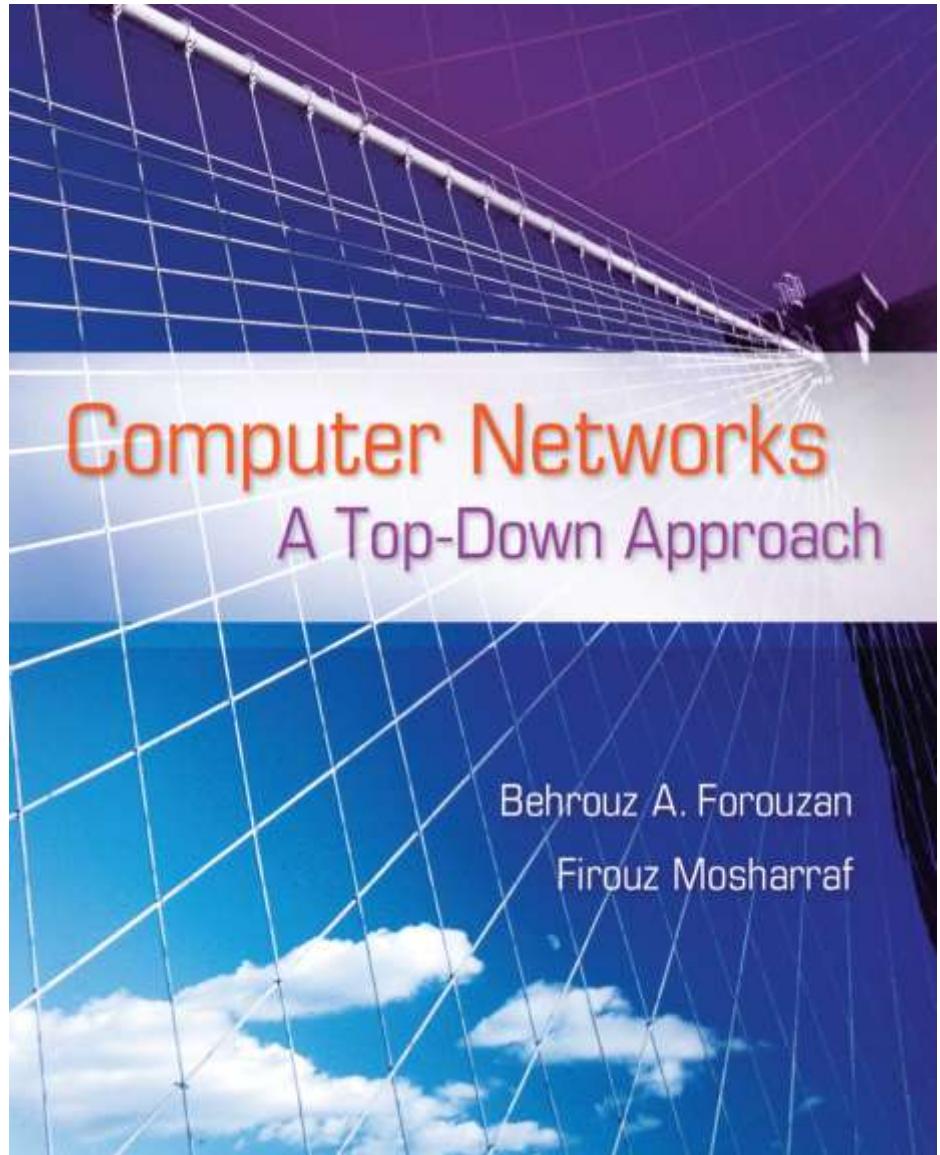
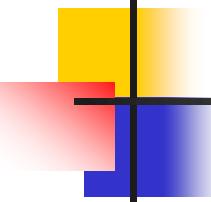


