x 2   3027   411   4096   Sep 24   2002   personal
drwxr-xr-x 2   3027   411   4096   Sep 24   2002   school
226 Directory send OK.
ftp> quit
221 Goodbye.
```

Security for FTP

The FTP protocol was designed when security was not a big issue. Although FTP requires a password, the password is sent in plaintext (unencrypted), which means it can be intercepted and used by an attacker. The data transfer connection also transfers data in plaintext, which is insecure. To be secure, one can add a Secure Socket Layer between the FTP application layer and the TCP layer. In this case FTP is called SSL-FTP. We also explore some secure file transfer applications when we discuss SSH later in the chapter.

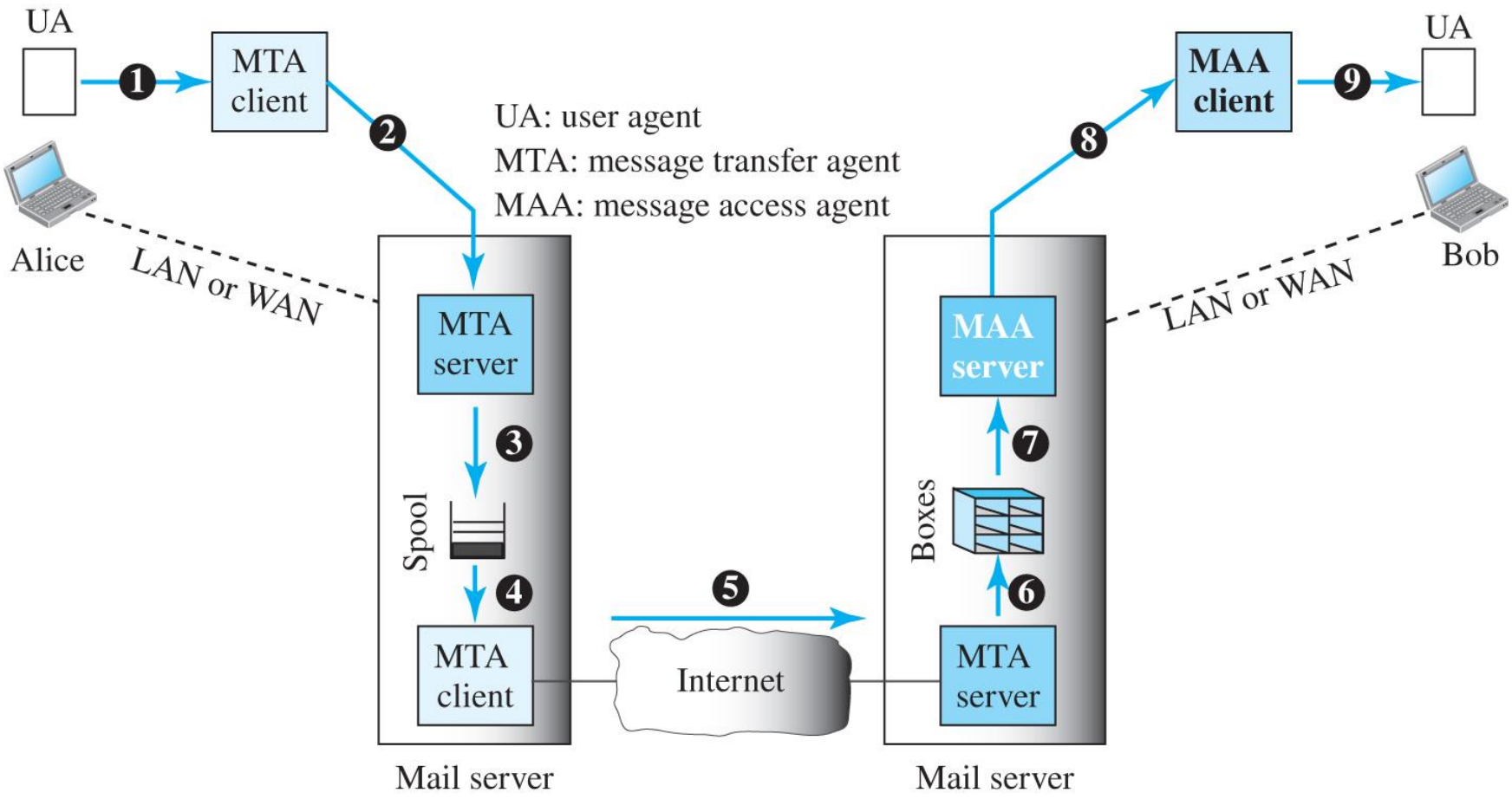
10.3.3 Electronic Mail

Electronic mail (or e-mail) allows users to exchange messages. The nature of this application, however, is different from other applications discussed so far. In an application such as HTTP or FTP, the server program is running all the time, waiting for a request from a client. When the request arrives, the server provides the service. There is a request and there is a response. In the case of electronic mail, the situation is different. First, e-mail is considered a one-way transaction. When Alice sends an e-mail to Bob, she may expect a response, but this is not a mandate.

Architecture

To explain the architecture of e-mail, we give a common scenario, as shown in Figure 10.19. Another possibility is the case in which Alice or Bob is directly connected to the corresponding mail server, in which LAN or WAN connection is not required, but this variation in the scenario does not affect our discussion.

Figure 10.19 Common scenario



User Agent

The first component of an electronic mail system is the user agent (UA). It provides service to the user to make the process of sending and receiving a message easier. A user agent is a software package (program) that composes, reads, replies to, and forwards messages. It also handles local mailboxes on the user computers.

Figure 10.20 Format of an e-mail

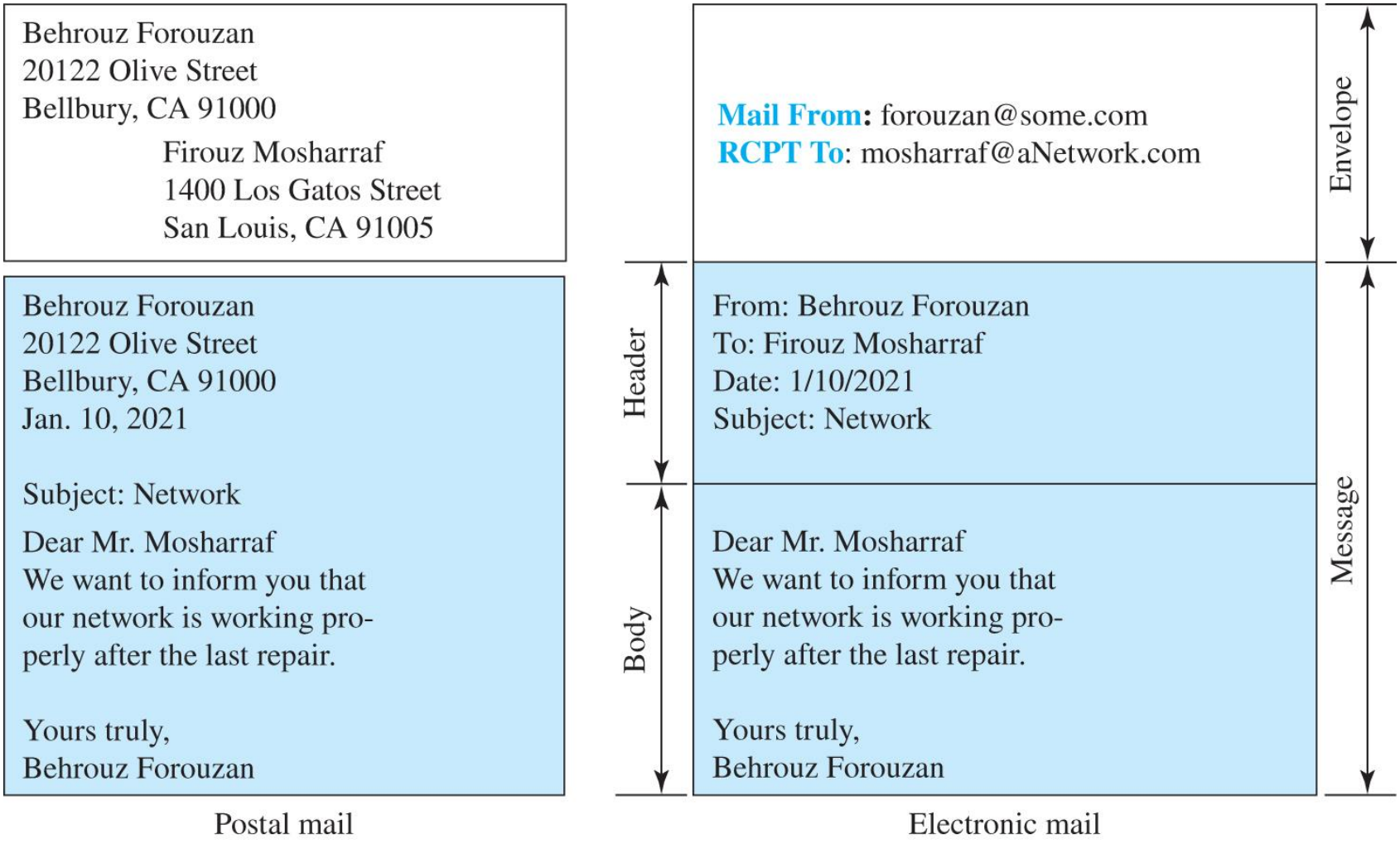
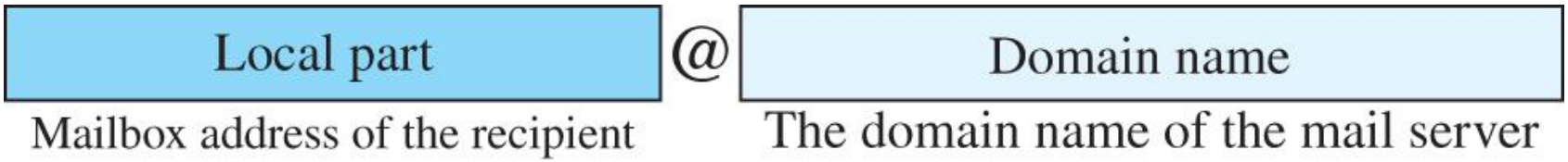


Figure 10.21 E-mail address

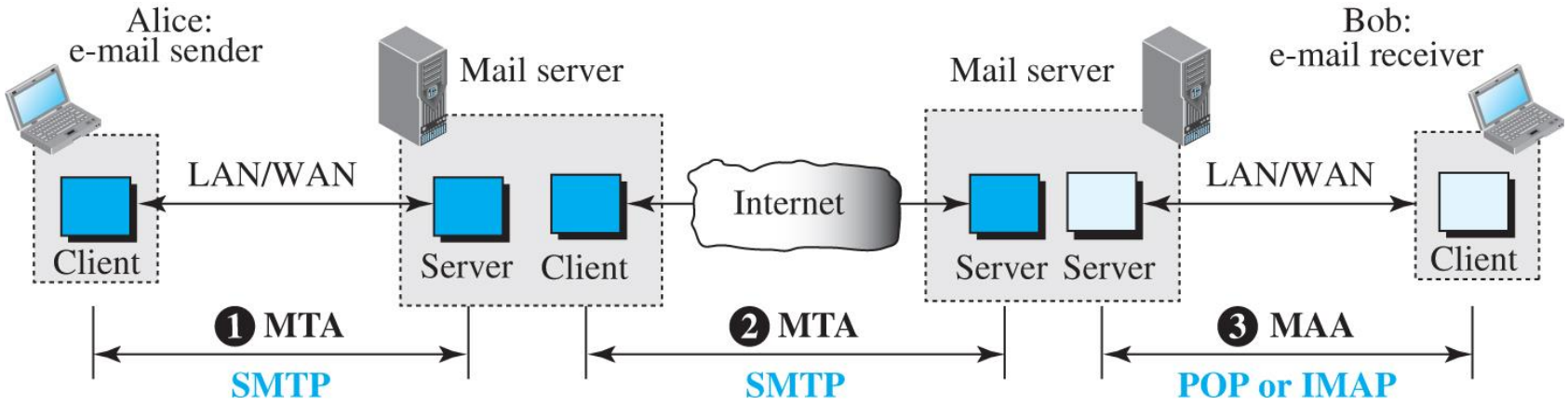


Access the text alternative for slide images.

Message Transfer Agent: SMTP

Based on the common scenario (Figure 10.19), we can say that the e-mail is one of those applications that needs three uses of client-server paradigms to accomplish its task. It is important that we distinguish these three when we are dealing with e-mail. Figure 10.22 shows these three client-server applications. We refer to the first and the second as Message Transfer Agents (MTAs), the third as Message Access Agent (MAA).

Figure 10.22 Protocols used in electronic mail



Access the text alternative for slide images.

Table 10.6 SMTP commands

Keyword	*Argument(s)*	*Description*
HELO	Sender's host name	Identifies itself
MAIL FROM	Sender of the message	Identifies the sender of the message
RCPT TO	Intended recipient	Identifies the recipient of the message
DATA	Body of the mail	Sends the actual message
QUIT		Terminates the message
RSET		Aborts the current mail transaction
VRFY	Name of recipient	Verifies the address of the recipient
NOOP		Checks the status of the recipient
TURN		Switches the sender and the recipient
EXPN	Mailing list	Asks the recipient to expand the mailing list
HELP	Command name	Asks the recipient to send information about the command sent as the argument
SEND FROM	Intended recipient	Specifies that the mail be delivered only to the terminal of the recipient, and not to the mailbox
SMOL FROM	Intended recipient	Specifies that the mail be delivered to the terminal or the mailbox of the recipient
SMAL FROM	Intended recipient	Specifies that the mail be delivered to the terminal and the mailbox of the recipient

Table 10.7 SMTP responses 1

Code	*Description*
Positive Completion Reply	
211	System status or help reply
214	Help message
220	Service ready
221	Service closing transmission channel
250	Request command completed
251	User not local; the message will be forwarded
Positive Intermediate Reply	
354	Start mail input
Transient Negative Completion Reply	
421	Service not available
450	Mailbox not available
451	Command aborted: local error
452	Command aborted; insufficient storage

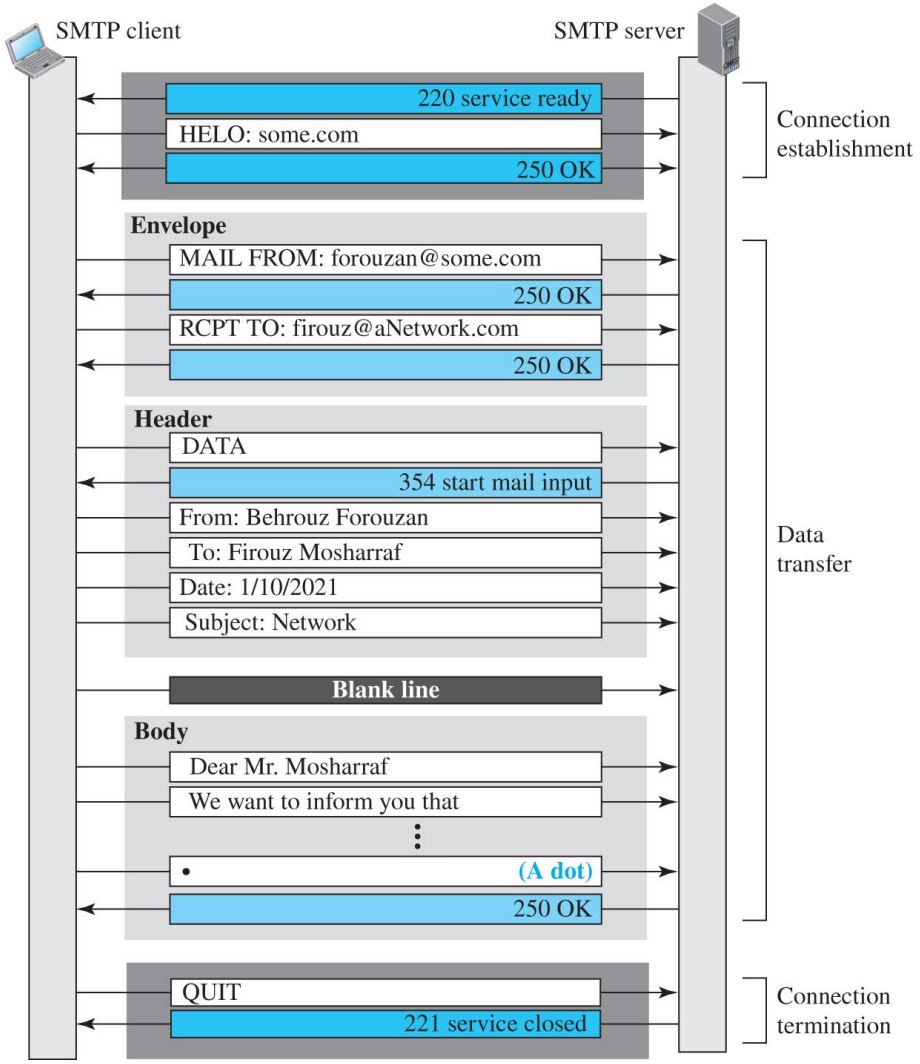
Table 10.7 SMTP responses 2

	Permanent Negative Completion Reply
500	Syntax error; unrecognized command
501	Syntax error in parameters or arguments
502	Command not implemented
503	Bad sequence of commands
504	Command temporarily not implemented
550	Command is not executed; mailbox unavailable
551	User not local
552	Requested action aborted; exceeded storage location
553	Requested action not taken; mailbox name not allowed
554	Transaction failed

Example 10.13

To show the three mail transfer phases, we show all of the steps described above using the information depicted in Figure 10.23. In the figure, we have separated the messages related to the envelope, header, and body in the data transfer section. Note that the steps in this figure are repeated two times in each e-mail transfer: once from the e-mail sender to the local mail server and once from the local mail server to the remote mail server. The local mail server, after receiving the whole e-mail message, may spool it and send it to the remote mail server at another time.

Figure 10.23 Example 10.13

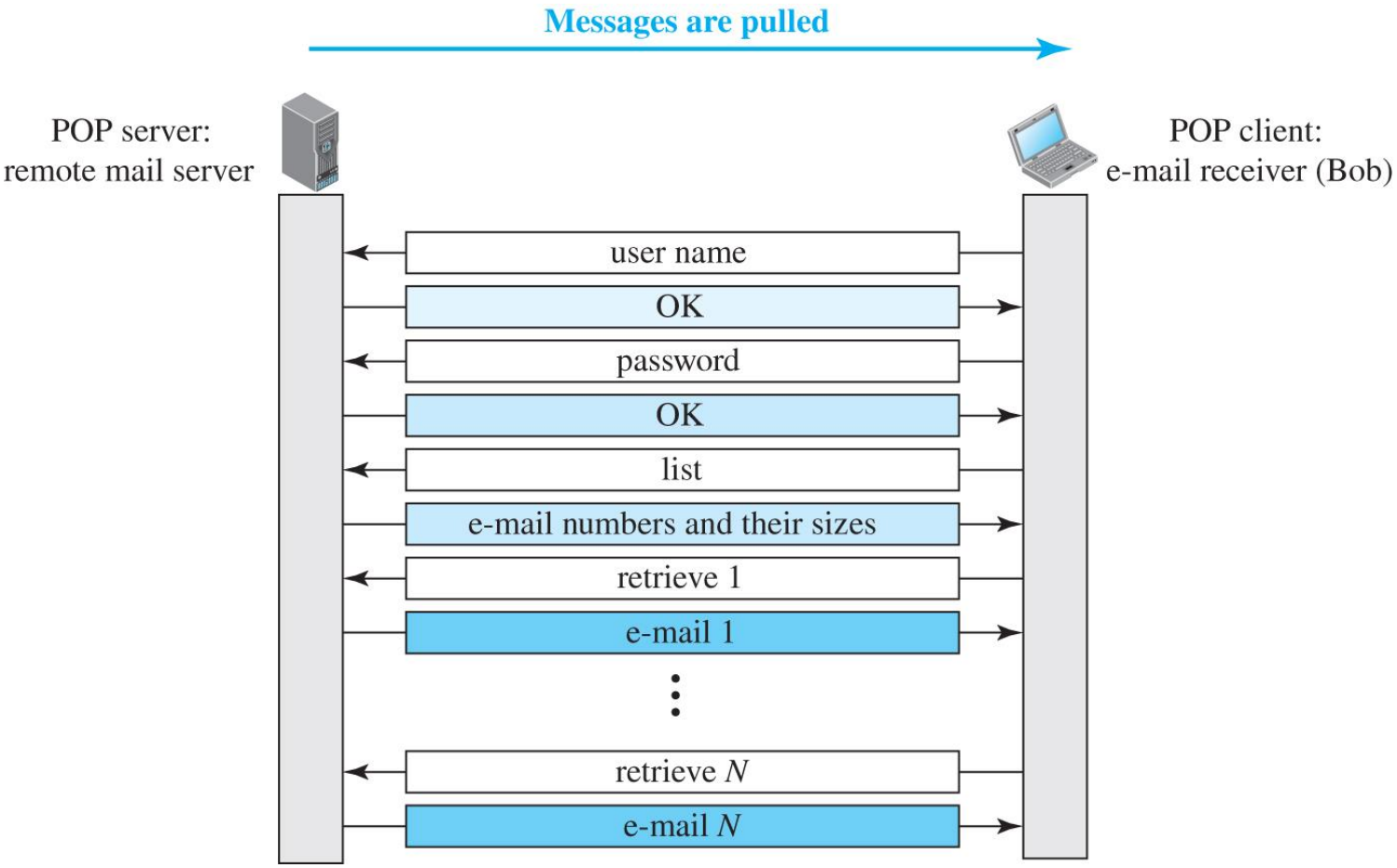


Access the text alternative for slide images.

Message Access Agent: POP and IMAP

The first and second stages of mail delivery use SMTP. However, SMTP is not involved in the third stage because SMTP is a push protocol; it pushes the message from the client to the server. In other words, the direction of the bulk data (messages) is from the client to the server. On the other hand, the third stage needs a pull protocol; the client must pull messages from the server. The direction of the bulk data is from the server to the client. The third stage uses a message access agent.