# *Chapter 2*

## *Application Layer*





# **Chapter 2: Outline**

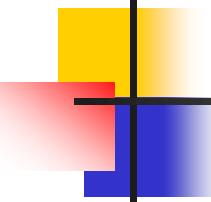
***2.1 INTRODUCTION***

***2.2 CLIENT-SERVER PARADIGM***

***2.3 STANDARD APPLICATIONS***

***2.4 PEER-TO-PEER PARADIGM***

***2.5 SOCKET-INTERFACE PROGRAMMING***



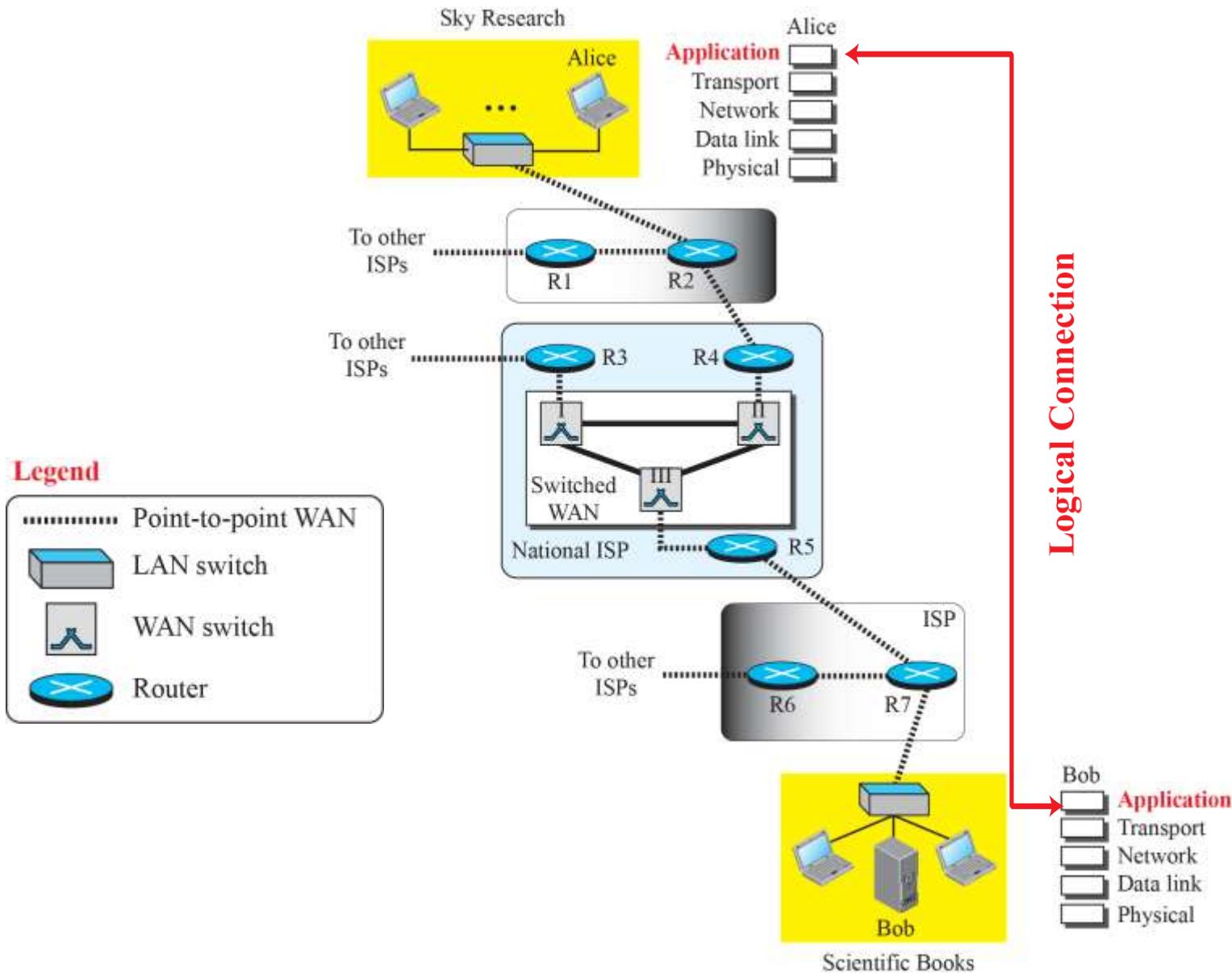
# Chapter 2: Objective

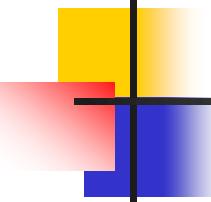
- *We introduce the nature of services provided by the Internet: the client-server paradigm and the peer-to-peer paradigm.*
- *We discuss the concept of the client-server paradigm.*
- *We discuss some predefined or standard applications based on the client-server paradigm such as surfing the Web, file transfer, e-mail, and so on.*
- *We discuss the concept and protocols in the peer-to-peer paradigm such as Chord, Pastry, and Kademlia.*
- *We show how a new application can be created in the client-server paradigm by writing two programs in the C language.*

## 2-1 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 2.1 shows the idea behind this logical connection.*

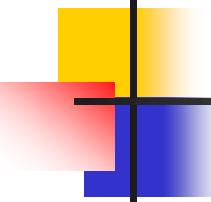
**Figure 2.1: Logical connection at the application layer**





## *2.1.1 Providing Services*

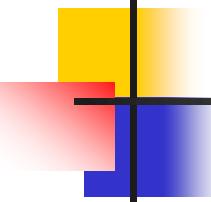
*The Internet was originally designed to provide service to users around the world. Since the application layer is the only layer that provides services to the Internet user, it allows new application protocols to be easily added to the Internet, which has been occurring during the lifetime of the Internet. When the Internet was created, only a few application protocols were available to the users; today we cannot give a number for these protocols because new ones are being added constantly.*



## *2.1.1 Providing Services (Cont.)*

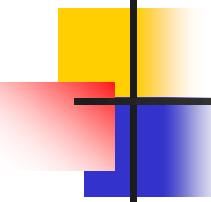
### *Standard and Nonstandard Protocols*

- ❖ *Standard Application-Layer Protocols*
- ❖ *Nonstandard Application-Layer Protocols*



## ***2.1.2 Application-Layer Paradigm***

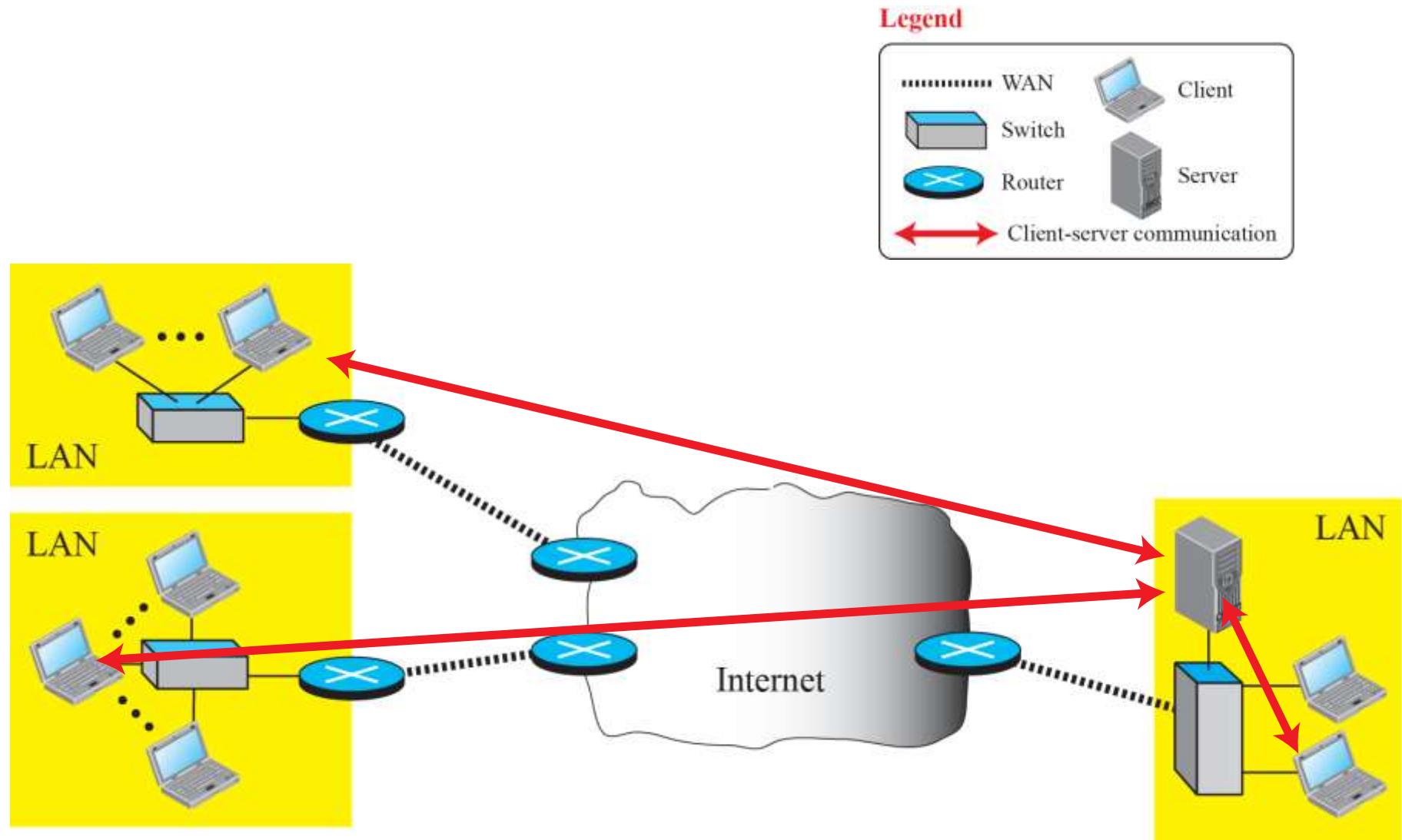
*It should be clear that to use the Internet we need two application programs to interact with each other: one running on a computer somewhere in the world, the other running on another computer somewhere else in the world. The two programs need to send messages to each other through the Internet infrastructure. However, we have not discussed what the relationship should be between these programs. Should both application programs be able to request services and provide services, or should the application programs just do one or the other?*