Figure 10.24 POP3



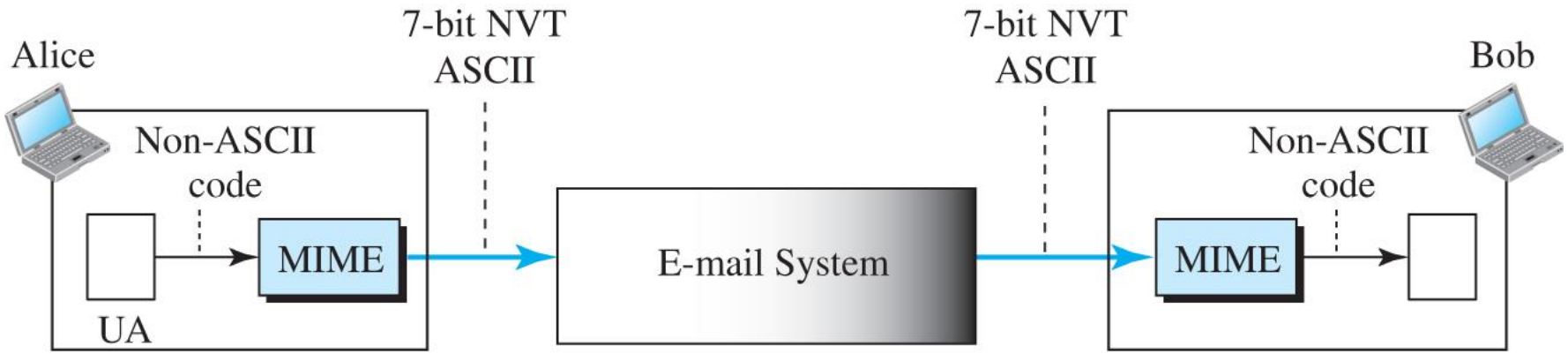
Access the text alternative for slide images.

MIME

- *Electronic mail has a simple structure. Its simplicity, however, comes with a price. It can send messages only in NVT 7-bit ASCII format. In other words, it has some limitations. It cannot be used for languages other than English (such as French, German, Hebrew, Russian, Chinese, and Japanese). Also, it cannot be used to send binary files or video or audio data.*

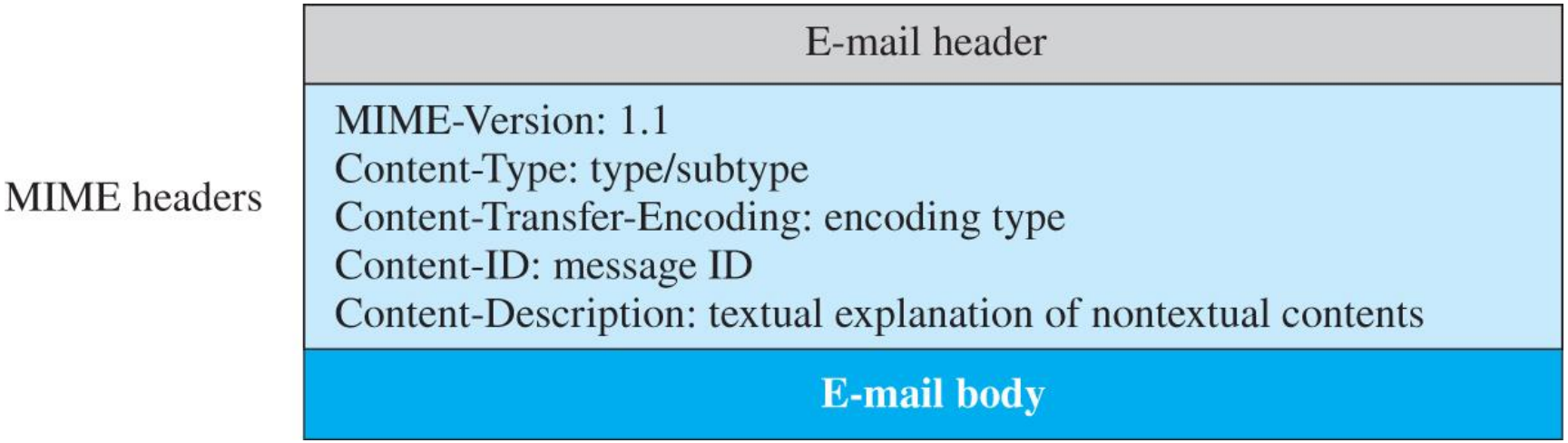
- *Multipurpose Internet Mail Extensions (MIME) is a supplementary protocol that allows non-ASCII data to be sent through e-mail.*

Figure 10.25 MIME



Access the text alternative for slide images.

Figure 10.26 MIME header



Access the text alternative for slide images.

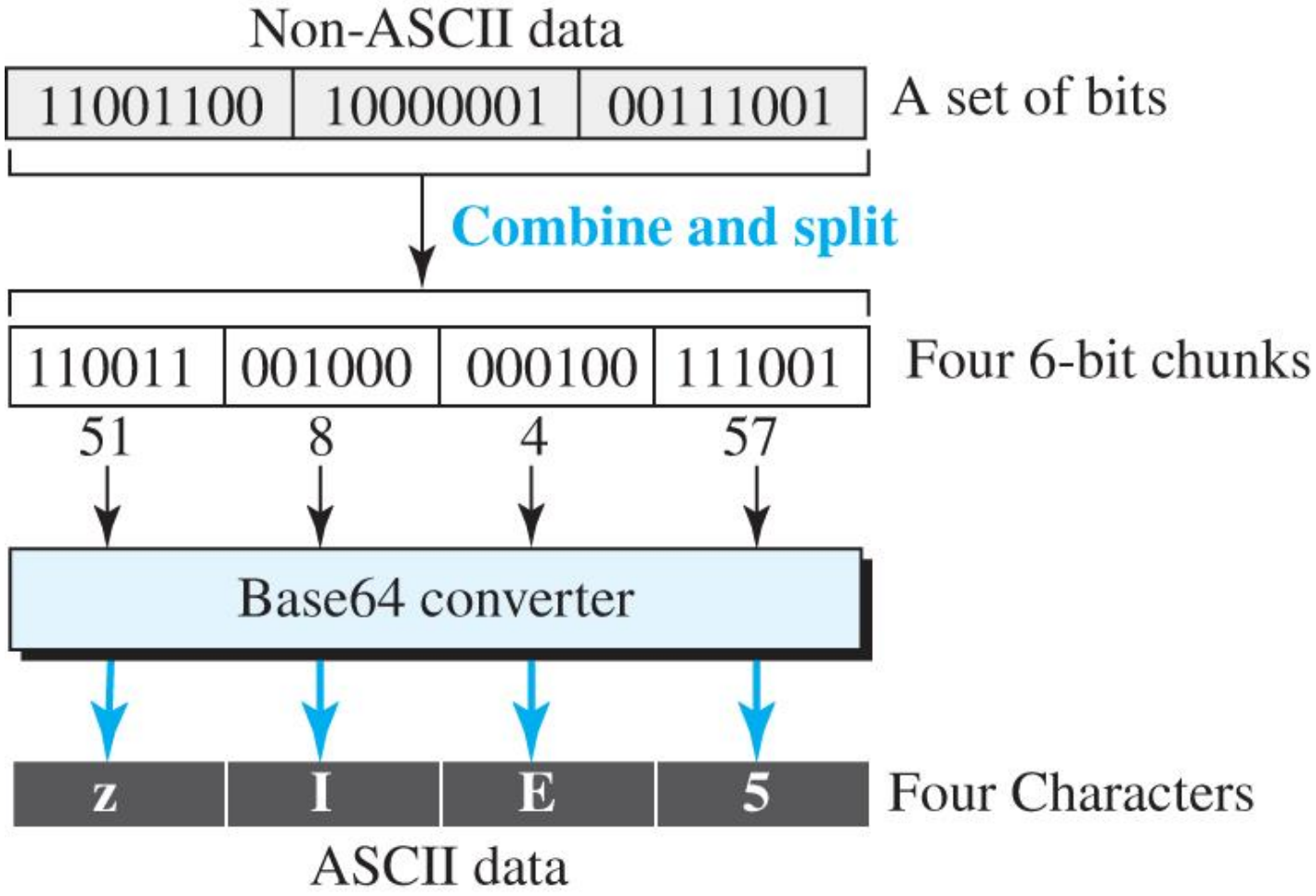
Table 10.8 Data types and subtypes in MIME

Type	*Subtype*	*Description*
Text	Plain	Unformatted
	HTML	HTML format (see Appendix C)
Multipart	Mixed	Body contains ordered parts of different data types
	Parallel	Same as above, but no order
	Digest	Similar to Mixed, but the default is message/RFC822
	Alternative	Parts are different versions of the same message
Message	RFC822	Body is an encapsulated message
	Partial	Body is a fragment of a bigger message
	External-Body	Body is a reference to another message
Image	JPEG	Image is in JPEG format
	GIF	Image is in GIF format
Video	MPEG	Video is in MPEG format
Audio	Basic	Single channel encoding of voice at 8 kHz
Application	PostScript	Adobe PostScript
	Octet-stream	General binary data (8-bit bytes)

Table 10.9 Methods for content-transfer-encoding

Type	*Description*
7-bit	NVT ASCII characters with each line less than 1000 characters
8-bit	Non-ASCII characters with each line less than 1000 characters
Binary	Non-ASCII characters with unlimited-length lines
Base64	6-bit blocks of data encoded into 8-bit ASCII characters
Quoted-printable	Non-ASCII characters encoded as an equal sign plus an ASCII code

Figure 10.27 Base64 conversion

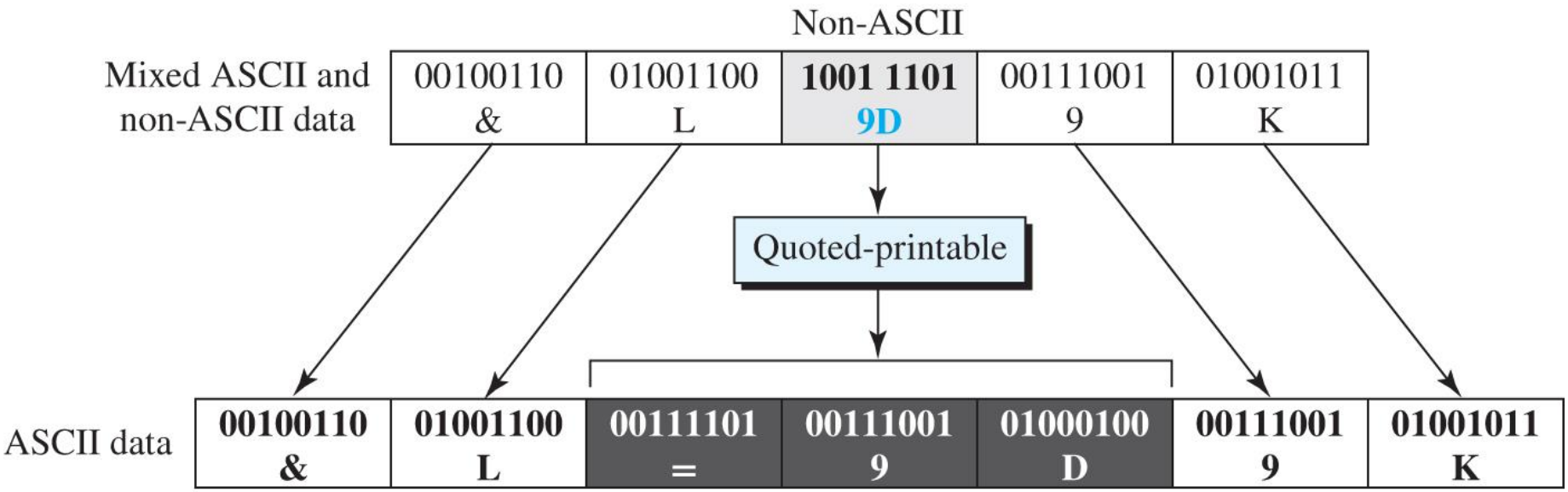


Access the text alternative for slide images.

Table 10.10 Base64 converting table

Value	*Code*	*Value*	*Code*	*Value*	*Code*	*Value*	*Code*	*Value*	*Code*	*Value*	*Code*
0	**A**	11	**L**	22	**W**	33	**h**	44	**s**	55	**3**
1	**B**	12	**M**	23	**X**	34	**i**	45	**t**	56	**4**
2	**C**	13	**N**	24	**Y**	35	**j**	46	**u**	57	**5**
3	**D**	14	**O**	25	**Z**	36	**k**	47	**v**	58	**6**
4	**E**	15	**P**	26	**a**	37	**l**	48	**w**	59	**7**
5	**F**	16	**Q**	27	**b**	38	**m**	49	**x**	60	**8**
6	**G**	17	**R**	28	**c**	39	**n**	50	**y**	61	**9**
7	**H**	18	**S**	29	**d**	40	**o**	51	**z**	62	+
8	**I**	19	**T**	30	**e**	41	**p**	52	**0**	63	/
9	**J**	20	**U**	31	**f**	42	**q**	53	**1**		
10	**K**	21	**V**	32	**g**	43	**r**	54	**2**		

Figure 10.28 Quoted-printable

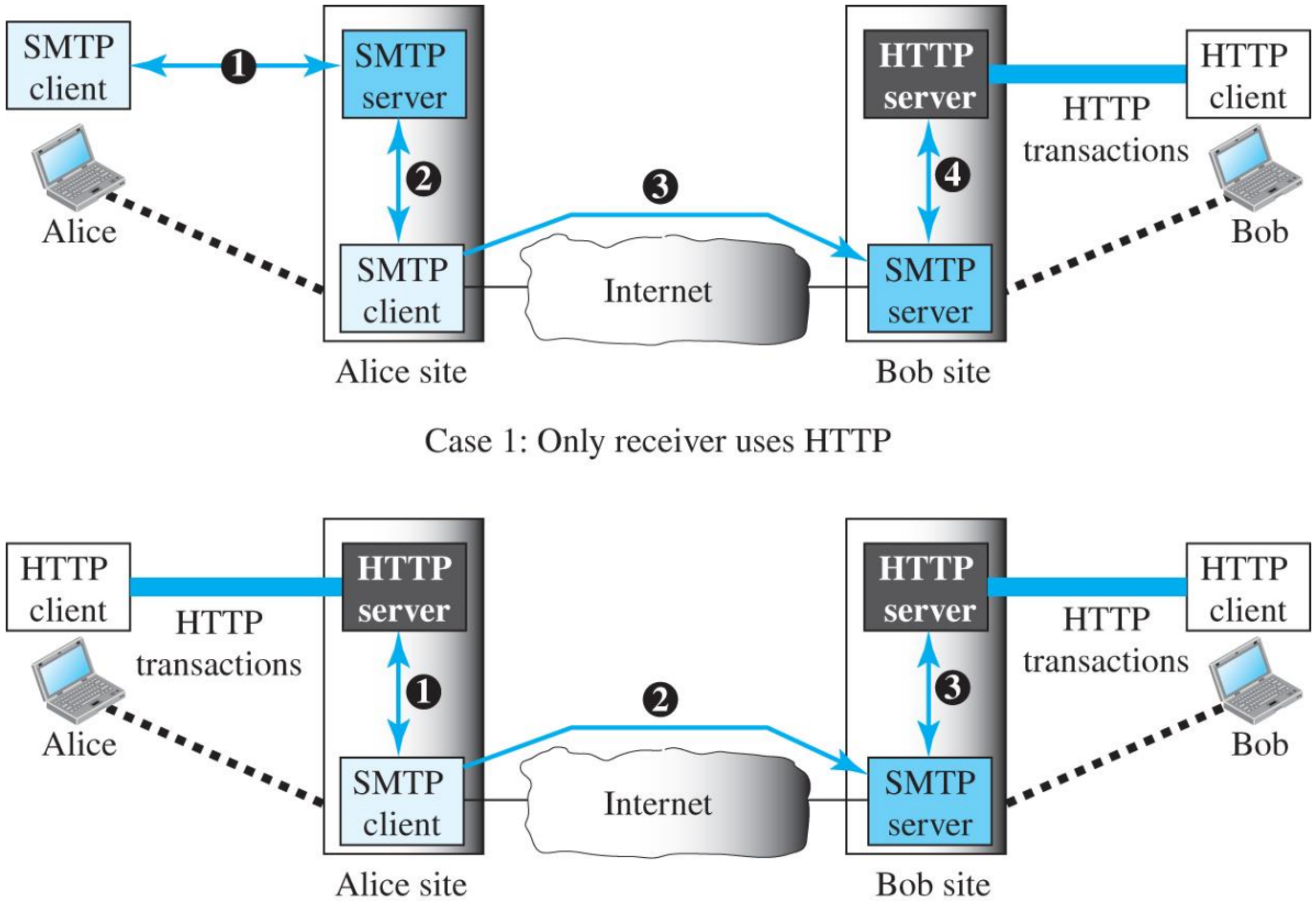


Access the text alternative for slide images.

Web-Based Mail

E-mail is such a common application that some websites today provide this service to anyone who accesses the site. Three common sites are Hotmail, Yahoo, and Google mail. The idea is very simple. Figure 10.29 shows two cases.

Figure 10.29 Web-based e-mail, cases I and II



Access the text alternative for slide images.

E-Mail Security

The protocol discussed in this chapter does not provide any security provisions per se. However, e-mail exchanges can be secured using two application-layer securities designed in particular for e-mail systems. Two of these protocols, Pretty Good Privacy (PGP) and Secure/Multipurpose Internet Mail Extensions (S/MIME), are discussed in Chapter 13 after we have discussed basic network security.

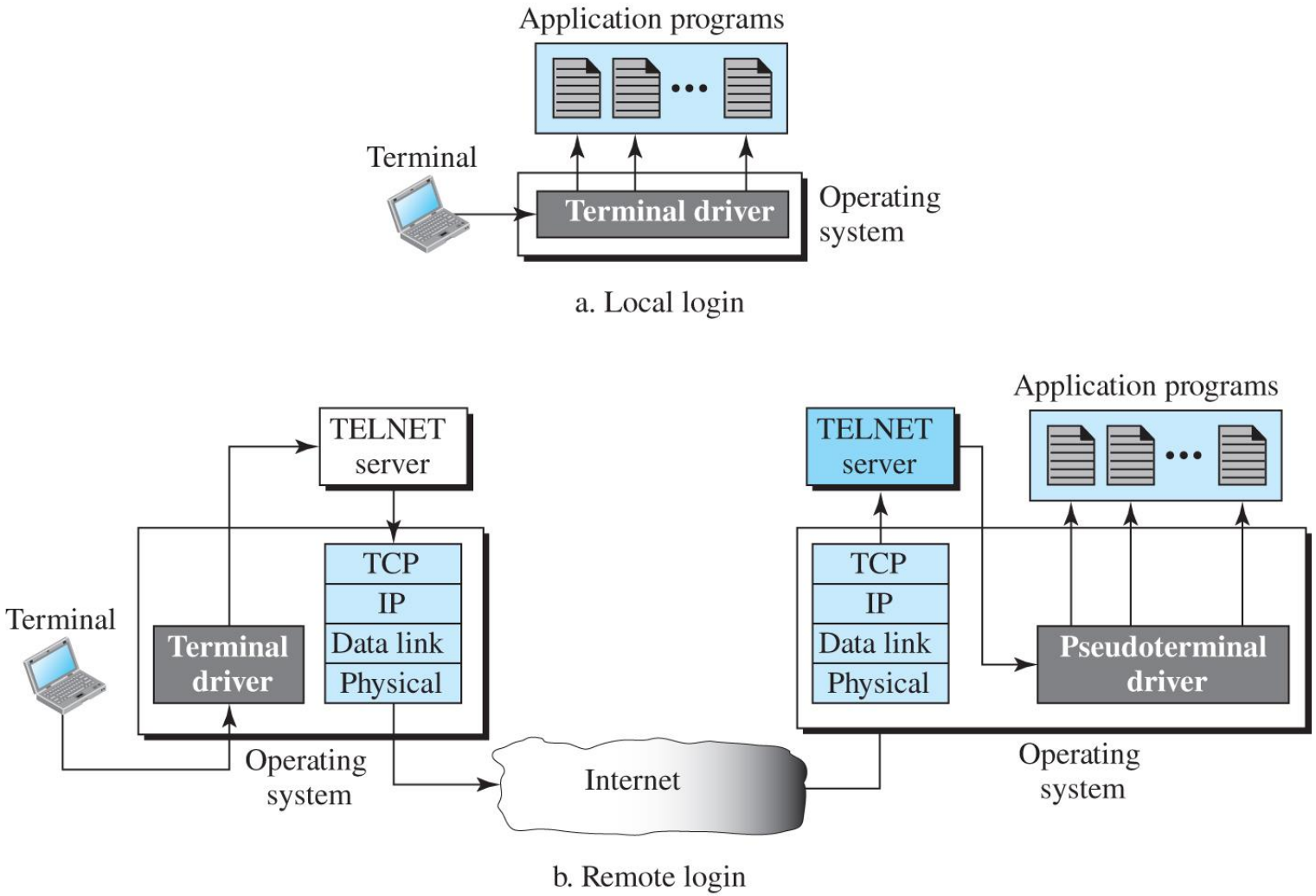
10.3.4 TELNET

A server program can provide a specific service to its corresponding client program. However, it is impossible to have a client/server pair for each type of service we need; the number of servers soon becomes intractable. One of the original remote logging protocols is TELNET, which is an abbreviation for TErminaL NETwork. Although TELNET requires a logging name and password, it is vulnerable to hacking because it sends all data including the password in plaintext (not encrypted).

Local versus Remote Logging

We first discuss the concept of local and remote logging as shown in Figure 10.30.

Figure 10.30 Local versus remote logging

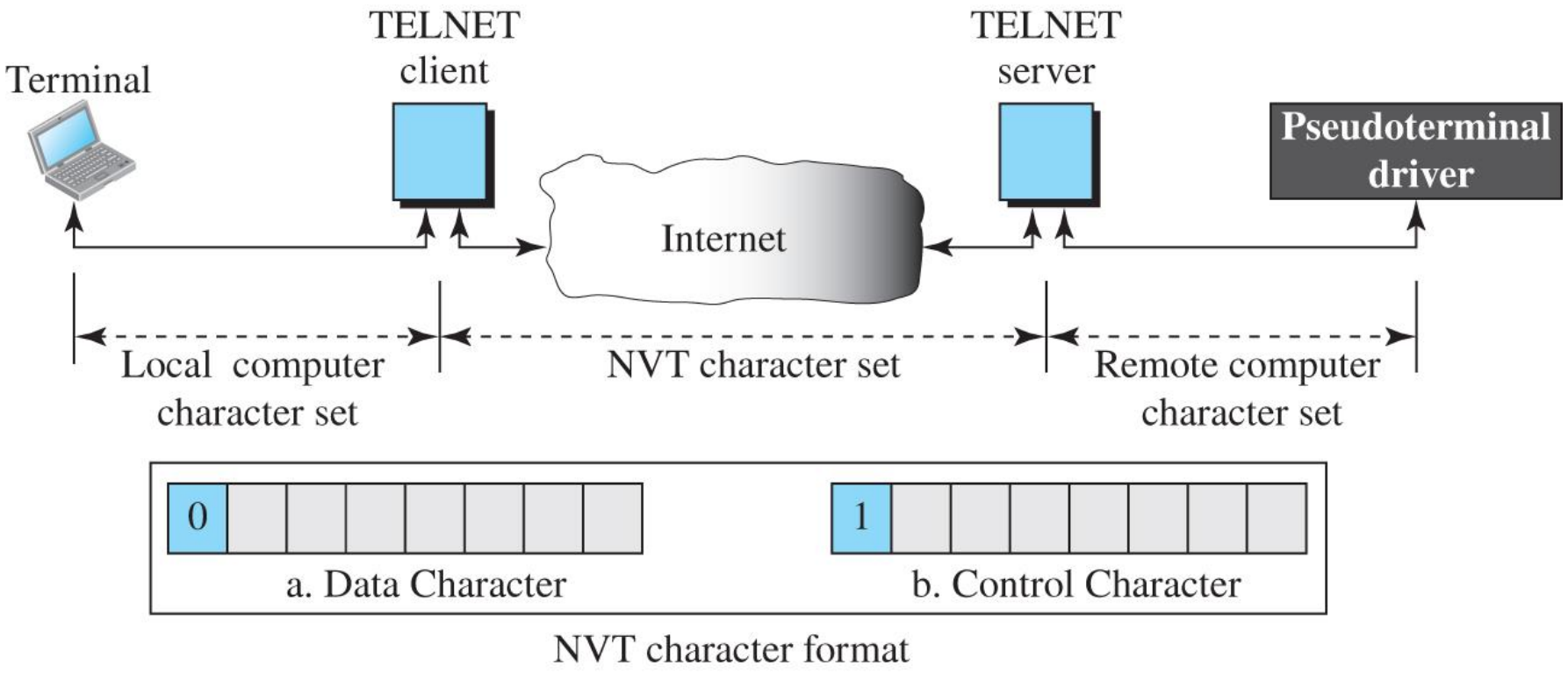


Access the text alternative for slide images.

Network Virtual Terminal (NVT)

The mechanism to access a remote computer is complex. This is because every computer and its operating system accepts a special combination of characters as tokens. For example, the end-of-file token in a computer running the DOS operating system is Ctrl+z, while the UNIX operating system recognizes Ctrl+d.

Figure 10.31 Concept of NVT



Access the text alternative for slide images.

Operation

TELNET lets the client and server negotiate options before or during the use of the service. Options are extra features available to a user with a more sophisticated terminal. Users with simpler terminals can use default features.

User Interface

The operating system (UNIX, for example) defines an interface with user-friendly commands. An example of such a set of commands can be found in Table 10.11.

Table 10.11 Examples of interface commands

Command	*Meaning*	*Command*	*Meaning*
open	Connect to a remote computer	**set**	Set the operating parameters
close	Close the connection	**status**	Display the status information
display	Show the operating parameters	**send**	Send special characters
mode	Change to line or character mode	**quit**	Exit TELNET

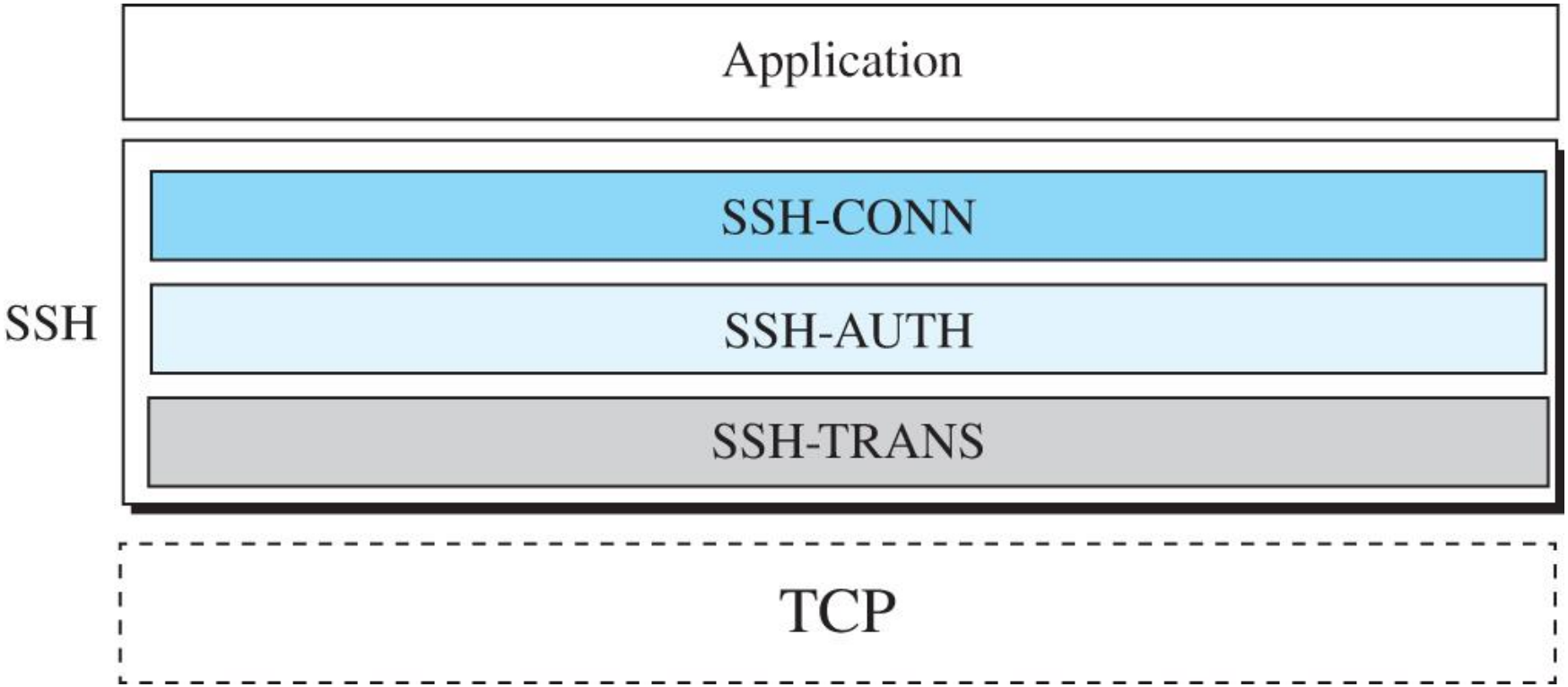
10.3.5 Secure Shell (SSH)

Although Secure Shell (SSH) is a secure application program that can be used today for several purposes such as remote logging and file transfer, it was originally designed to replace TELNET. There are two versions of SSH: SSH-1 and SSH-2, which are totally incompatible. The first version, SSH-1, is now deprecated because of security flaws in it. In this section, we discuss only SSH-2.