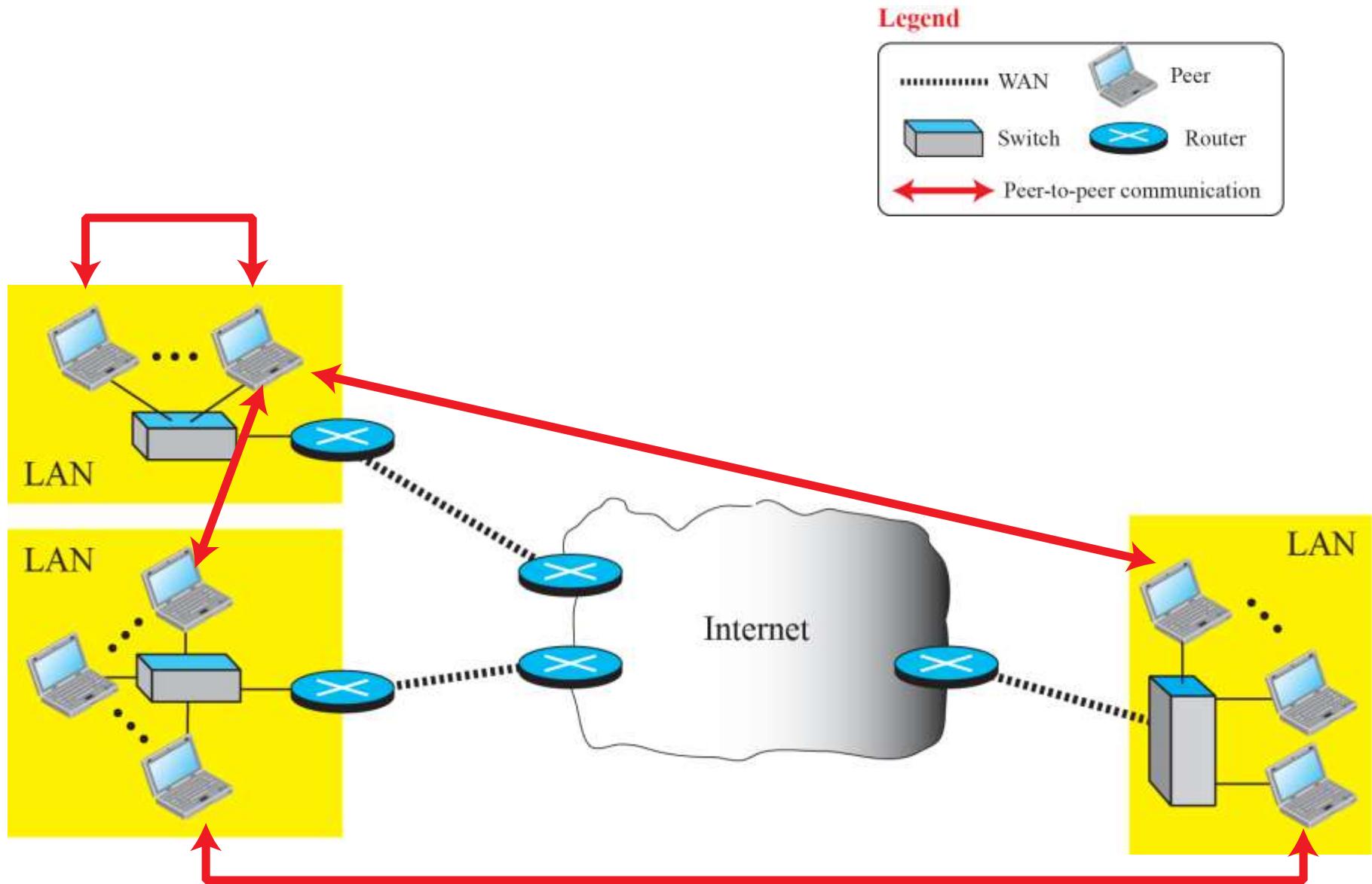
## **2.1.2 Application-Layer Paradigm (cont)**