Components

SSH is an application-layer protocol with three components, as shown in Figure 10.32.

Figure 10.32 Components of SSH



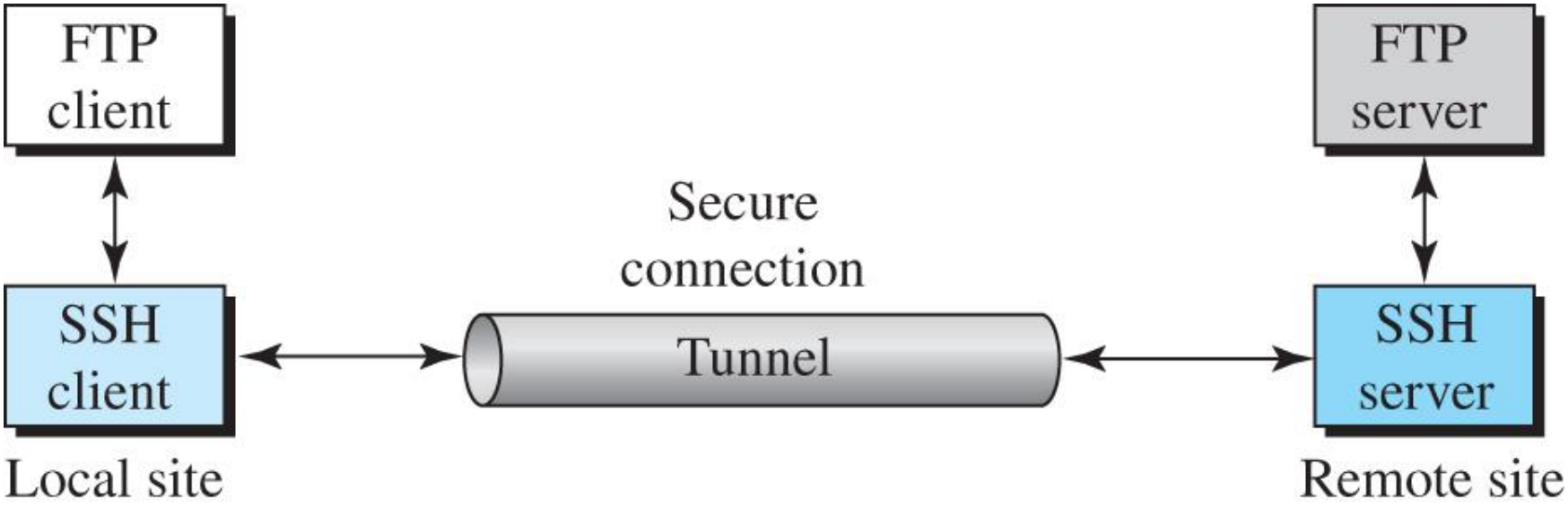
Applications 1

Although SSH is often thought of as a replacement for TELNET, SSH is, in fact, a general-purpose protocol that provides a secure connection between a client and server.

Port Forwarding

One of the interesting services provided by the SSH protocol is port forwarding. We can use the secured channels available in SSH to access an application program that does not provide security services. Applications such as TELNET and Simple Mail Transfer Protocol (SMTP), which are discussed later, can use the services of the SSH port forwarding mechanism. The SSH port forwarding mechanism creates a tunnel through which the messages belonging to other protocols can travel. For this reason, this mechanism is sometimes referred to as SSH tunneling. Figure 10.33 shows the concept of port forwarding for securing the FTP application.

Figure 10.33 Port forwarding

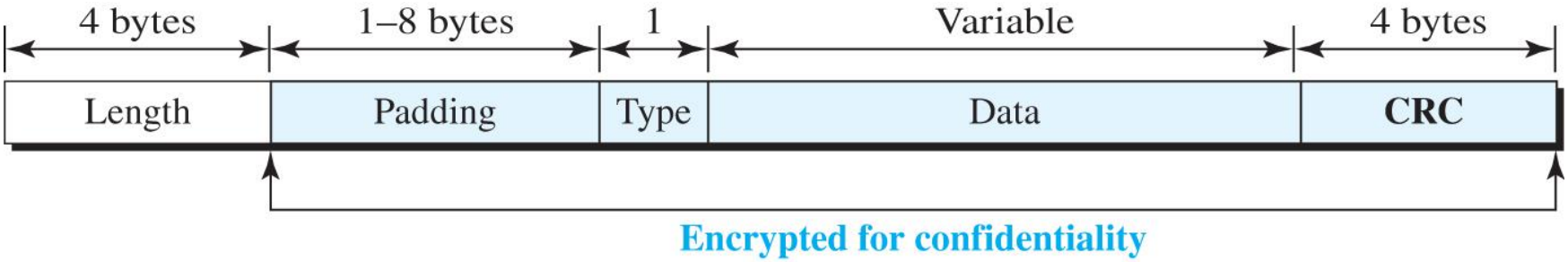


Access the text alternative for slide images.

Format of the SSH Packets

Figure 10.34 shows the format of packets used by the SSH protocols.

Figure 10.34 SSH packet format

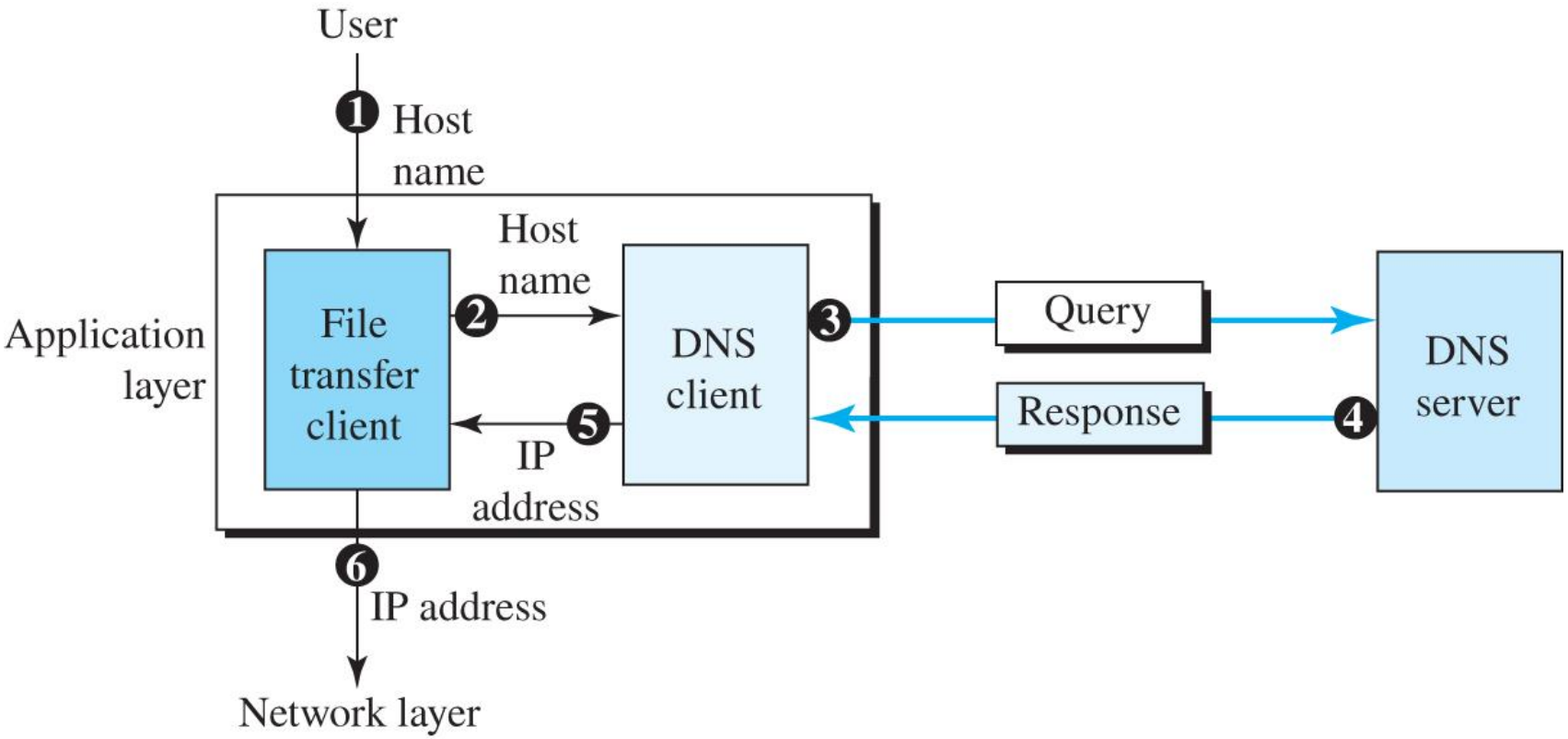


Access the text alternative for slide images.

10.3.6 Domain Name System (DNS)

The last client-server application program we discuss has been designed to help other application programs. The Internet needs to have a directory system that can map a name to an address. This is analogous to the telephone network. Figure 10.35 shows how TCP/IP uses a DNS client and a DNS server to map a name to an address.

Figure 10.35 Purpose of DNS

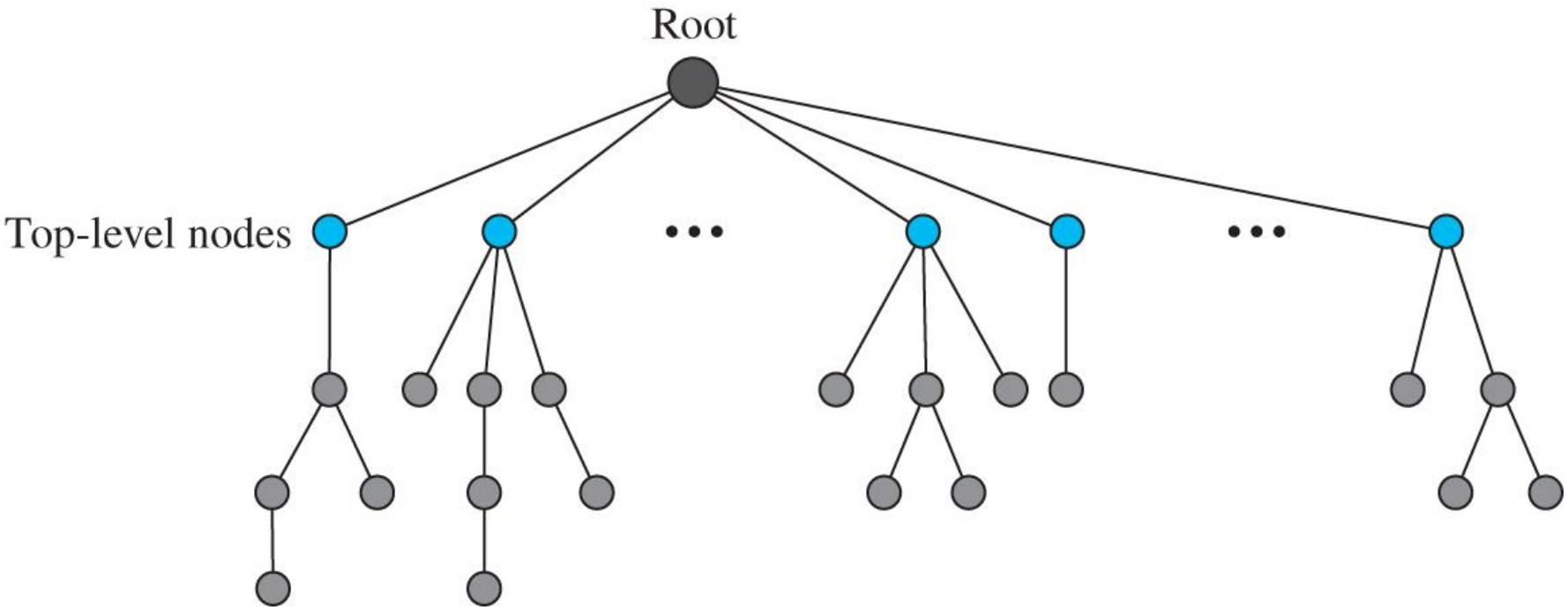


Access the text alternative for slide images.

Name Space

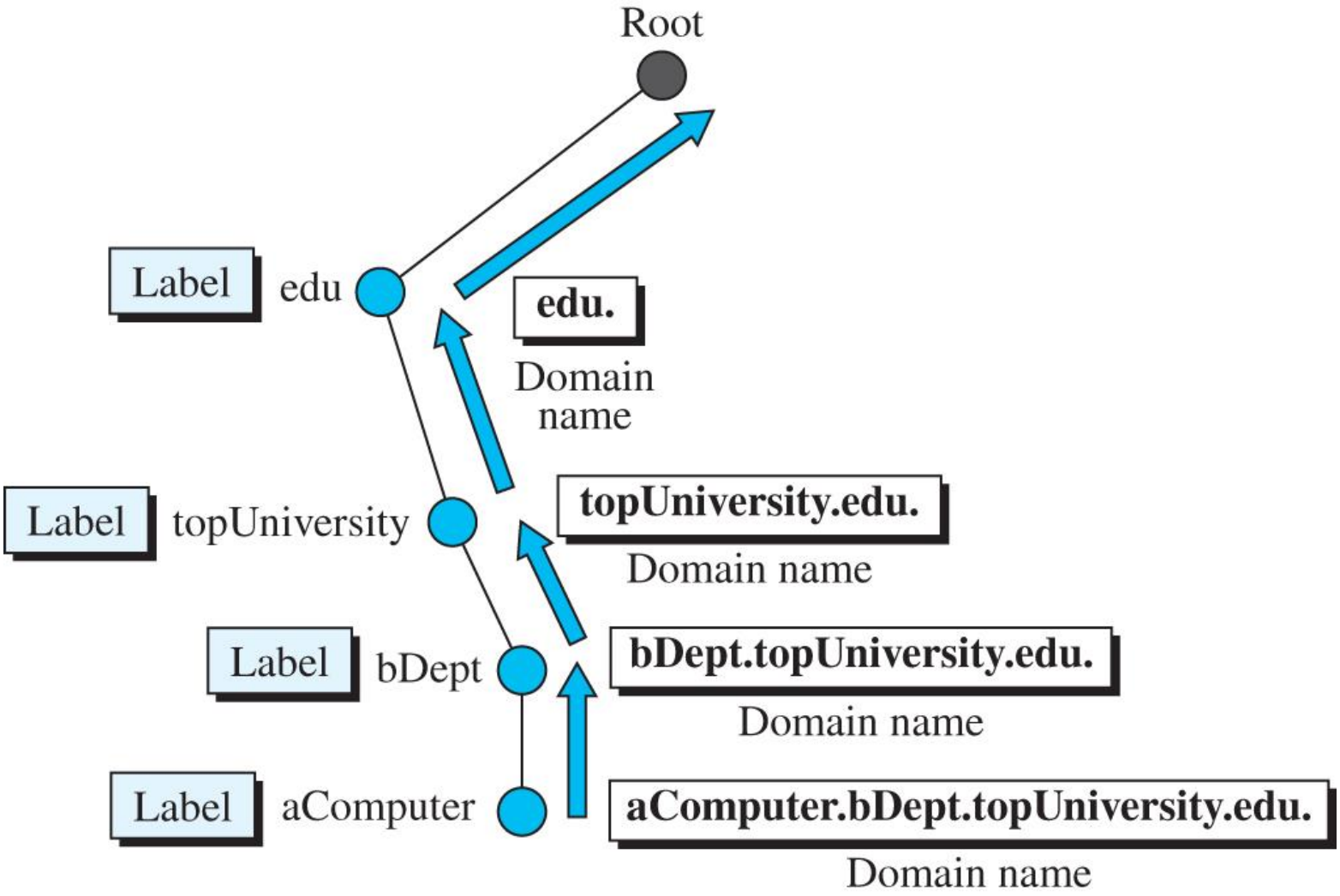
To be unambiguous, the names assigned to machines must be carefully selected from a name space with complete control over the binding between the names and IP addresses. In other words, the names must be unique because the addresses are unique. A name space that maps each address to a unique name can be organized in two ways: flat or hierarchical.

Figure 10.36 Domain name space



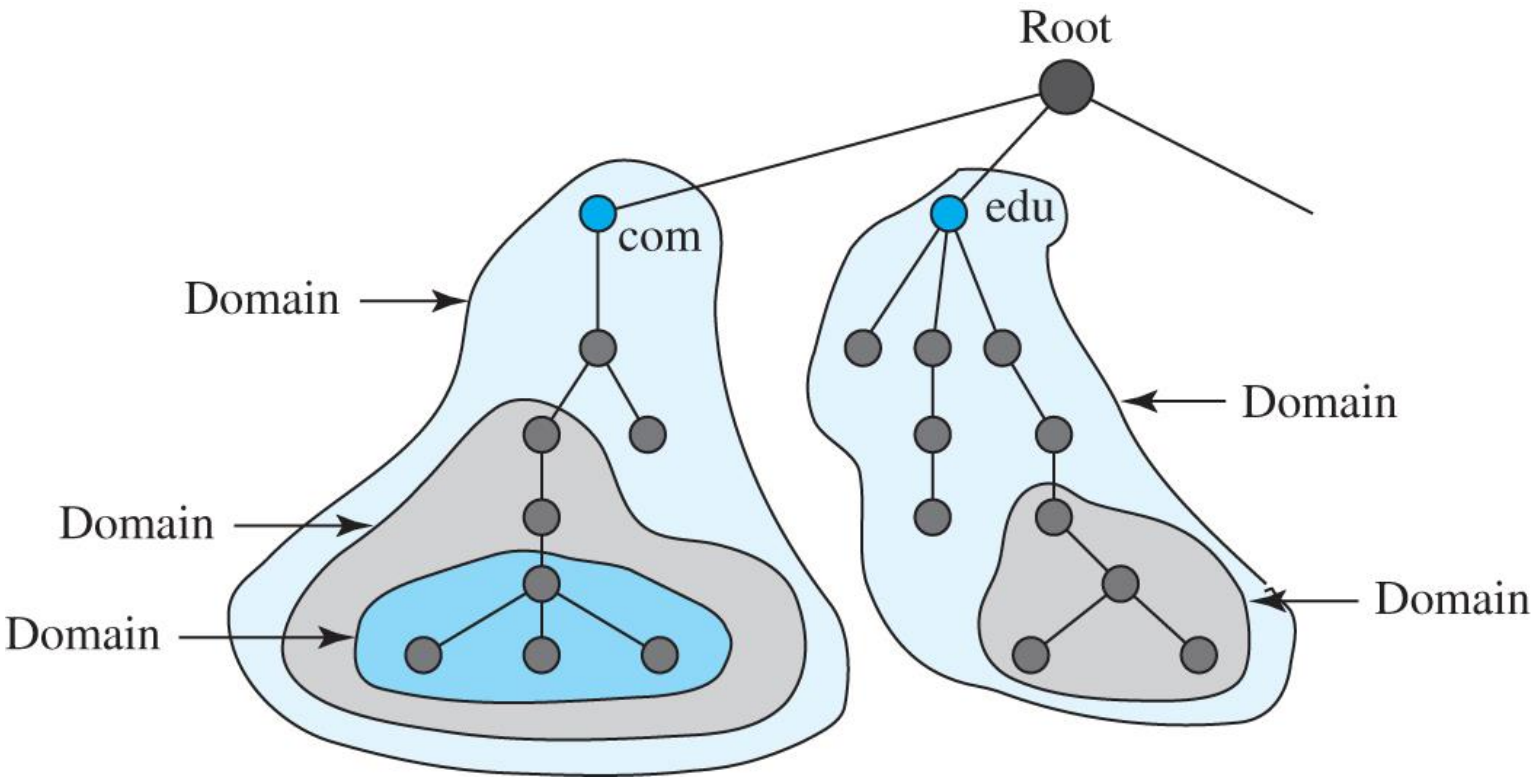
Access the text alternative for slide images.

Figure 10.37 Domain names and labels



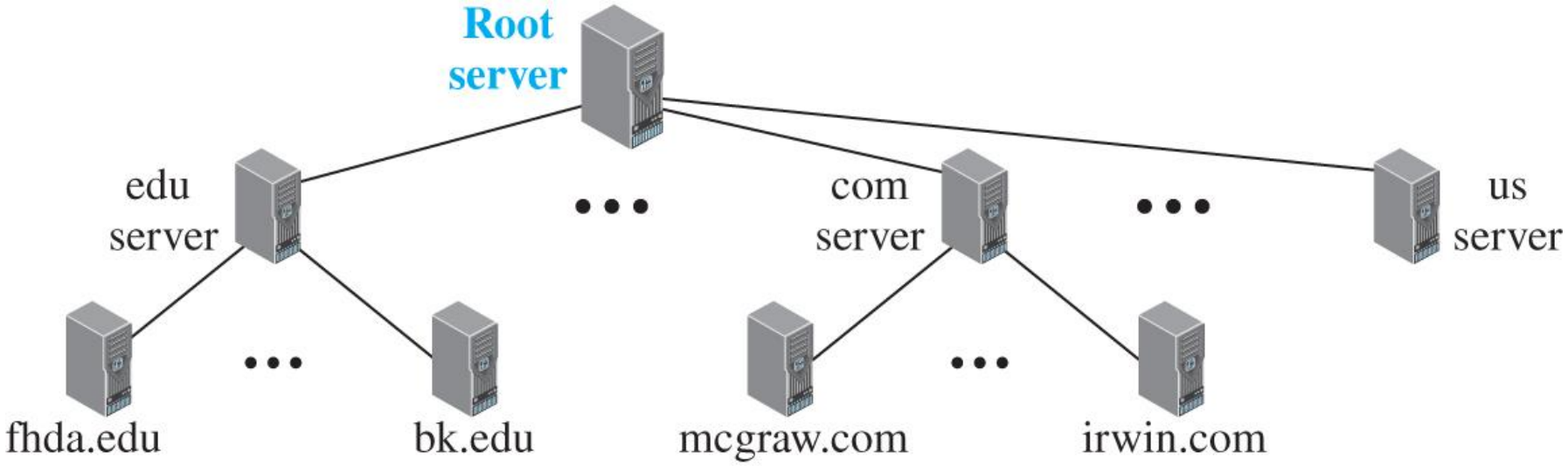
Access the text alternative for slide images.

Figure 10.38 Domains



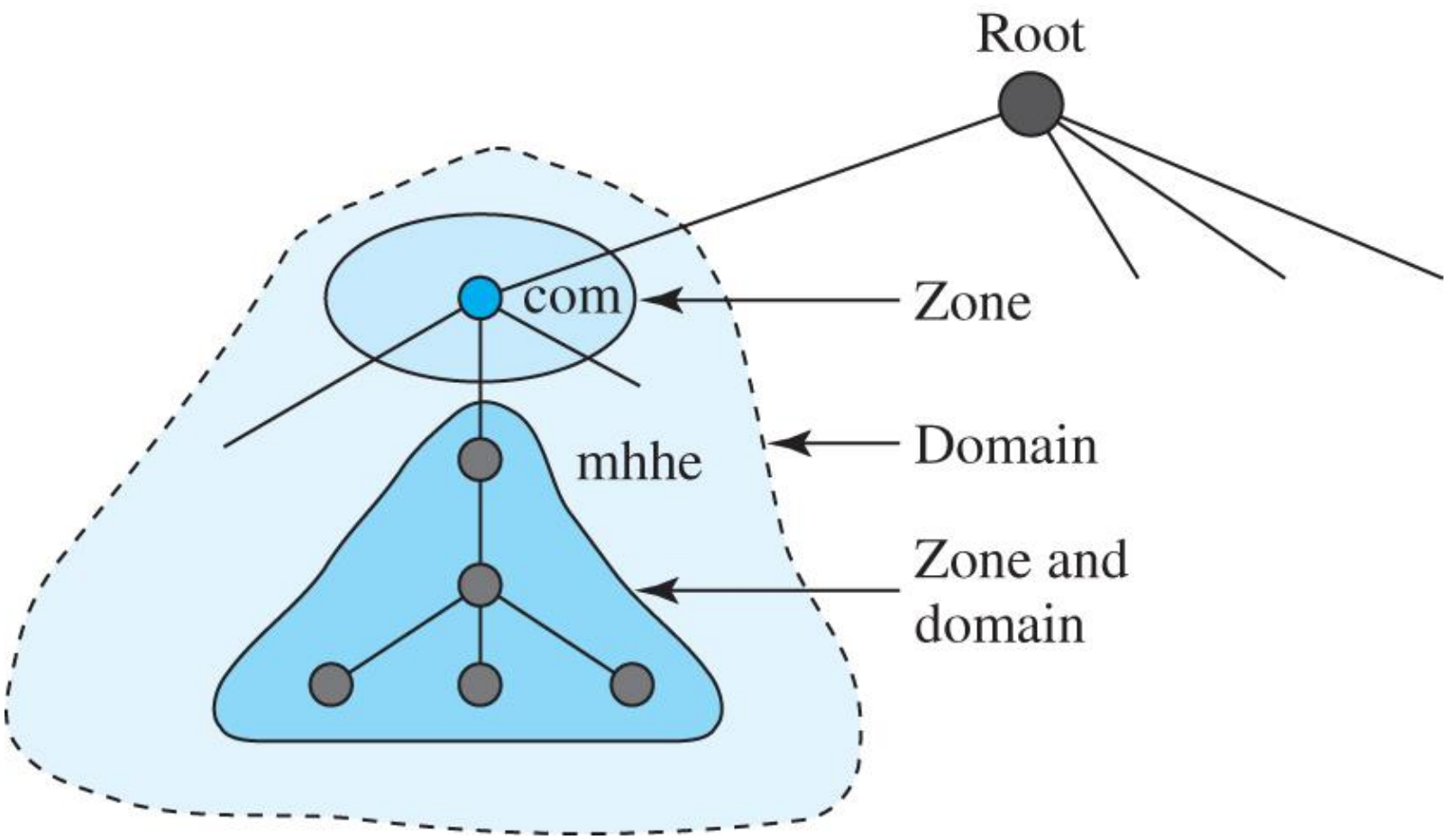
Access the text alternative for slide images.

Figure 10.39 Hierarchy of name servers



Access the text alternative for slide images.

Figure 10.40 Zone

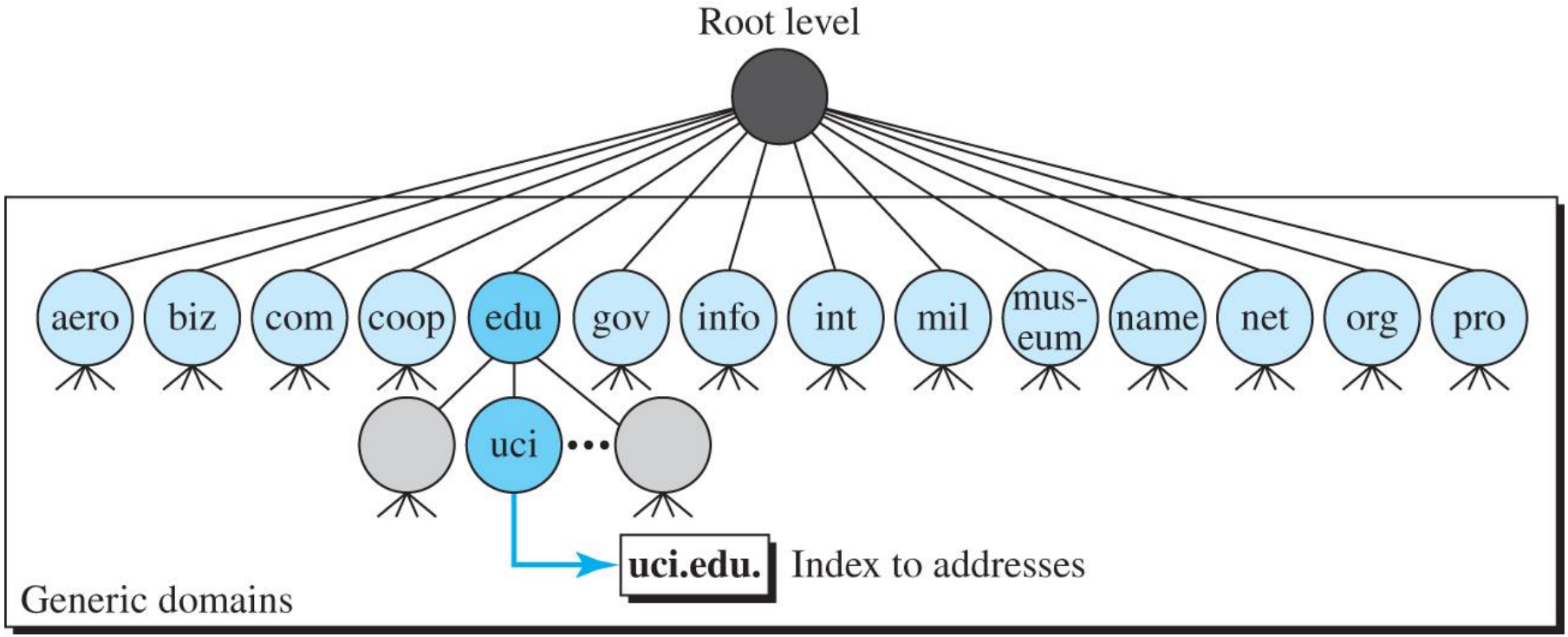


Access the text alternative for slide images.

DNS in the Internet

DNS is a protocol that can be used in different platforms. In the Internet, the domain name space (tree) was originally divided into three different sections: generic domains, country domains, and the inverse domains. However, due to the rapid growth of the Internet, it became extremely difficult to keep track of the inverse domains, which could be used to find the name of a host when given the IP address. The inverse domains are now deprecated. We, therefore, concentrate on the first two.

Figure 10.41 Generic domains

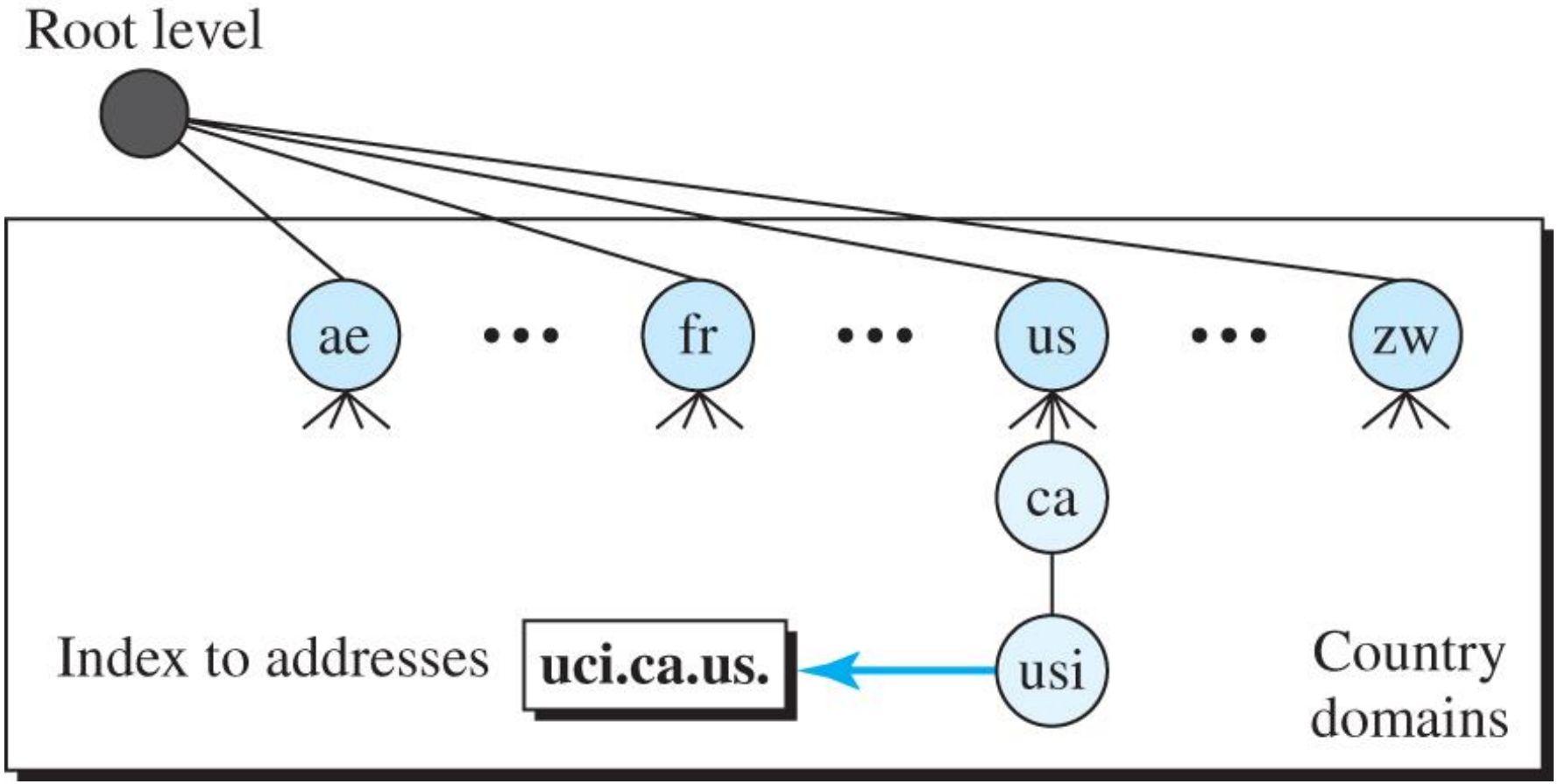


Access the text alternative for slide images.

Table 10.12 Generic domain labels

Label	*Description*	*Label*	*Description*
aero	Airlines and aerospace	**int**	International organizations
biz	Businesses or firms	**mil**	Military groups
com	Commercial organizations	**museum**	Museum
coop	Cooperative organizations	**name**	Personal names (individuals)
edu	Educational institutions	**net**	Network support centers
gov	Government institutions	**org**	Nonprofit organizations
info	Information service providers	**pro**	Professional organizations

Figure 10.42 Country domains

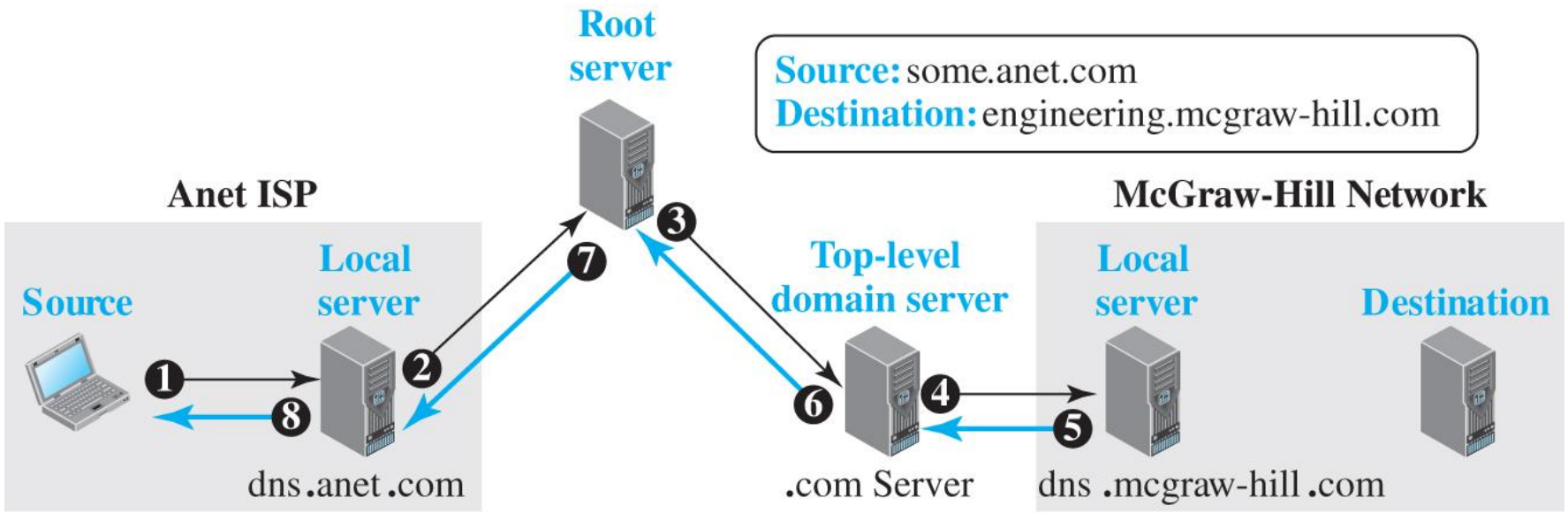


Access the text alternative for slide images.

Resolution

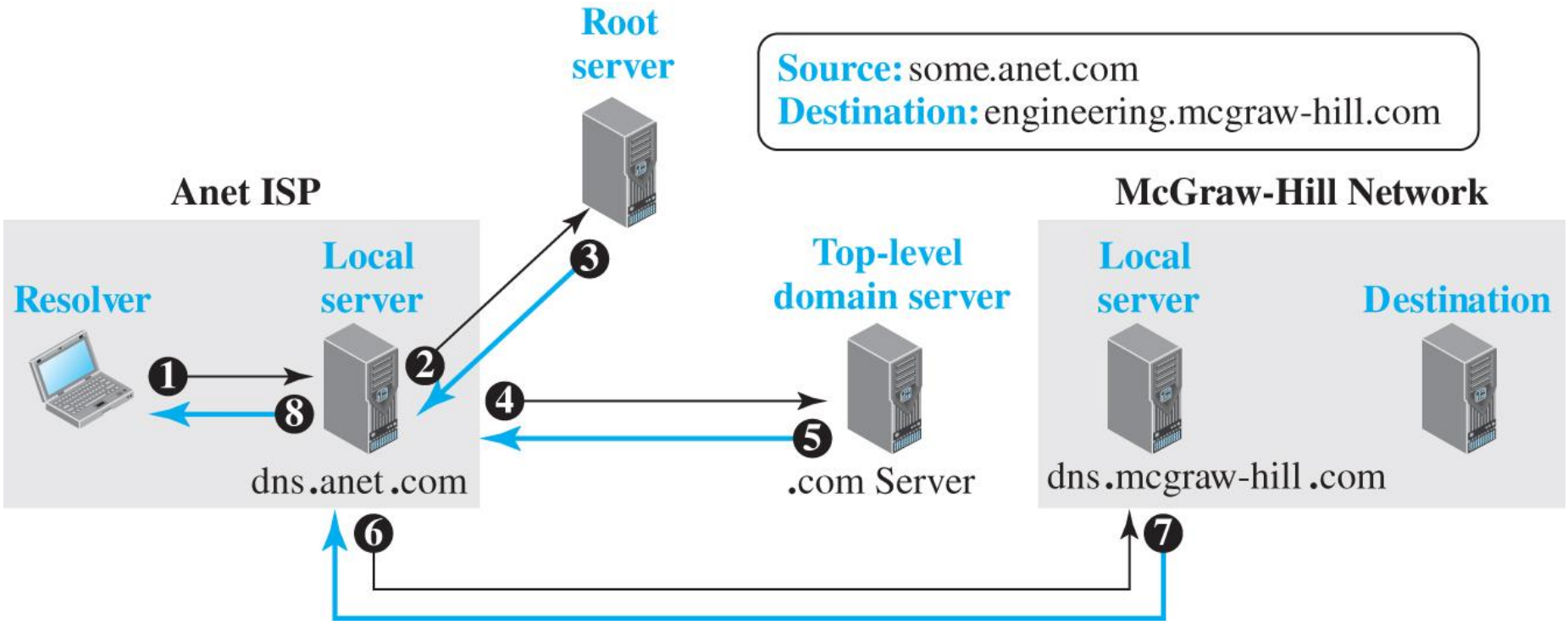
Mapping a name to an address is called name-address resolution. DNS is designed as a client-server application. A host that needs to map an address to a name or a name to an address calls a DNS client called a resolver. The resolver accesses the closest DNS server with a mapping request. If the server has the information, it satisfies the resolver; otherwise, it either refers the resolver to other servers or asks other servers to provide the information.

Figure 10.43 Recursive resolution



Access the text alternative for slide images.

Figure 10.44 Iterative resolution



Access the text alternative for slide images.

Resource Records

The zone information associated with a server is implemented as a set of resource records. In other words, a name server stores a database of resource records. A resource record is a 5-tuple structure, as shown below:

(Domain Name, Type, Class, TTL, Value)

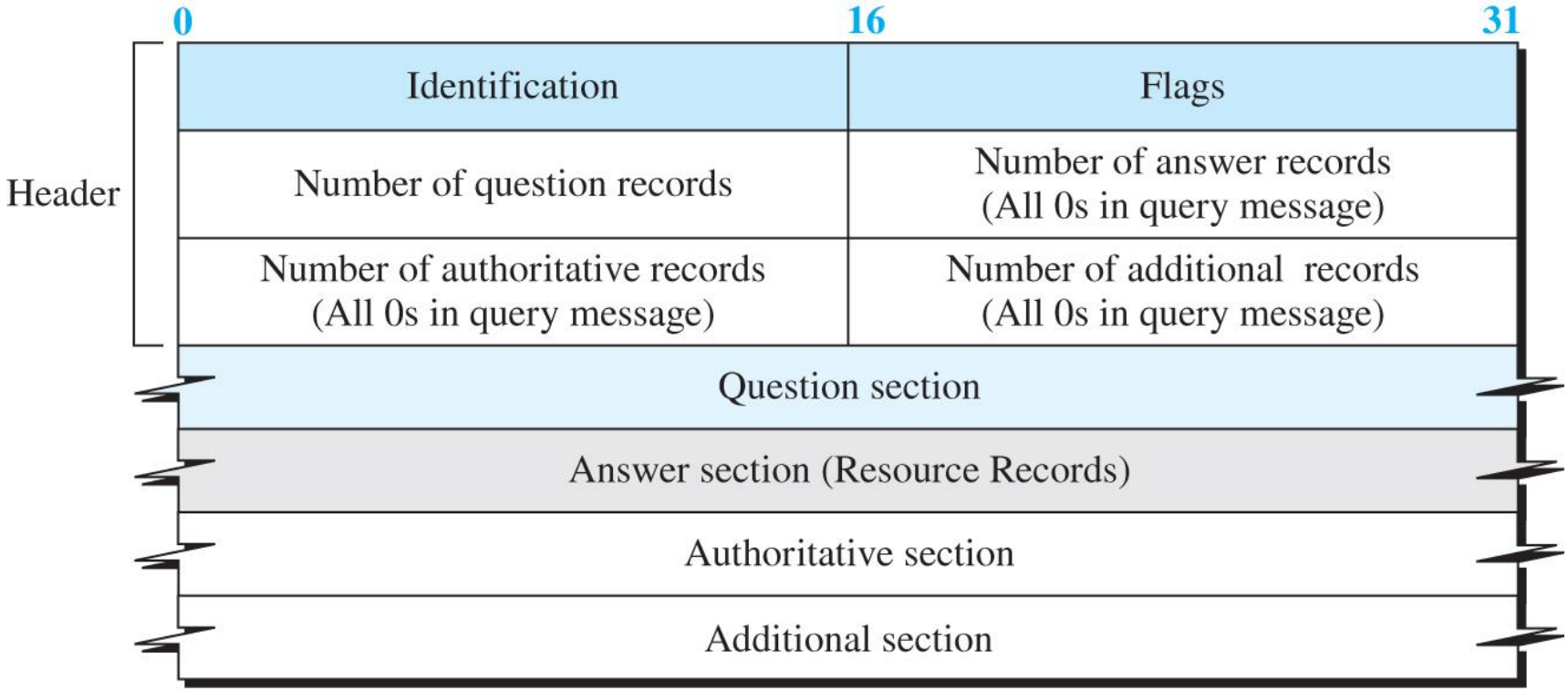
Table 10.13 DNS types

Type	*Interpretation of value*
A	A 32-bit IPv4 address (see Chapter 7)
NS	Identifies the authoritative servers for a zone
CNAME	Defines an alias for the official name of a host
SOA	Marks the beginning of a zone
MX	Redirects mail to a mail server
AAAA	An IPv6 address (see Chapter 7)

DNS Messages

To retrieve information about hosts, DNS uses two types of messages: query and response. Both types have the same format as shown in Figure 10.45.

Figure 10.45 DNS message



Note:
The query message contains only the question section.
The response message includes the question section,
the answer section, and possibly two other sections.

Access the text alternative for slide images.

Example 10.14

In UNIX and Windows, the nslookup utility can be used to retrieve address/name mapping. The following shows how we can retrieve an address when the domain name is given.

```
$nslookup www.forouzan.biz
Name: www.forouzan.biz
Address: 198.170.240.179
```

Encapsulation

DNS can use either UDP or TCP. In both cases the well-known port used by the server is port 53. UDP is used when the size of the response message is less than 512 bytes because most UDP packages have a 512-byte packet size limit. If the size of the response message is more than 512 bytes, a TCP connection is used. In that case, one of two scenarios can occur.

Registrars

How are new domains added to DNS? This is done through a registrar, a commercial entity accredited by ICANN. A registrar first verifies that the requested domain name is unique and then enters it into the DNS database.

DDNS

When the DNS was designed, no one predicted that there would be so many address changes. In DNS, when there is a change, such as adding a new host, removing a host, or changing an IP address, the change must be made to the DNS master file. These types of changes involve a lot of manual updating. The size of today's Internet does not allow for this kind of manual operation.

Security of DNS

DNS is one of the most important systems in the Internet infrastructure; it provides crucial services to Internet users. Applications such as Web access or e-mail are heavily dependent on the proper operation of DNS.

10-4 PEER-TO-PERR PARADIGM

In this section, we discuss the peer-to peer paradigm. Peer-to-peer gained popularity with Napster, an online music file. Napster paved the way for peer-to-peer file-distribution models that came later. Gnutella was followed by Fast-Track, BitTorrent, WinMX, and GNUnet.

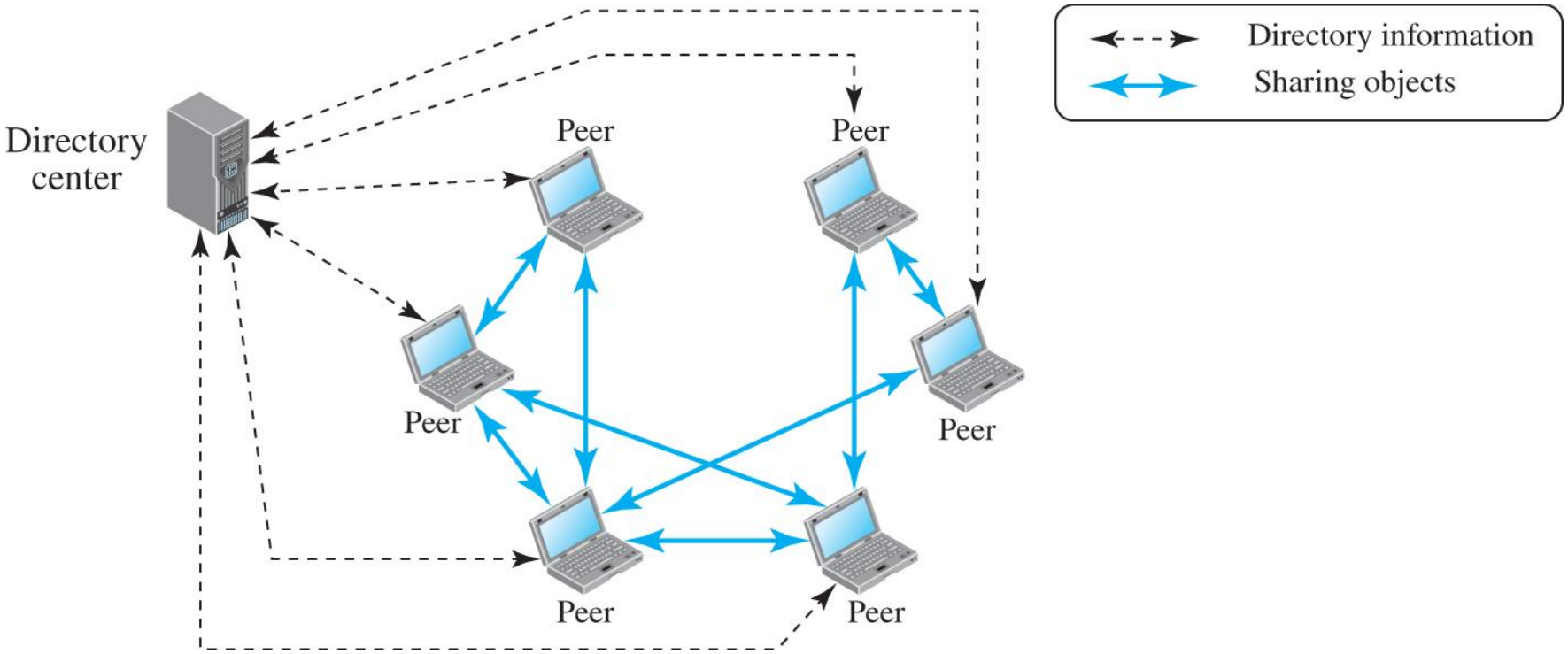
10.4.1 P2P Networks

Internet users that are ready to share their resources become peers and form a network. When a peer in the network has a file) to share, it makes it available to the rest of the peers. An interested peer can connect itself to the computer where the file is stored and download it. After a peer downloads a file, it can make it available for other peers to download. As more peers join and download that file, more copies of the file become available to the group.

Centralized Networks

In a centralized P2P network, the directory uses the client-server paradigm, but the storing and downloading of the files are done using the peer-to-peer paradigm. For this reason, a centralized P2P network is sometimes referred to as a hybrid P2P network. Napster was an example of a centralized P2P. In this type of network, a peer first registers itself with a central server. The peer then provides its IP address and a list of files it has to share. To avoid system collapse, Napster used several servers for this purpose, but we show only one in Figure 10.46.

Figure 10.46 Centralized network



Access the text alternative for slide images.

Decentralized Network

A decentralized P2P network does not depend on a centralized directory system. In this model, peers arrange themselves into an overlay network, which is a logical network made on top of the physical network. Depending on how the nodes in the overlay network are linked, a decentralized P2P network is classified as either unstructured or structured.

10.4.2 Distributed Hash Function

A Distributed Hash Table (DHT) distributes data among a set of nodes according to some predefined rules. Each peer in a DHT-based network becomes responsible for a range of data items. To avoid the flooding overhead that we discussed for unstructured P2P networks, DHT-based networks allow each peer to have a partial knowledge about the whole network. This knowledge can be used to route the queries about the data items to the responsible nodes using effective and scalable procedures.

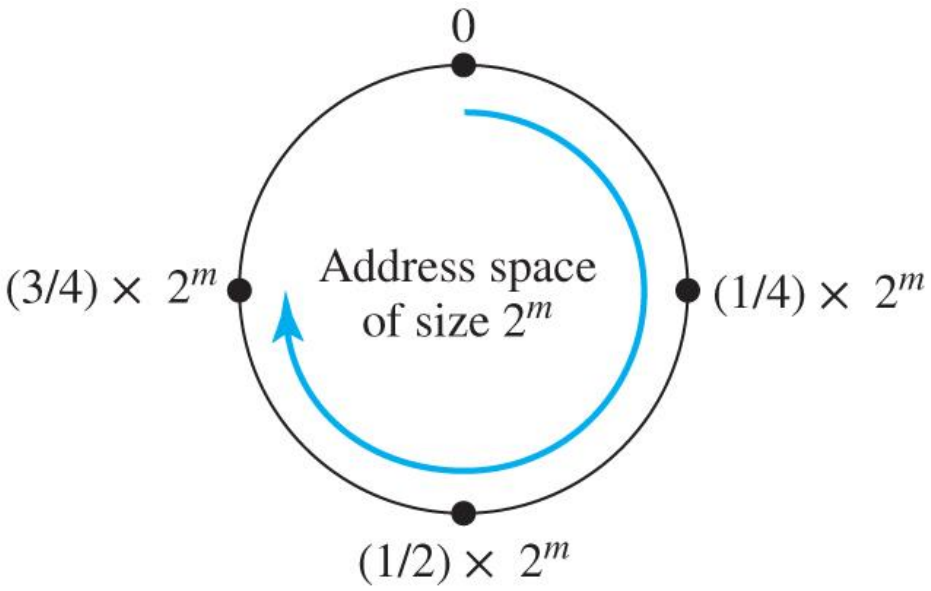
Address Space

In a DHT-based network, each data item and the peer is mapped to a point in a large address of size 2m. The address space is designed using modular arithmetic, which means that we can think of points in the address space as distributed evenly on a circle with 2m points (0 to 2m - 1) using clockwise direction as shown in Figure 10.47. Most of the DHT implementations use m = 160.

Figure 10.47 Address space

Note:

1. Space range is 0 to $2^m - 1$.
2. Calculation is done modulo 2^m.

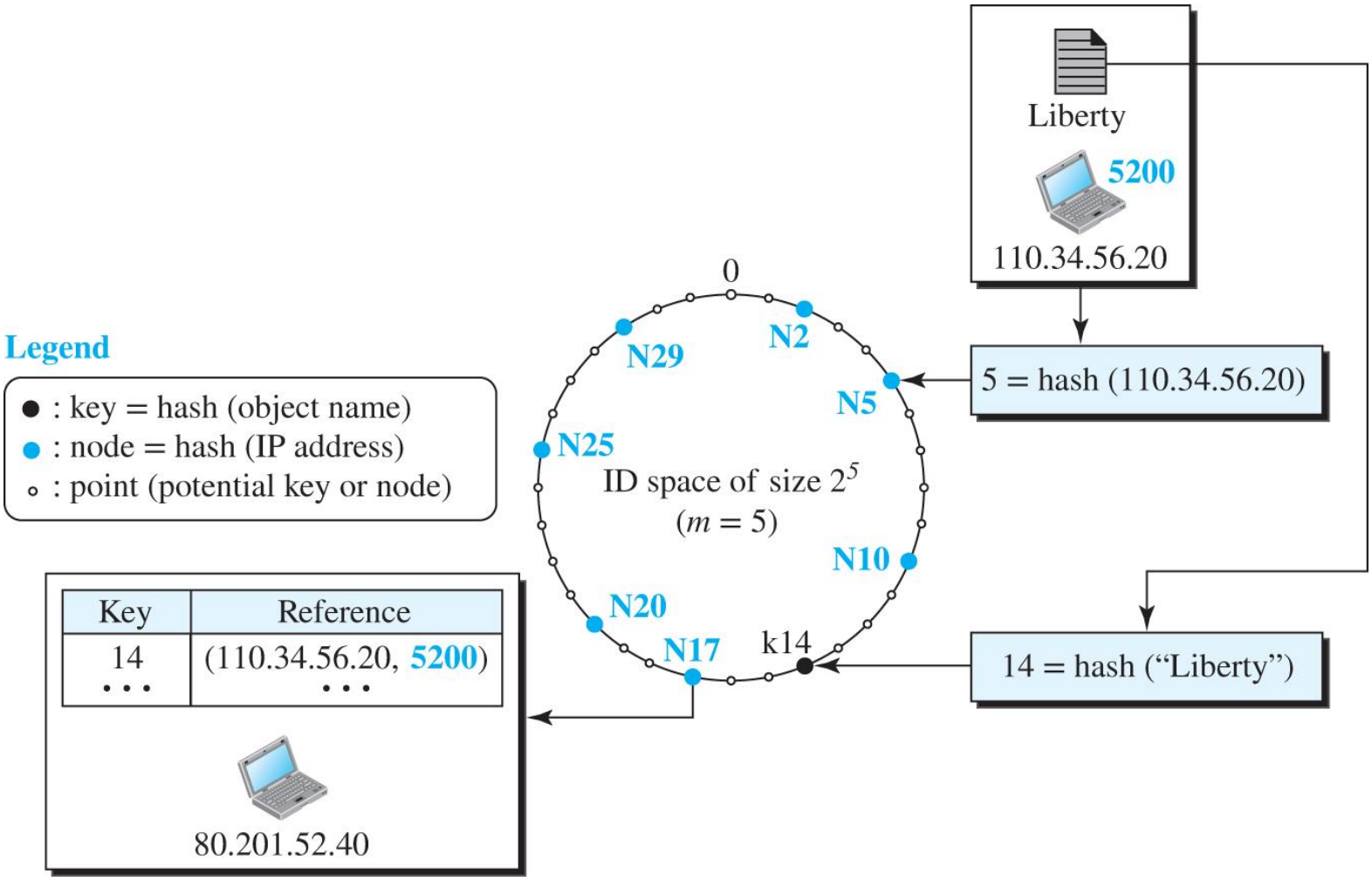


Access the text alternative for slide images.

Example 10.15

Although the normal value of m is 160, for the purpose of demonstration, we use $m = 5$ to make our examples tractable. In Figure 2.48, we assume that several peers have already joined the group. The node N5 with IP address 110.34.56.20 has a file named Liberty that wants to share with its peers. The node makes a hash of the file name, "Liberty," to get the key = 14. Since the closest node to key 14 is node N17, N5 creates a reference to file name (key), its IP address, and the port number (and possibly some other information about the file) and sends this reference to be stored in node N17. In other words, the file is stored in N5, the key of the file is k14 (a point in the DHT ring), but the reference to the file is stored in node N17.

Figure 10.48 Example 10.15



Access the text alternative for slide images.

10.4.3 Chord

There are several protocols that implement DHT systems. In this section, we introduce the Chord protocol for its simplicity and elegant approach to routing queries. Chord was published by Stoica et al in 2001. We briefly discuss the main feature of this algorithm here.

Identifier Space 1

Data items and nodes in Chord are m-bit identifiers that create an identifier space of size 2m points distributed in a circle in the clockwise direction. We refer to the identifier of a data item as k (for key) and the identifier of a peer as N (for node). Arithmetic in the space is done modulo 2m, which means that the identifiers are wrapped from 2m - 1 back to 0. Although some implementations use a collision-resistant hash function like SHA1 with m = 160, we use m = 5 in our discussion to make the discussion simpler.

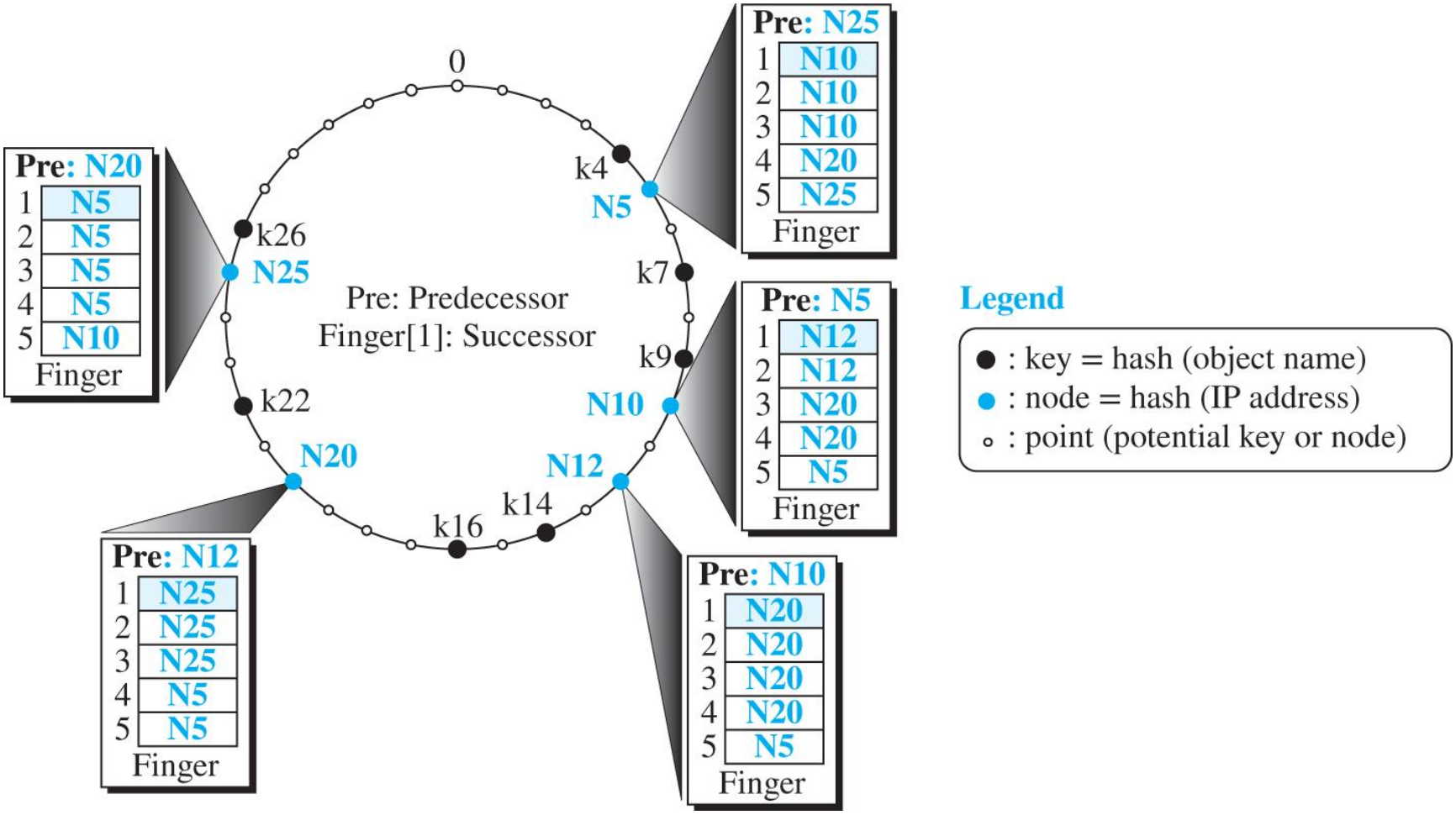
Finger Table

A node in the Chord algorithm should be able to resolve a query: given a key, the node should be able to find the node identifier responsible for that key or forward the query to another node. Forwarding, however, means that each node needs to have a routing table. Chord requires that each node knows about m successor nodes and one predecessor node. Each node creates a routing table, called a finger table by Chord, that looks like Table 10.14. Note that the target key at row i is N + 2 ^ (i-1).

Table 10.14 Finger table

i	*Target Key*	*Successor of Target Key*	*Information about Successor*
1	$N + 1$	Successor of $N + 1$	IP address and port of successor
2	$N + 2$	Successor of $N + 2$	IP address and port of successor
⋮	⋮	⋮	⋮
m	$N + 2^{m-1}$	Successor of $N + 2^{m-1}$	IP address and port of successor

Figure 10.49 An example of a ring in Chord



Access the text alternative for slide images.

Table 10.15 Lookup 1