- Traditional Paradigm: Client-Server***
- New Paradigm: Peer-to-Peer***
- Mixed Paradigm***

**Figure 2.2: Example of a client-server paradigm**

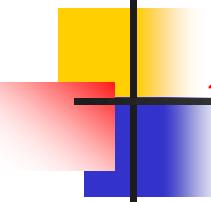


**Figure 2.3: Example of a peer-to-peer paradigm**



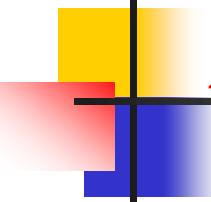
## 2-2 CLIENT-SERVER PARADIGM

*In this 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.*



## **2.2.1 Application Programming Interface**

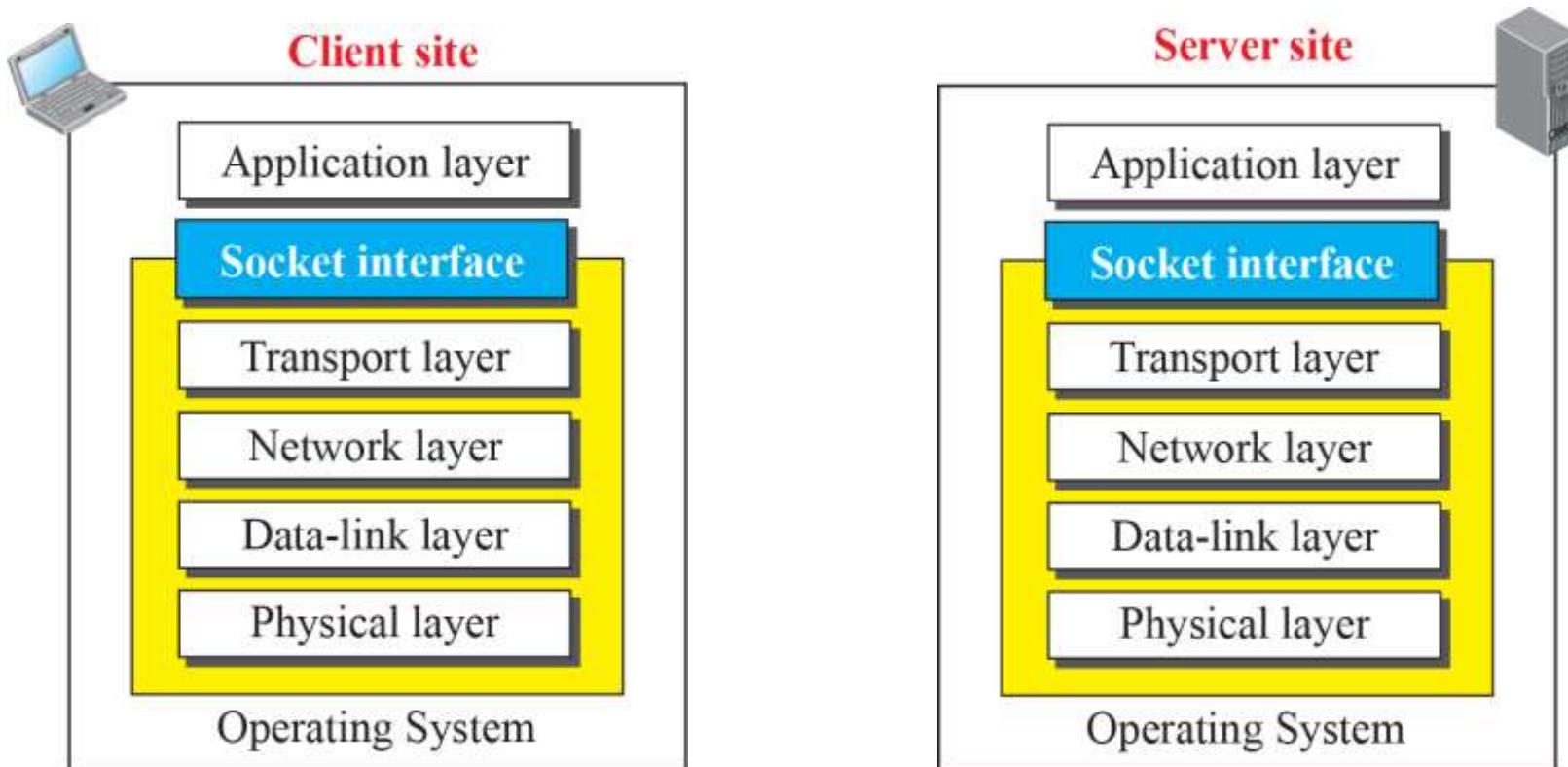
*A computer language has a set of instructions for mathematical operations, a set of instructions for string manipulation, a set of instructions for input/output access, and so on. If we need a process to be able to communicate with another process, we need a new set of instructions to tell the lowest 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 Application Programming Interface (API).*