```
Lookup (key)
{
    if (node is responsible for the key)
        return (node's ID)
    else
        return find_succesor (key)
}
find_successor (id)
{
    x = find_ predecessor (id)
    return x.finger[1]
}
```

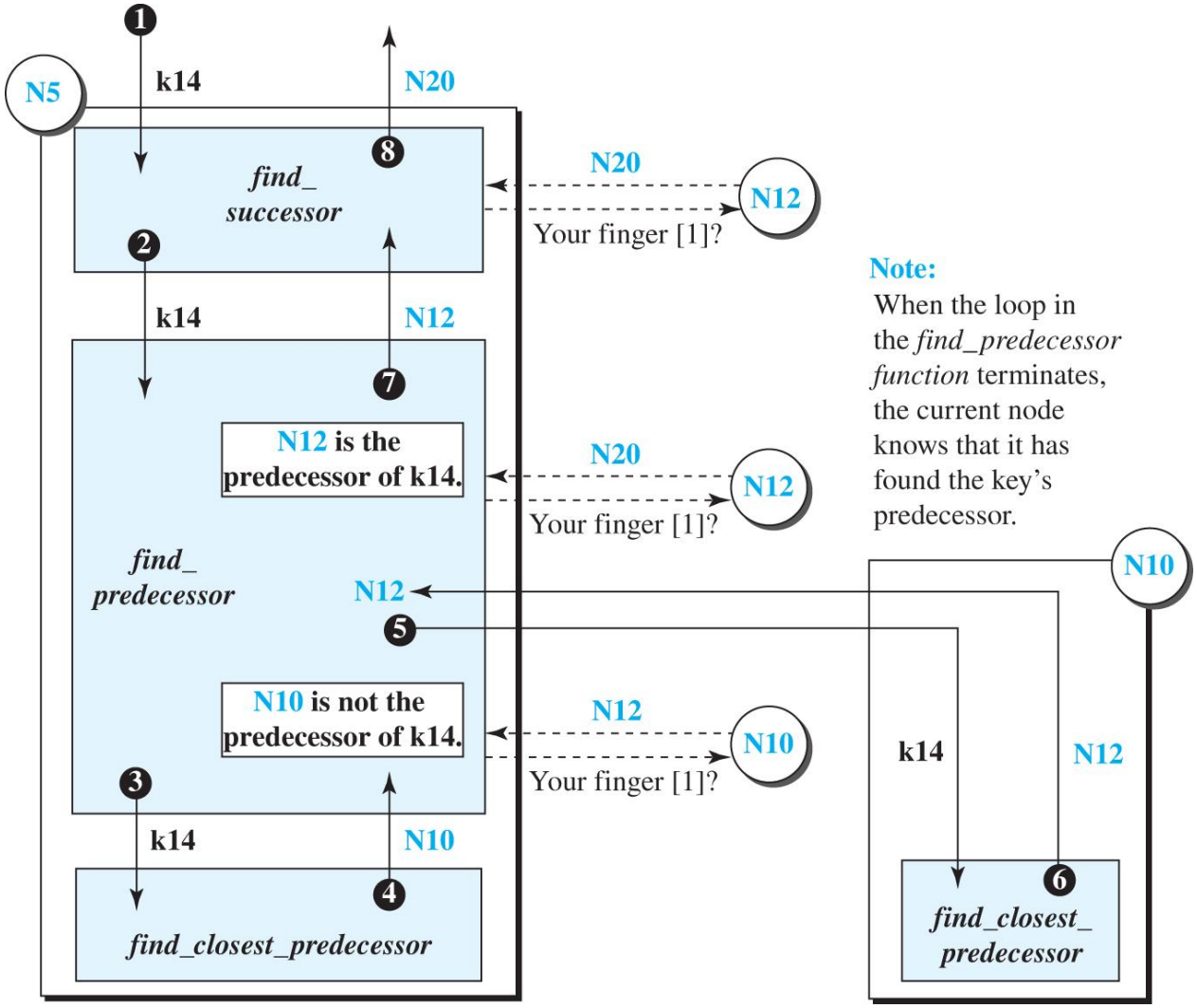
Table 10.15 Lookup 2

```
find_predecessor (id)
{
        x = N                                          // N is the current node
        while (id ∉ (x, x.finger[1]))
        {
                x = x.find_closest_predecessor (id)            // Let x find it
        }
        return x
}
find_closest_predecessor (id)
{
        for (i = m downto 1)
        {
                if (finger [i] ∈ (N, id))                      //N is the current node
                        return (finger [i])
        }
        return N                                       //The node itself is closest predecessor
}
```

Example 10.16

Assume node N5 in Figure 10.49 needs to find the responsible node for key k14. Figure 10.50 shows the sequence of 8 events to do so.

Figure 10.50 Example 10.16



Access the text alternative for slide images.

Table 10.16 Stabilize

```
Stabilize ()
{
    P = finger[1].Pre          //Ask the successor to return its predecessor
    if (P ∈ (N, finger[1]))  finger[1] = P      // P is the possible successor of N
    finger[1].notify (N)                  // Notify P to change its predecessor
}
Notify (x)
{
    if (Pre = null or x ∈ (Pre, N))    Pre = x
}
```

Table 10.17 Fix_Finger

```
Fix_Finger ()
{
    Generate (i ∈ (1, m])              //Randomly generate i such as 1 < i ⩽ m
    finger[i] = find_successor (N + 2^(i−1))      // Find value of finger[i]
}
```

Table 10.18 Join

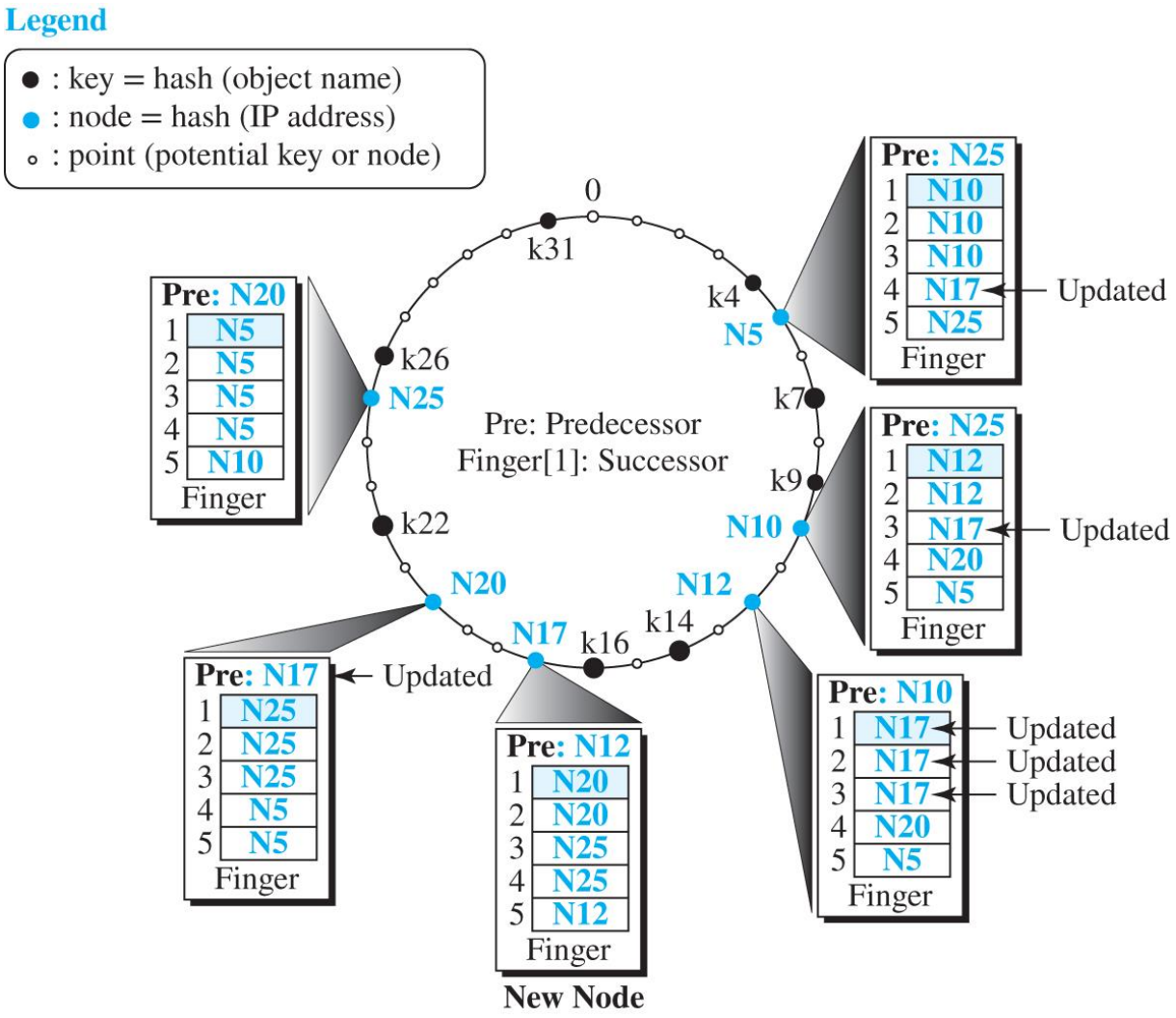
```
Join (x)
{
        Initialize (x)
        finger[1].Move_Keys (N)
}
Initialize (x)
{
        Pre = null
        if (x = null) finger[1] = N
        else finger[1] = x. Find_Successor (N)
}
Move_Keys (x)
{
        for (each key k)
        {
            if (x ∈ [k, N)) move (k to node x)          // N is the current node
        }
}
```

Example 10.17

We assume that node N17 joins the ring in Figure 10.49 with the help of N5. Figure 10.51 shows the ring after the ring has been stabilized. The following five steps shows the process:

1. N17 set its predecessor to null and its successor to N20.
2. N17 then asks N20 to send k14 and k16 to N17.
3. N17 validates its own successor and asks N20 to change its predecessor to N17
4. The predecessor of N17 is updated to N12.
5. The finger table of nodes N17, N10, N5, and N12 is changed.

Figure 10.51 Example 10.17



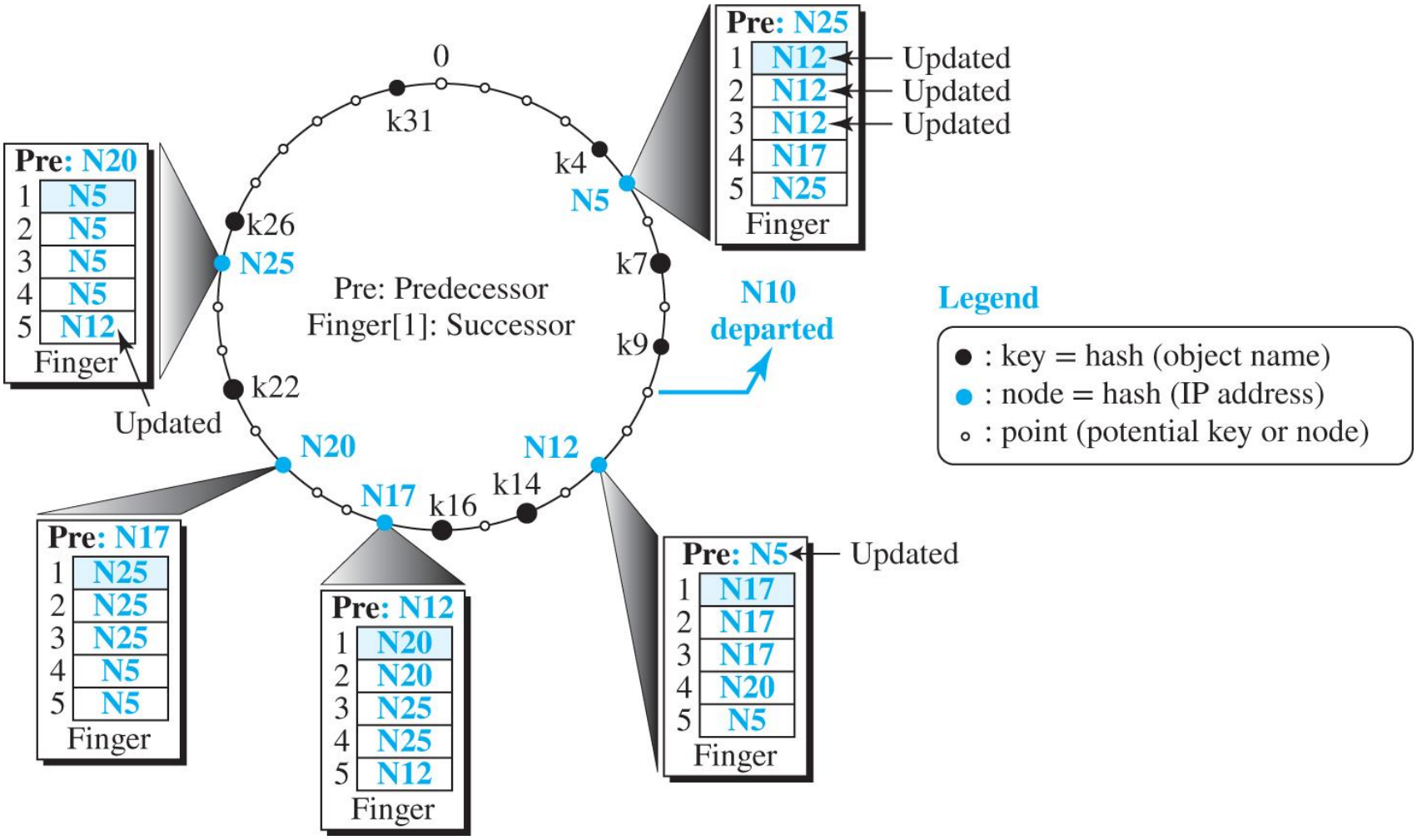
Access the text alternative for slide images.

Example 10.18

We assume that a node, N10, leaves the ring in Figure 2.51. Figure 2.52 shows the ring after it has been stabilized. The following shows the process:

1. Node N5 finds out about N10's departure when it does not receive a pong message to its ping message. Node N5 changes its successor to N12 in the list of successors.
2. Node N5 immediately launches the *stabilize* function and asks N12 to change its predecessor to N5.
3. Hopefully, k7 and k9, which were under the responsibility of N10, have been duplicated in N12 before the departure of N10.
4. Nodes N5 and N25 update their finger tables.

Figure 10.52 Example 10.18



Access the text alternative for slide images.

Applications 2

Chord is used in several applications including Collaborative File System (CFS), ConChord, and Distributive Domain Name System (DDNS).

10.4.4 Pastry

Another popular protocol in the P2P paradigm is Pastry, designed by Rowstron and Druschel. Pastry uses DHT, as described before, but there are some fundamental differences between Pastry and Chord in the identifier space and routing process that we describe next.

Identifier Space 2

In Pastry, like Chord, nodes and data items are m-bit identifiers that create an identifier space of 2^m points distributed uniformly on a circle in the clockwise direction. The common value for m is 128. The protocol uses the SHA-1 hashing algorithm with m = 128. However, in this protocol, an identifier is seen as an n-digit string in base 2b in which b is normally 4 and n = (m / b). In other words, an identifier is a 32-digit number in base 16 (hexadecimal).

Routing

A node in Pastry should be able to resolve a query; given a key, the node should be able to find the node identifier responsible for that key or forward the query to another node. Each node in Pastry uses two entities to do so: a routing table and a leaf set.

Table 10.19 Routing table for a node in Pastry

Common prefix length	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*A*	*B*	*C*	*D*	*E*	*F*
0																
1																
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
31																

Example 10.19

Let us assume that $m = 8$ bits and $b = 2$. This means that we have up to $2^m = 256$ identifiers, and each identifier has m/b = 4 digits in base $2^b = 4$. Figure 10.53 shows the situation in which there are some live nodes and some keys mapped to these nodes. The key k1213 is stored in two nodes because it is equidistant from them.

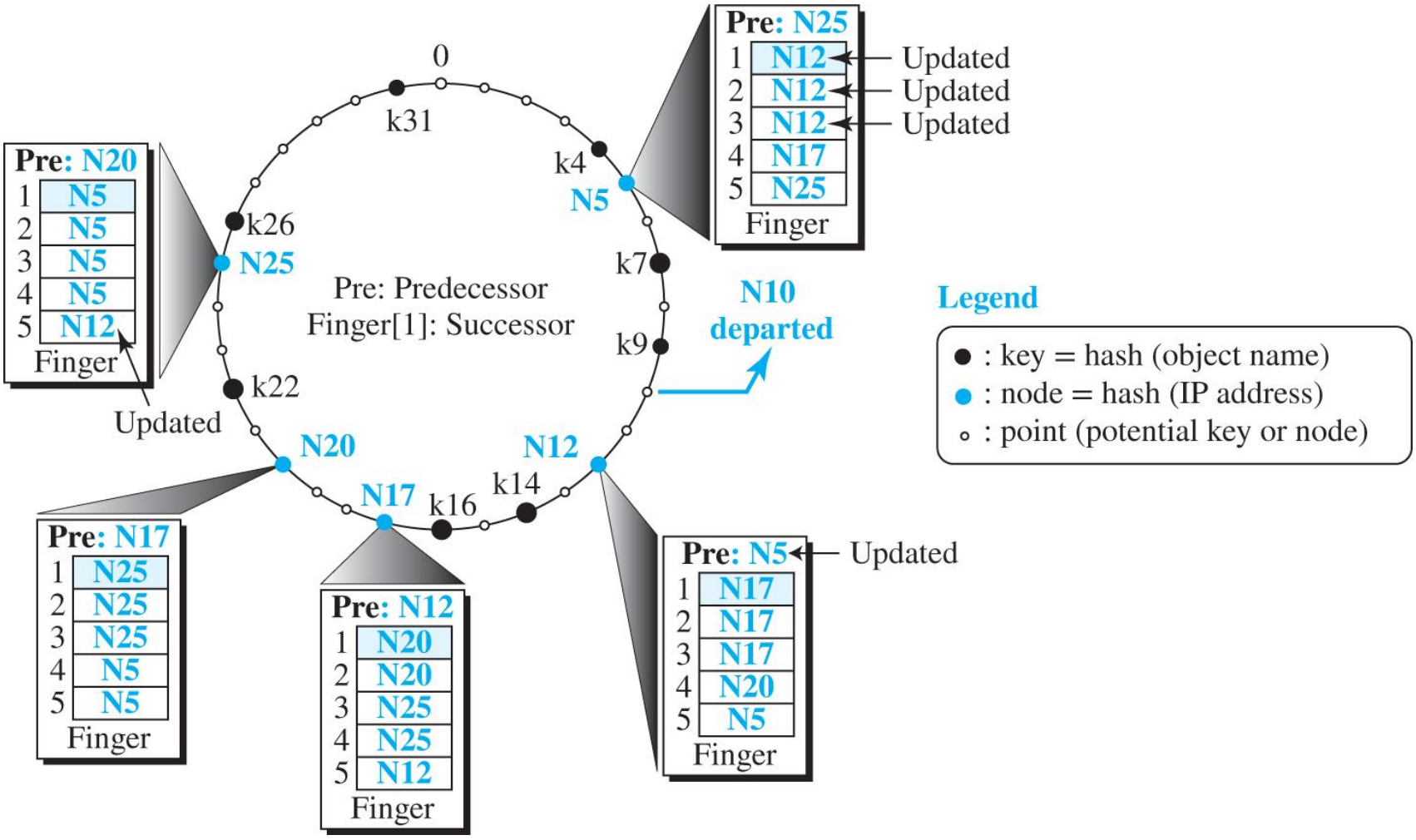
Lookup

As we discussed in Chord, one of the operations used in Pastry is lookup: given a key, we need to find the node that stores the information about the key or the key itself. Table 10.20 gives the lookup operation in pseudocode. In this algorithm, N is the identifier of the local node, the node that receives a message and needs to find the node that stores the key in the message.

Example 10.21

In Figure 10.53, we assume that node N2210 receives a query to find the node responsible for key 2008. Since this node is not responsible for this key, it first checks its leaf set. The key 2008 is not in the range of the leaf set, so the node needs to use its routing table. Since the length of the common prefix is 1, p = 1. The value of the digit at position 1 in the key is v = 0. The node checks the identifier in Table [1, 0], which is 2013. The query is forwarded to node 2013, which is actually responsible for the key. This node sends its information to the requesting node.

Figure 10.53 An example of a Pastry ring



Access the text alternative for slide images.

Table 10.20 Lookup (Pastry)