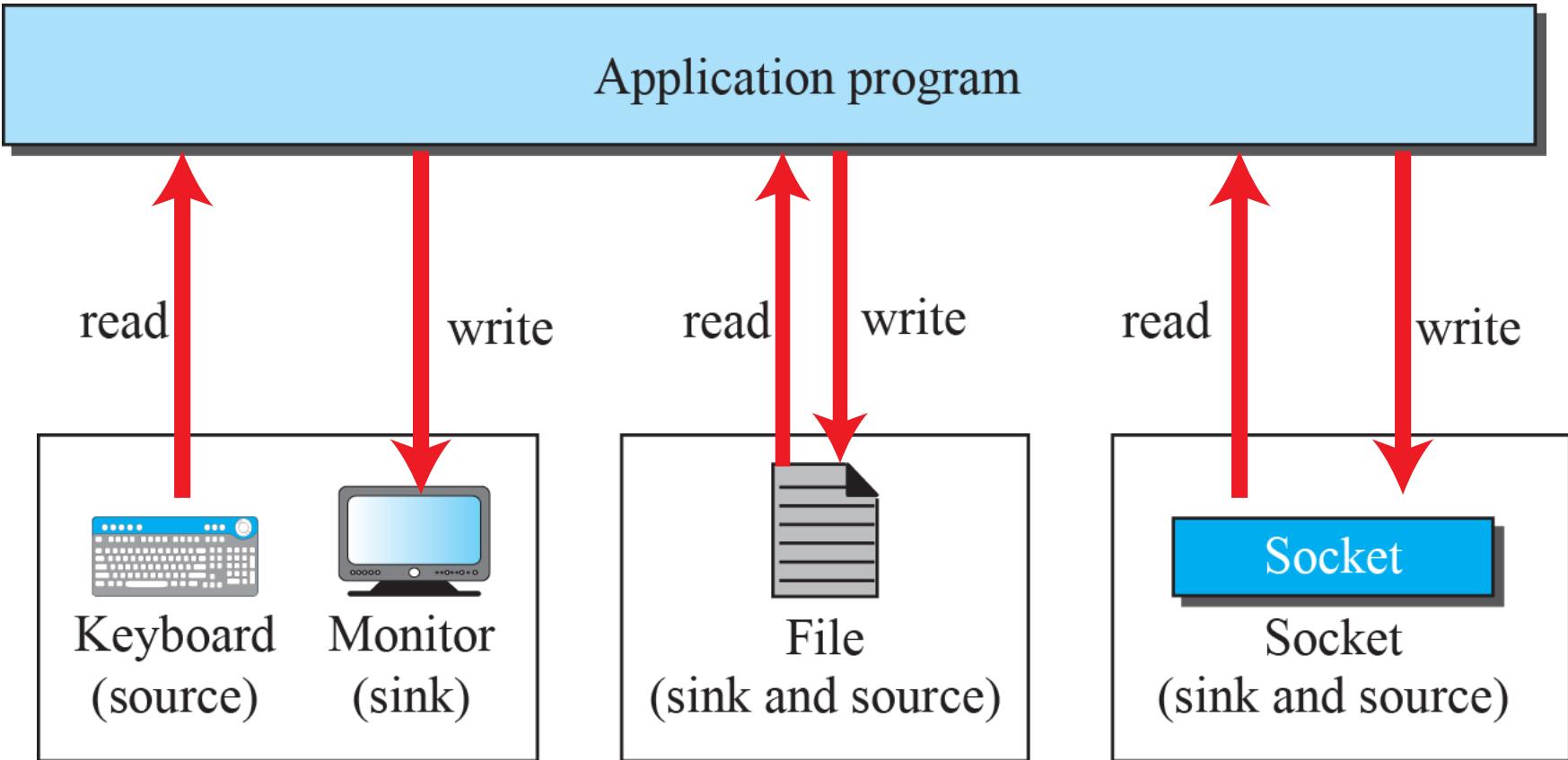
## 2.2.1 (*continued*)