```
Lookup (key)
{
    if (key is in the range of N's leaf set)
        forward the message to the closest node in the leaf set
    else
        route (key, Table)
}
route (key, Table)
{
    p = length of shared prefix between key and N
    v = value of the digit at position p of the key          // Position starts from 0
    if (Table [p, v] exists)
        forward the message to the node in Table [p, v]
    else
    forward the message to a node sharing a prefix as long as the current node, but
    numerically closer to the key.
}
```

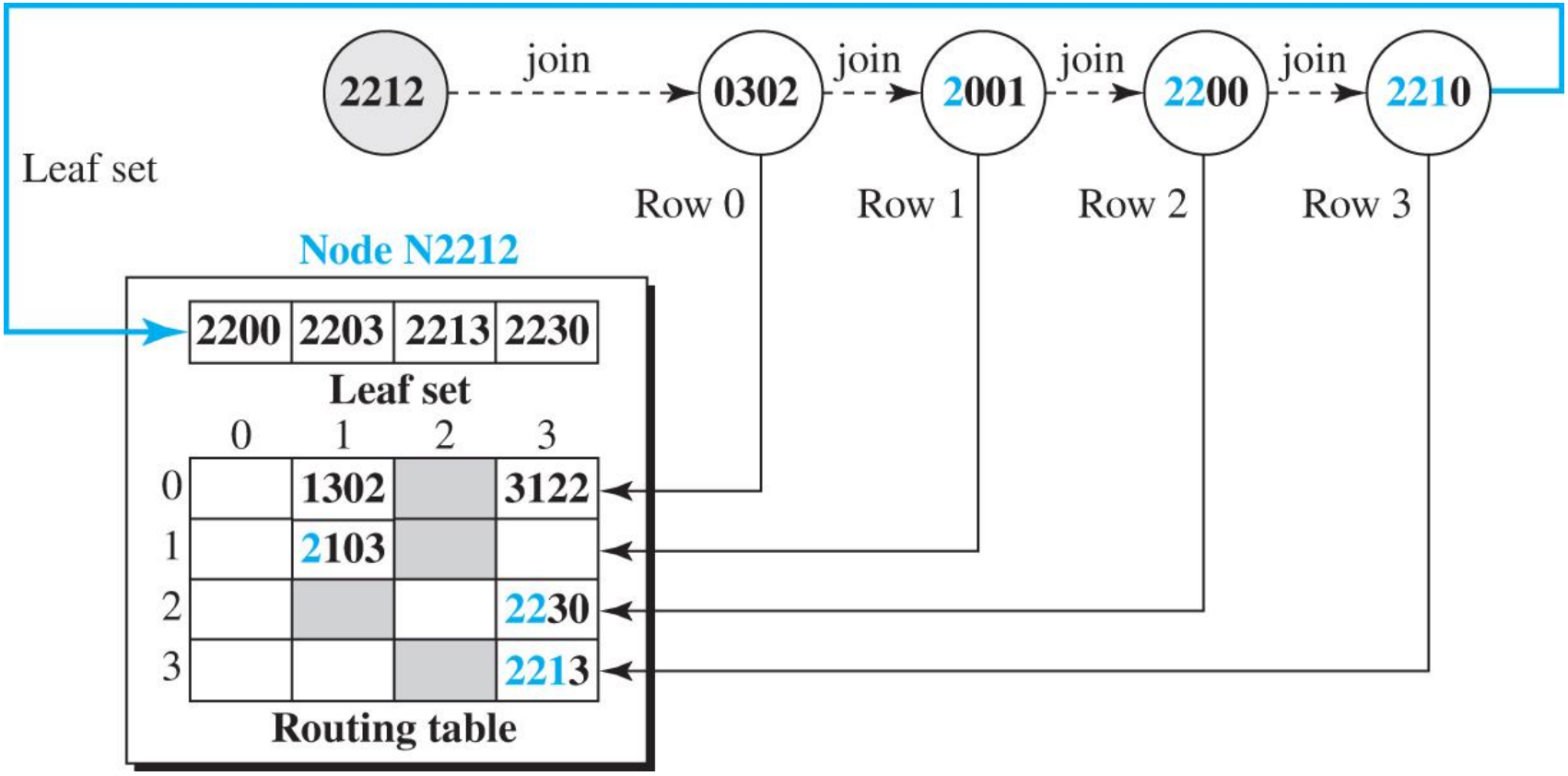
Join 1

The process of joining the ring in Pastry is simpler and faster than in Chord. The new node, X, should know at least one node N0, which should be close to X (based on the proximity metric); this can be done by running an algorithm called Nearby Node Discovery. Node X sends a join message to N0. In our discussion, we assume that N0's identifier has no common prefix with X's identifier.

Example 10.22

Figure 10.54 shows how a new node X with node identifier N2212 uses the information in four nodes in Figure 2.53 to create its initial routing table and leaf set for joining the ring. Note that the contents of these two tables will become closer to what they should be in the updating process. In this example, we assume that node 0302 is a nearby node to node 2212 based on the proximity metric.

Figure 10.54 Example 10.22



Access the text alternative for slide images.

Leave or Fail 1

Each Pastry node periodically tests the liveliness of the nodes in its leaf set and routing table by exchanging probe messages. If a local node finds that a node in its leaf set is not responding to the probe message, it assumes that the node has failed or departed. To replace it in its leaf set, the local node contacts the live node in its leaf set with the largest identifier and repairs its leaf set with the information in the leaf set of that node. Since there is an overlap in the leaf set of close-by nodes, this process is successful.

Application 3

Pastry is used in some applications including PAST, a distributed file system, and SCRIBE, a decentralized publish/subscribe system.

10.4.5 Kademlia

Another DHT peer-to-peer network is Kademlia, designed by Maymounkov and Mazières. Kademlia, like Pastry, routes messages based on the distance between nodes, but the interpretation of the distance metric in Kademlia is different from the one in Pastry, as we describe below. In this network, the distance between the two identifiers (nodes or keys) is measured as the bitwise exclusive-or (XOR), between them.

Identifier Space 3

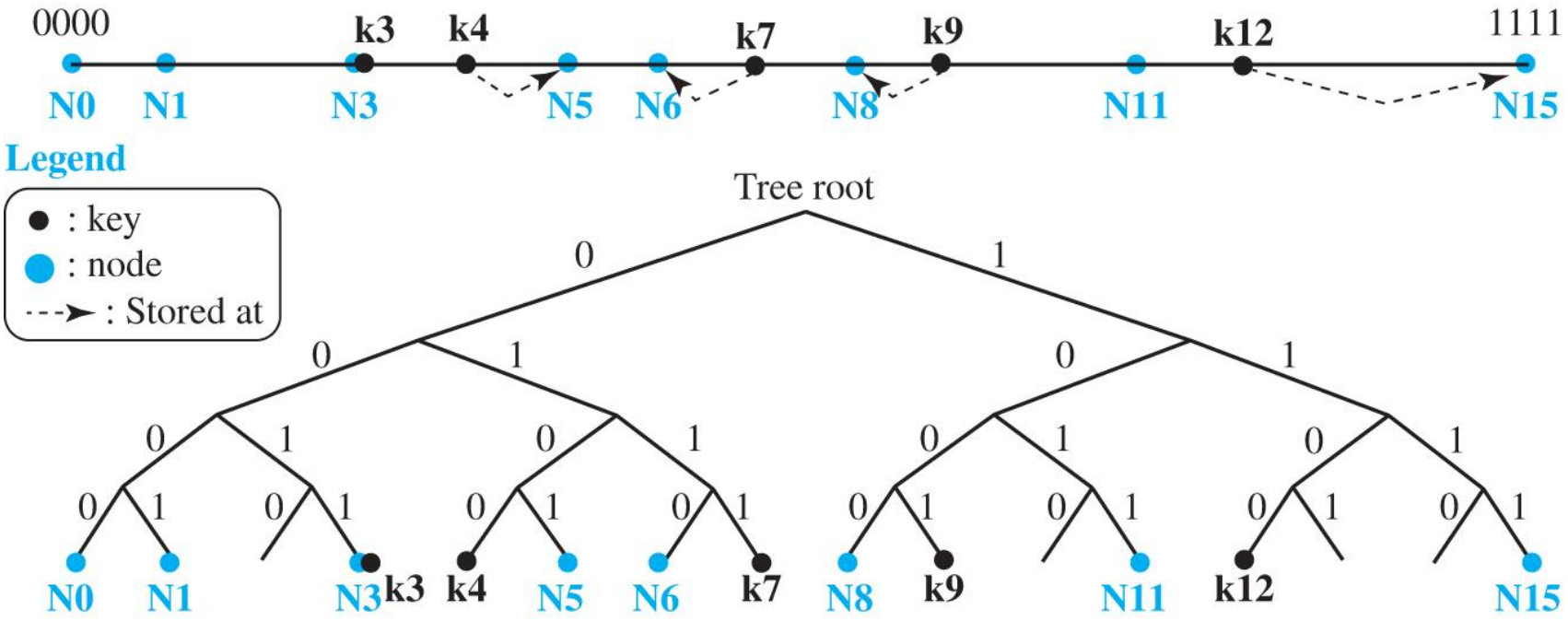
In Kademlia, nodes and data items are m-bit identifiers that create an identifier space of 2m points distributed on the leaves of a binary tree. The protocol uses the SHA-1 hashing algorithm with m = 160.

Example 10.23

- For simplicity, let us assume that $m = 4$. In this space, we can have 16 identifiers distributed on the leaves of a binary tree. Figure 10.55 shows the case with only eight live nodes and five keys.

- As the figure shows, the key k3 is stored in N3 because $3 \oplus 3 = 0$. Although the key k7 looks numerically equidistant from N6 and N8, it is stored only in N6 because $6 \oplus 7 = 1$ but $6 \oplus 8 = 14$. Another interesting point is that the key k12 is numerically closer to N11, but it is stored in N15 because $11 \oplus 12 = 7$, but $15 \oplus 12 = 3$.

Figure 10.55 Example 10.23



Access the text alternative for slide images.

Routing Table

Kademlia keeps only one routing table for each node; there is no leaf set. Each node in the network divides the binary tree into m subtrees that do not include the node itself. Subtree i includes nodes that share i leftmost bit (common prefix) with the corresponding node. The routing table is made of m rows but only one column. In our discussion, we assume that each row holds the identifier of one of the nodes in the corresponding subtree, but later we show that Kademlia allows up to k nodes in each row. The idea is the same as that used by Pastry, but the length of the common prefix is based on the number of bits instead of the number of digits in base 2b.

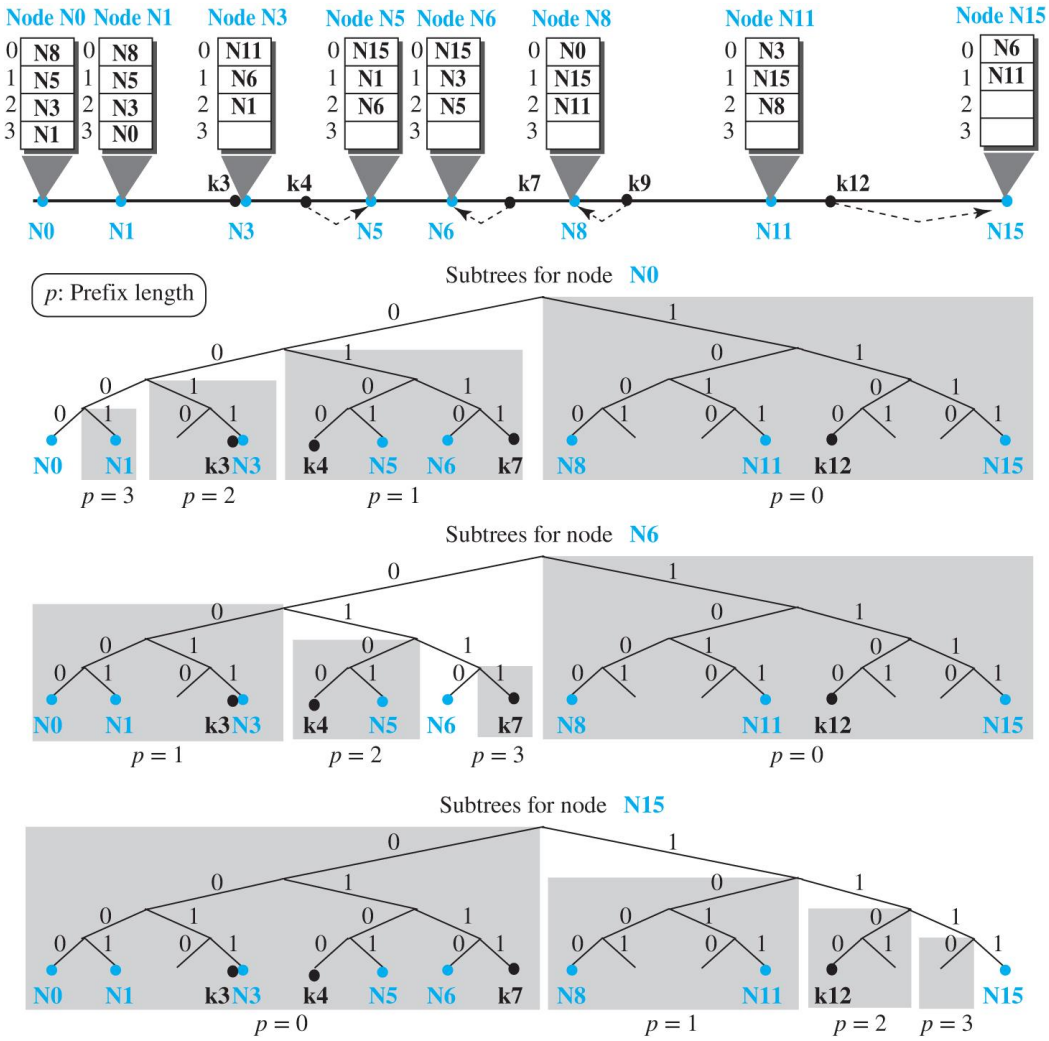
Table 10.21 Routing table for a node in Kademlia

Common prefix length	*Identifiers*
0	Closest node(s) in subtree with common prefix of length 0
1	Closest node(s) in subtree with common prefix of length 1
⋮	⋮
$m-1$	Closest node(s) in subtree with common prefix of length $m-1$

Example 10.24

Let us find the routing table for Example 2.23. To make the example simple, we assume that each row uses only one identifier. Since m = 4, each node has four subtrees corresponding to four rows in the routing table. The identifier in each row represents the node that is closest to the current node in the corresponding subtree. Figure 2.56 shows all routing tables, but only three of the subtrees. We have chosen these three, out of eight, to make the figure smaller.

Figure 10.56 Example 10.24



Access the text alternative for slide images.

Example 10.25

In Figure 10.56, we assume node N0 $(0000)_2$ receives a lookup message to find the node responsible for k12 $(1100)_2$. The length of the common prefix between the two identifiers is 0. Node N0 sends the message to the node in row 0 of its routing table, node N8. Now node N8 $(1000)_2$ needs to look for the node closest to k12 $(1100)_2$. The length of the common prefix between the two identifiers is 1. Node N8 sends the message to the node in row 1 of its routing table, node N15, which is responsible for k12. The routing process is terminated. The route is N0 → N8 → N15. It is interesting to note that node N15, $(1111)_2$, and k12, $(1100)_2$, have a common prefix of length 2, but row 2 of N15 is empty, which means that N15 itself is responsible for k12.

Example 10.26

In Figure 10.56, we assume node N5 $(0101)_2$ receives a lookup message to find the node responsible for k7 $(0111)_2$. The length of the common prefix between the two identifiers is 2. Node N5 sends the message to the node in row 2 of its routing table, node N6, which is responsible for k7. The routing process is terminated. The route is N5 → N6.

Example 10.27

In Figure 10.56, we assume node N11 $(1011)_2$ receives a lookup message to find the node responsible for k4 $(0100)_2$. The length of the common prefix between the two identifiers is 0. Node N11 sends the message to the node in row 0 of its routing table, node N3. Now node N3 $(0011)_2$ needs to look for the node closest to k4 $(0100)_2$. The length of the common prefix between the two identifiers is 1. Node N3 sends the message to the node in row 1 of its routing table, node N6. And so on. The route is N11 → N3→ N6 → N5.

K-Buckets

In our previous discussion, we assumed that each row in the routing table lists only one node in the corresponding subtree. For more efficiency, Kademlia requires that each row keeps at least up to k nodes from the corresponding subtree. The value of k is system independent, but for an actual network it is recommended that it be around 20. For this reason, each row in the routing table is referred to as a k-bucket. Having more than one node in each row allows the node to use an alternative node when a node leaves the network or fails. Kademlia keeps those nodes in a bucket that has been connected in the network for a long time.

Join 2

As in Pastry, a node that needs to join the network needs to know at least one other node. The joining node sends its identifier to the node as though it is a key to be found. The response it receives allows the new node to create its k-buckets.

Leave or Fail 2

When a node leaves the network or fails, other nodes update their k-buckets using the concurrent process described before.

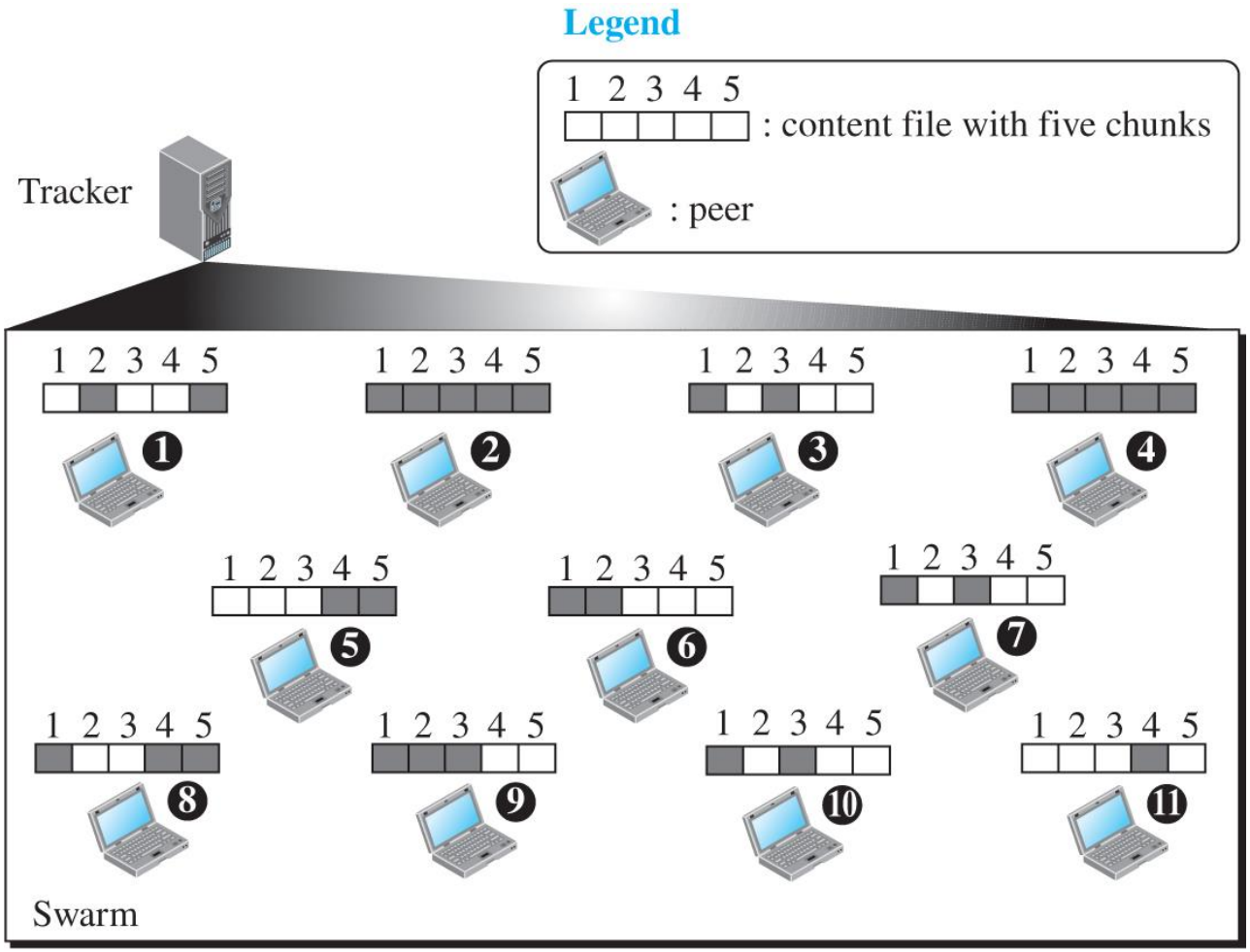
10.4.6 A Popular P2P Network: BitTorrent

BitTorrent is a P2P protocol, designed by Bram Cohen, for sharing a large file among a set of peers. However, the term sharing in this context is different from other file sharing protocols. Instead of one peer allowing another peer to download the whole file, a group of peers takes part in the process to give all peers in the group a copy of the file. File sharing is done in a collaborating process called a torrent.

BitTorrent with a Tracker

In the original BitTorrent, there is another entity in a torrent, called the tracker, which, as the name implies, tracks the operation of the swarm, as described later. Figure 10.57 shows an example of a torrent with seeds, leeches, and the tracker.

Figure 10.57 Example of a torrent



Note: Peers 2 and 4 are seeds; others are leeches.

Access the text alternative for slide images.

Trackerless BitTorrent

In the original BitTorrent design, if the tracker fails, new peers cannot connect to the network and updating is interrupted. There are several implementations of BitTorrent that eliminate the need for a centralized tracker. In the implementation that we describe here, the protocol still uses the tracker, but not a central one. The job of tracking is distributed among some nodes in the network. In this section, we show how Kademlia DHT can be used to achieve this goal, but we avoid becoming involved in the details of a specific protocol.

10-5 SOCKET-INTERFACE PROGRAMMING

In this section, we show how to write some simple client-server programs using C, a procedural programming language.

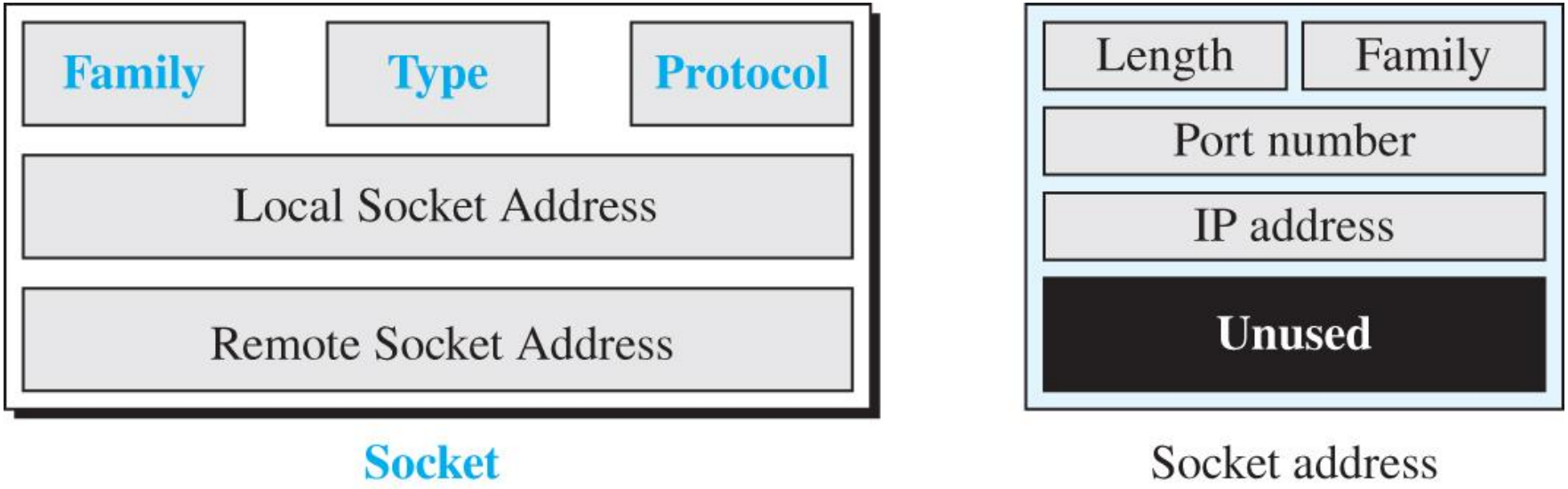
10.5.1 Socket Interface in C

In this section, we show how this interface is implemented in the C language. The important issue in socket interface is to understand the role of a socket in communication. The socket has no buffer to store data to be sent or received. It is capable of neither sending nor receiving data. The socket just acts as a reference or a label. The buffers and necessary variables are created inside the operating system.

Data Structure for Socket

The C language defines a socket as a structure (struct). The socket structure is made of five fields; each socket address itself is a structure made of five fields, as shown in Figure 10.58. Note that the programmer should not redefine this structure; it is already defined in the header files. We briefly discuss the five fields in a socket structure.

Figure 10.58 Socket data structure



Access the text alternative for slide images.

Header Files

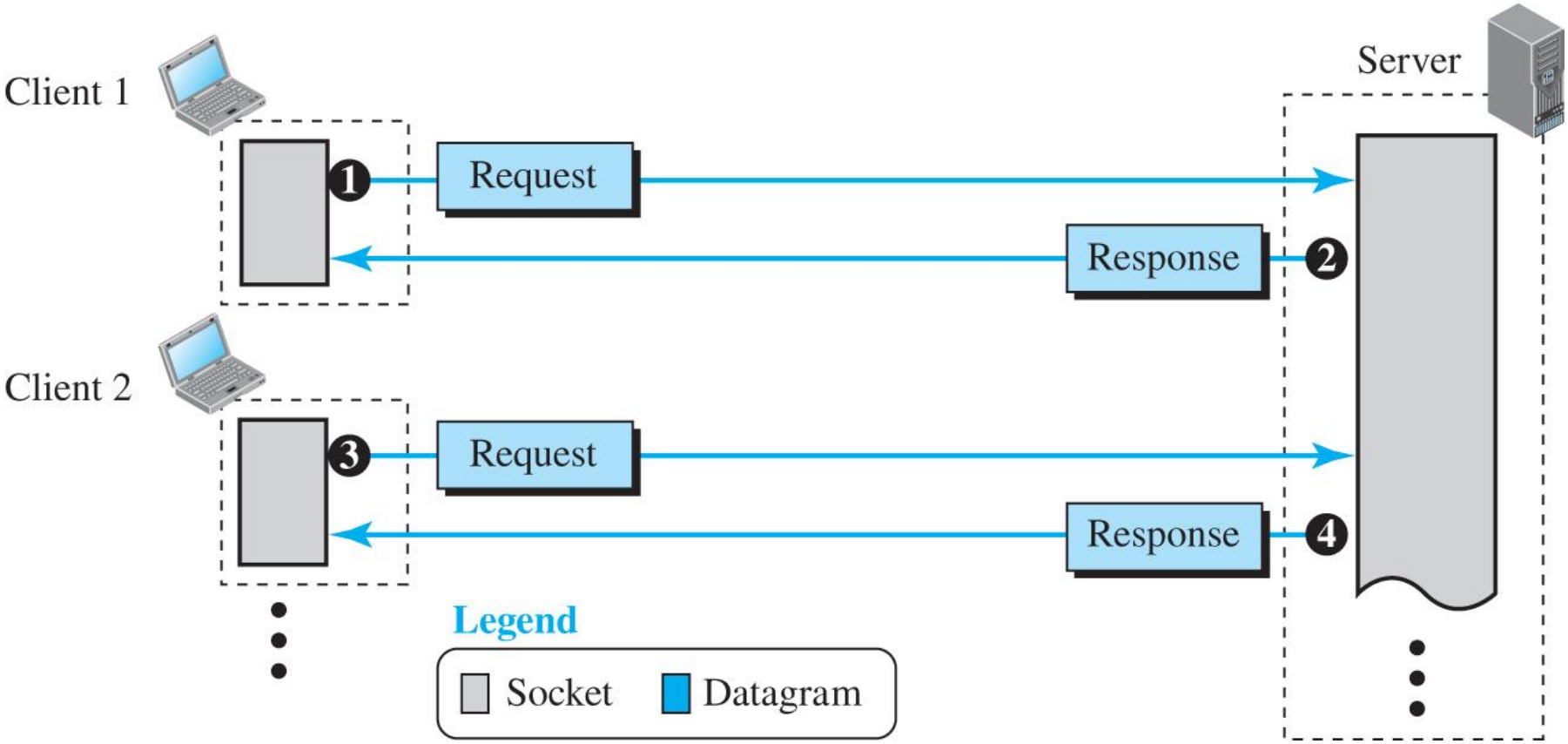
To be able to use the definition of the socket and all procedures (functions) defined in the interface, we need a set of header files.

```
// "headerFiles.h"
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <errno.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>
#include <arpa/innet.h>
#include <sys/wait.h>
```

Iterative Communication Using UDP

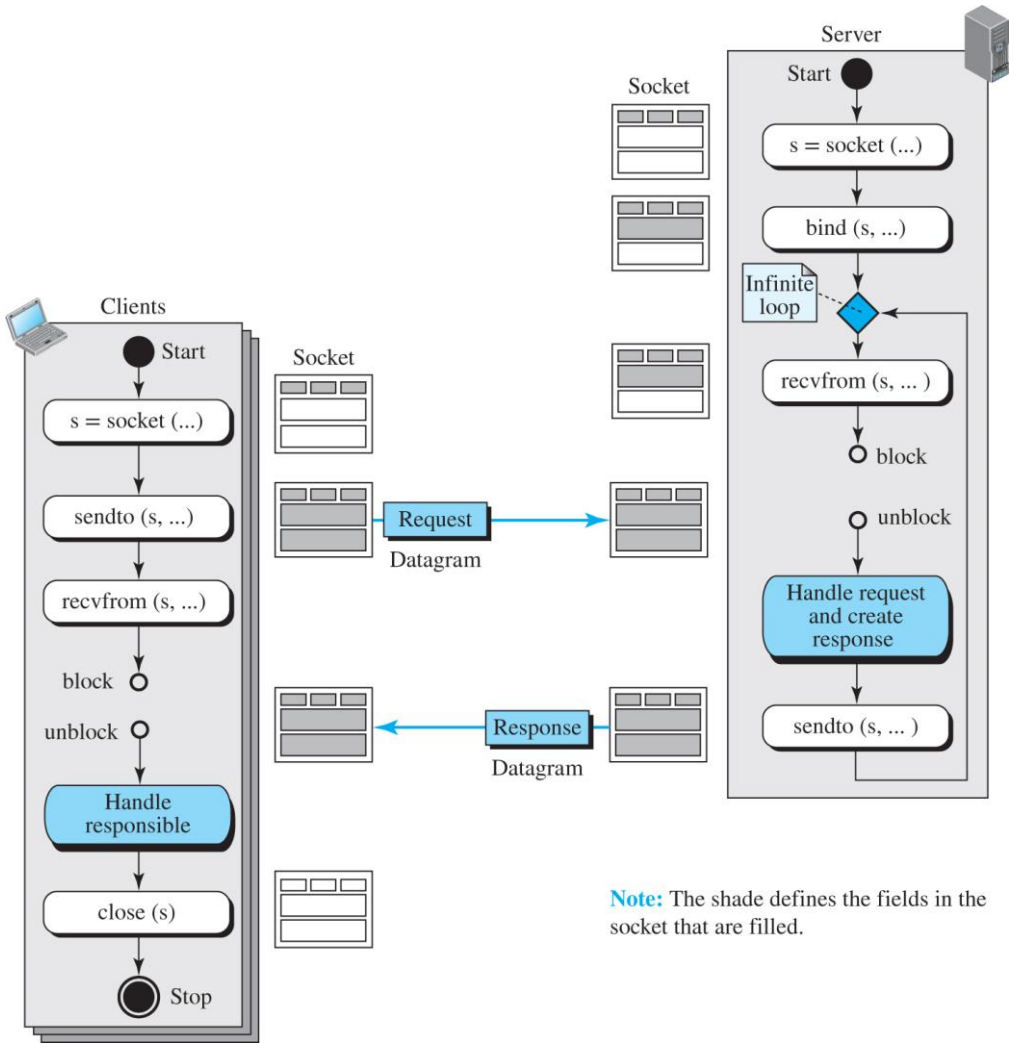
As we discussed earlier, UDP provides a connectionless server, in which a client sends a request and the server sends back a response.

Figure 10.59 Sockets for UDP communication



Access the text alternative for slide images.

Figure 10.60 Flow diagram for iterative UDP communication



Access the text alternative for slide images.

Table 10.22 Echo server program using UDP 1

```
1   // UDP echo server program
2   #include "headerFiles.h"
3   int main (void)
4   {
5     // Declare and define variables
6     int s;                                  // Socket descriptor (reference)
7     int len;                                // Length of string to be echoed
8     char buffer [256];                      // Data buffer
9     struct sockaddr_in servAddr;            // Server (local) socket address
10    struct sockaddr_in clntAddr;            // Client (remote) socket address
11    int clntAddrLen;                        // Length of client socket address
12    // Build local (server) socket address
13    memset (&servAddr, 0, sizeof (servAddr));                  // Allocate memory
14    servAddr.sin_family = AF_INET;                             // Family field
15    servAddr.sin_port = htons (SERVER_PORT)                    // Default port number
16    servAddr.sin_addr.s_addr = htonl (INADDR_ANY);             // Default IP address
17    // Create socket
18    if ((s = socket (PF_INET, SOCK_DGRAM, 0)) < 0);
```

Table 10.22 Echo server program using UDP 2

```
19      {
20          perror ("Error: socket failed!");
21          exit (1);
22      }
23      // Bind socket to local address and port
24      if ((bind (s, (struct sockaddr*) &servAddr, sizeof (servAddr))) < 0);
25      {
26          perror ("Error: bind failed!");
27          exit (1);
28      }
29      for ( ; ; )                 // Run forever
30      {
31                  // Receive String
32          len = recvfrom (s, buffer, sizeof (buffer), 0,
33             (struct sockaddr*)&clntAddr, &clntAddrLen);
34                  // Send String
35          sendto (s, buffer, len, 0, (struct sockaddr*)&clntAddr, sizeof(clntAddr));
36      } // End of for loop
37  } // End of echo server program
```

Table 10.23 Echo client program using UDP 1

```
1   // UDP echo client program
2   #include "headerFiles.h"
3   int main (int argc, char* argv[ ])          // Three arguments to be checked later
4   {
5     // Declare and define variables
6     int s;                                    // Socket descriptor
7     int len;                                  // Length of string to be echoed
8     char* servName;                           // Server name
9     int servPort;                             // Server port
10    char* string;                             // String to be echoed
11    char buffer [256 + 1];                    // Data buffer
12    struct sockaddr_in      servAddr;         // Server socket address
13    // Check and set program arguments
14    if (argc != 3)
15    {
16        printf ("Error: three arguments are needed!");
17        exit(1);
18    }
```

Table 10.23 Echo client program using UDP 2

```
19        servName = argv[1];
20        servPort = atoi (argv[2]);
21        string = argv[3];
22        // Build server socket address
23        memset (&servAddr, 0, sizeof (servAddr));
24        servAddr.sin_family = AF_INET;
25        inet_pton (AF_INET, servName, &servAddr.sin_addr);
26        servAddr.sin_port = htons (servPort);
27        // Create socket
28        if ((s = socket (PF_INET, SOCK_DGRAM, 0)) < 0);
29        {
30              perror ("Error: -Socket failed!");
31              exit (1);
32        }
33        // Send echo string
34        len = sendto (s, string, strlen (string), 0, (struct sockaddr)&servAddr, sizeof
          (servAddr));
35        // Receive echo string
36        recvfrom (s, buffer, len, 0, NULL, NULL);
37        // Print and verify echoed string
```

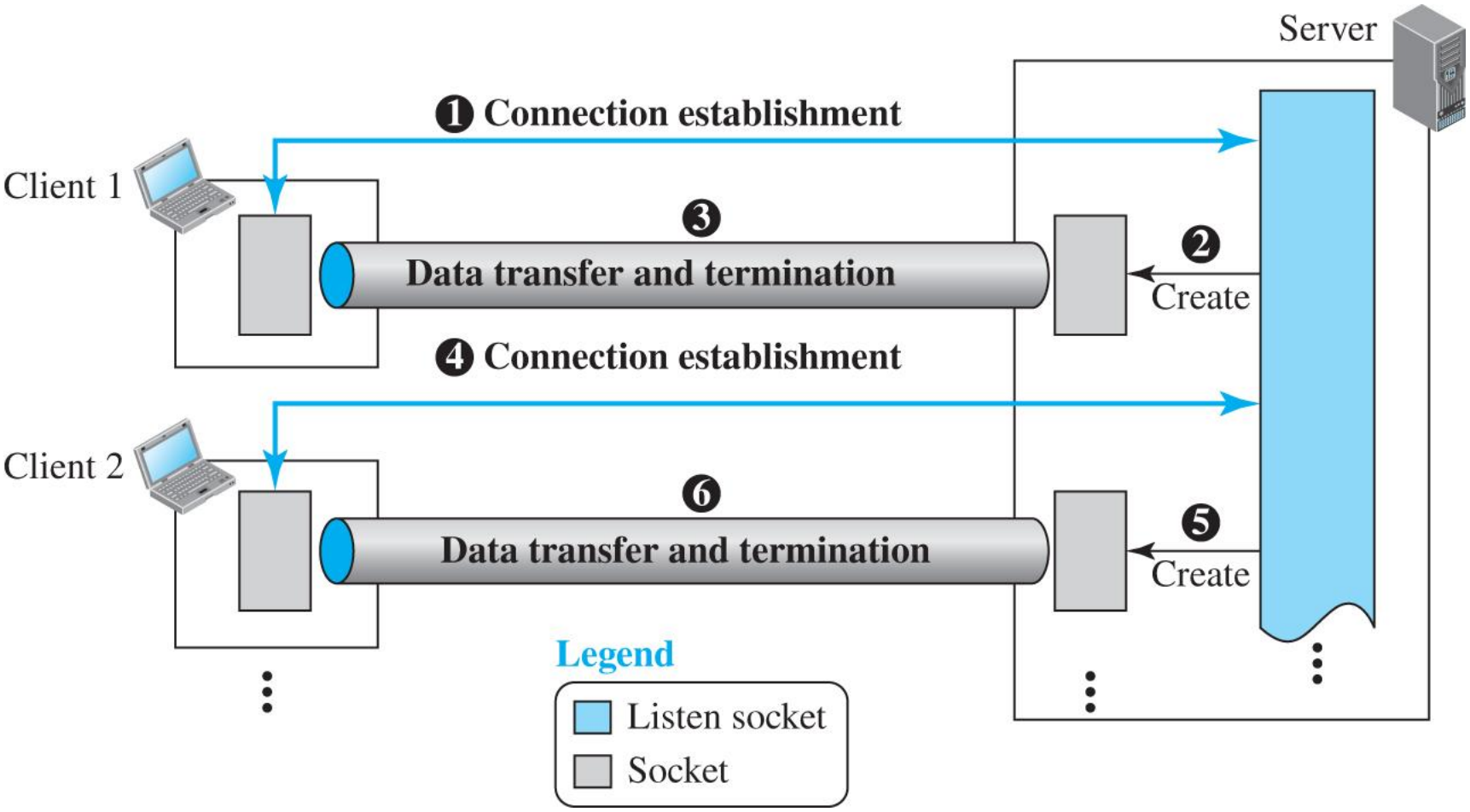
Table 10.23 Echo client program using UDP 3

```
38          buffer [len] = '\0';
39          printf ("Echo string received: ");
40          fputs (buffer, stdout);
41          // Close the socket
42          close (s);
43          // Stop the program
44          exit (0);
45      } // End of echo client program
```

Communication Using TCP

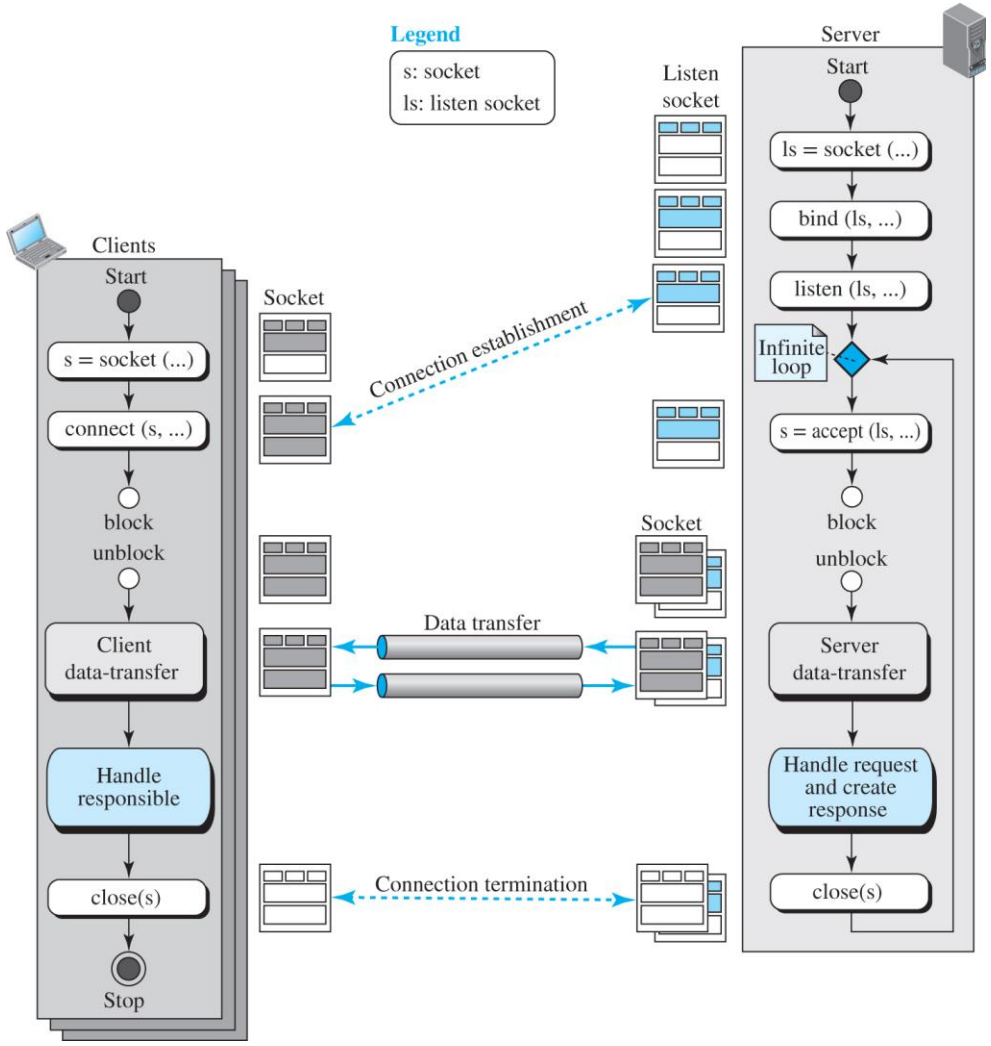
As we described before, TCP is a connection-oriented protocol. Before sending or receiving data, a connection needs to be established between the client and the server. After the connection is established, the two parties can send and receive chunks of data to each other as long as they have data to do so. TCP communication can be iterative (serving a client at a time) or concurrent (serving several clients at a time). In this section, we discuss only the iterative approach.

Figure 10.61 Sockets used in TCP communication



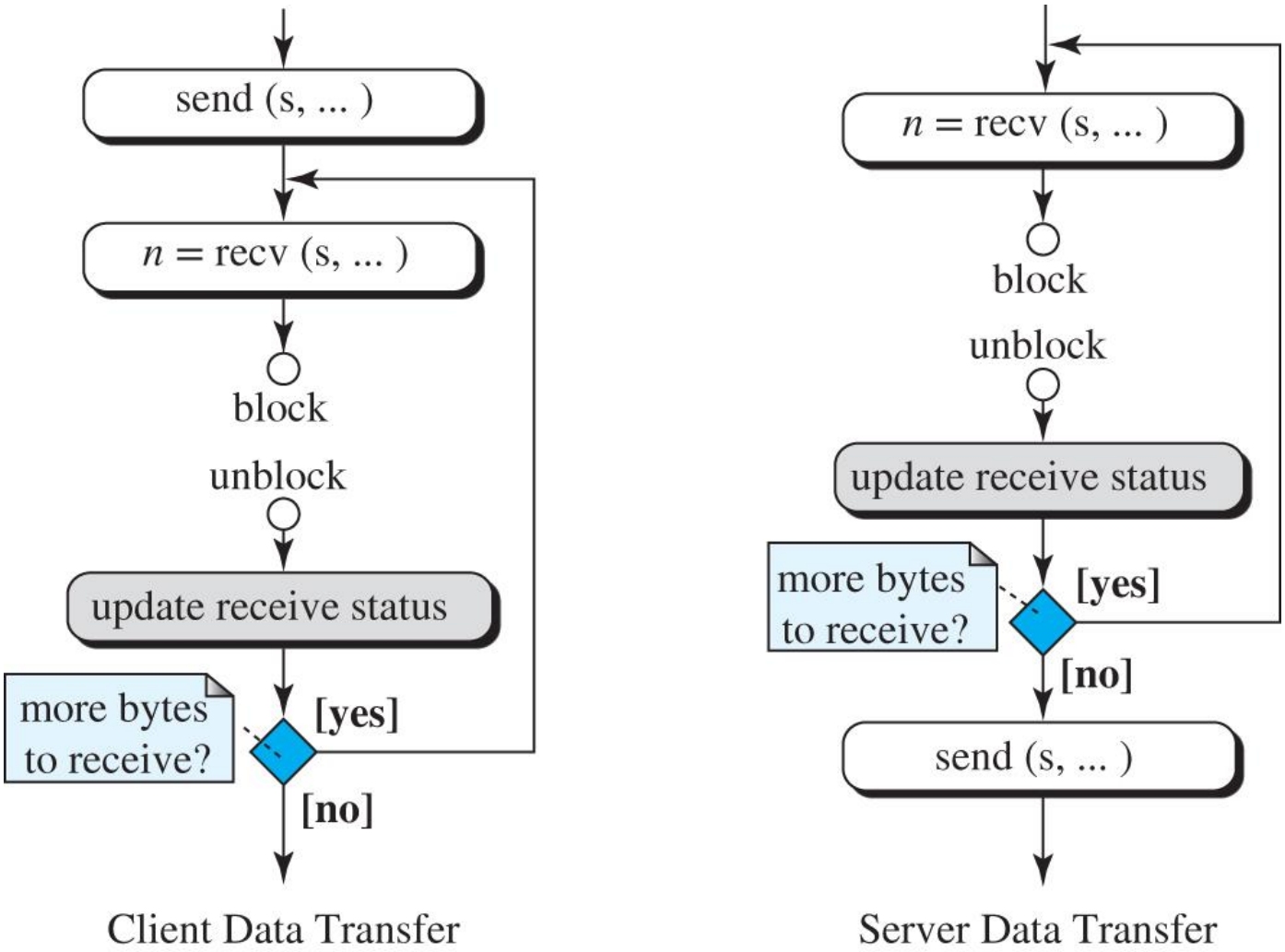
Access the text alternative for slide images.

Figure 10.62 Flow diagram for iterative TCP communication



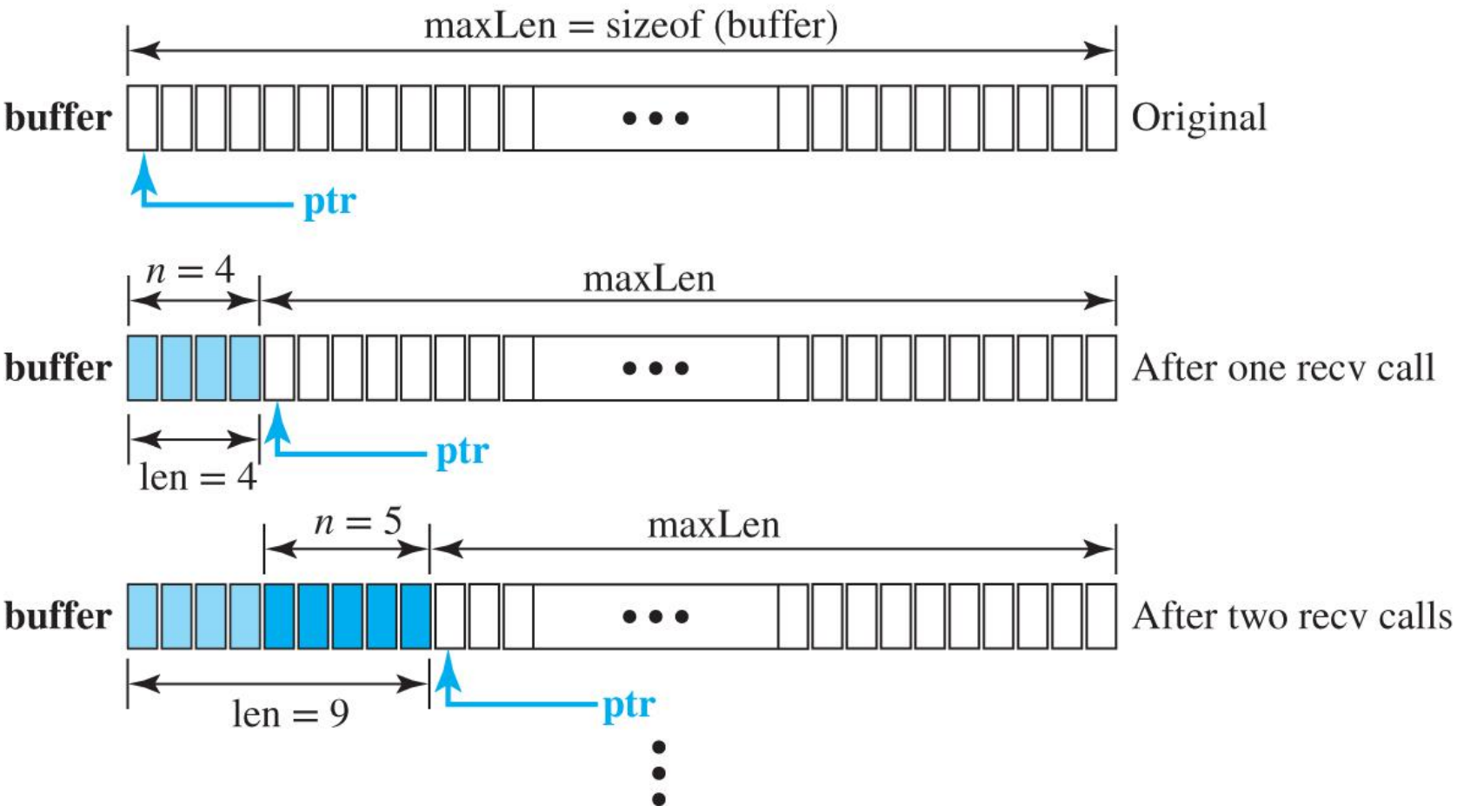
Access the text alternative for slide images.

Figure 10.63 Flow diagram for data-transfer boxes



Access the text alternative for slide images.

Figure 10.64 Buffer used for receiving



Access the text alternative for slide images.

Table 10.24 Echo server program using the services of TCP 1

```
1   // Echo server program
2   #include "headerFiles.h"
3   int main (void)
4   {
5      // Declare and define
6      int ls;                                  // Listen socket descriptor -(reference)
7      int s;                                   // socket descriptor (reference)
8      char buffer [256];                       // Data buffer
9      char* ptr = buffer;                      // Data buffer
10     int len = 0;                             // Number of bytes to send or -receive
11     int maxLen = sizeof (buffer);            // Maximum number of bytes to receive
12     int n = 0;                               // Number of bytes for each recv call
13     int waitSize = 16;                       // Size of waiting clients
14     struct sockaddr_in serverAddr;           // Server address
15     struct sockaddr_in clientAddr;           // Client address
16     int clntAddrLen;                         // Length of client address
17     // Create local (server) socket address
18     memset (&servAddr, 0, sizeof (servAddr));
19     servAddr.sin_family = AF_INET;
20     servAddr.sin_addr.s_addr = htonl (INADDR_ANY);          // Default IP address
```

Table 10.24 Echo server program using the services of TCP 2

```
21      servAddr.sin_port = htons (SERV_PORT);                    // Default port
22      // Create listen socket
23      if (ls = socket (PF_INET, SOCK_STREAM, 0) < 0);
24      {
25          perror ("Error: Listen socket failed!");
26          exit (1);
27      }
28      // Bind listen socket to the local socket address
29      if (bind (ls, &servAddr, sizeof (servAddr)) < 0);
30      {
31          perror ("Error: binding failed!");
32          exit (1);
33      }
34      // Listen to connection requests
35      if (listen (ls, waitSize) < 0);
36      {
37          perror ("Error: listening failed!");
38          exit (1);
39      }
40      // Handle the connection
```

Table 10.24 Echo server program using the services of TCP 3

```
41          for ( ; ; )                    // Run forever
42          {
43              // Accept connections from client
44              if (s = accept (ls, &clntAddr, &clntAddrLen) < 0);
45              {
46                  perror ("Error: accepting failed!);
47                  exit (1);
48              }
49          // Data transfer section
50          while ((n = recv (s, ptr, maxLen, 0)) > 0)
51              {
52                  ptr + = n;             // Move pointer along the buffer
53                  maxLen −= n;           // Adjust maximum number of bytes to receive
54                  len + = n;             // Update number of bytes received
55              }
56          send (s, buffer, len, 0);      // Send back (echo) all bytes received
57              // Close the socket
58                  close (s);
59          } // End of for loop
60  } // End of echo server program
```

Table 10.25 Echo client program using the services of TCP

```
1   // TCP echo client program
2   #include "headerFiles.h"
3   int main (int argc, char* argv[ ])          // Three arguments to be checked later
4   {
5      // Declare and define
6      int s;                                   // Socket descriptor
7      int n;                                   // Number of bytes in each recv call
8      char* servName;                          // Server name
9      int servPort;                            // Server port number
10     char* string;                            // String to be echoed
11     int len;                                 // Length of string to be echoed
12     char buffer [256 + 1];                   // Buffer
13     char* ptr = buffer;                      // Pointer to move along the buffer
14     struct sockaddr_in serverAddr;           // Server socket address
15     // Check and set arguments
16     if (argc != 3)
17     {
18         printf ("Error: three arguments are needed!");
19         exit (1);
20     }
```

Table 10.25 TCP Echo client program 1

```
21      servName = arg [1];
22      servPort = atoi (arg [2]);
23      string = arg [3];
24      // Create remote (server) socket address
25      memset (&servAddr, 0, sizeof(servAddr));
26      serverAddr.sin_family = AF_INET;
27      inet_pton (AF_INET, servName, &serverAddr.sin_addr);      // Server IP address
28      serverAddr.sin_port = htons (-servPort);                  // Server port number
29      // Create socket
30      if ((s = socket (PF_INET, SOCK_STREAM, 0)) < 0);
31      {
32          perror ("Error: socket creation failed!");
33          exit (1);
34      }
35      // Connect to the server
36      if (connect (sd, (struct sockaddr*)&servAddr, sizeof(servAddr)) < 0);
37      {
38          perror ("Error: connection failed!");
```

Table 10.25 TCP Echo client program 2

```
39              exit (1);
40          }
41          // Data transfer section
42          send (s, string, strlen(string), 0);
43          while ((n = recv (s, ptr, maxLen, 0)) > 0)
44          {
45              ptr + = n;                  // Move pointer along the buffer
46              maxLen −= n;                // Adjust the maximum number of bytes
47              len + = n;                  // Update the length of string received
48          } // End of while loop
49          // Print and verify the echoed string
50          buffer [len] = '\0';
51          printf ("Echoed string received: ");
52          fputs (buffer, stdout);
53          // Close socket
54          close (s);
55          // Stop program
56          exit (0);
57      } // End of echo client program
```

Because learning changes everything.®

www.mheducation.com