- *Sockets*
- *Socket Addresses*
- *Finding Socket Addresses*
  - ❖ *Server Site*
  - ❖ *Client Site*

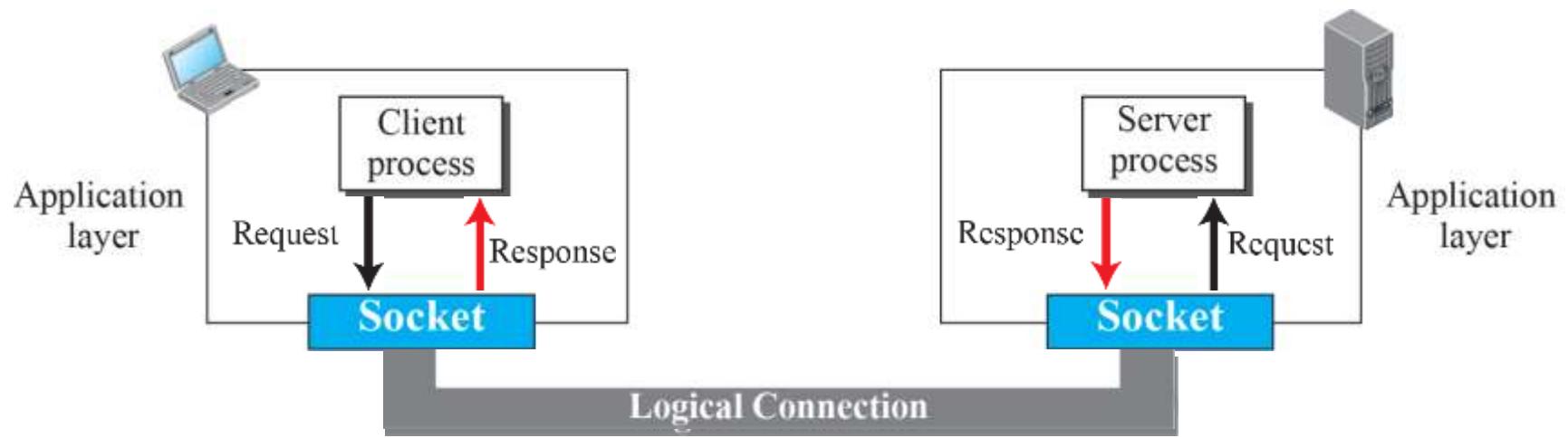
**Figure 2.4: Position of the socket interface**



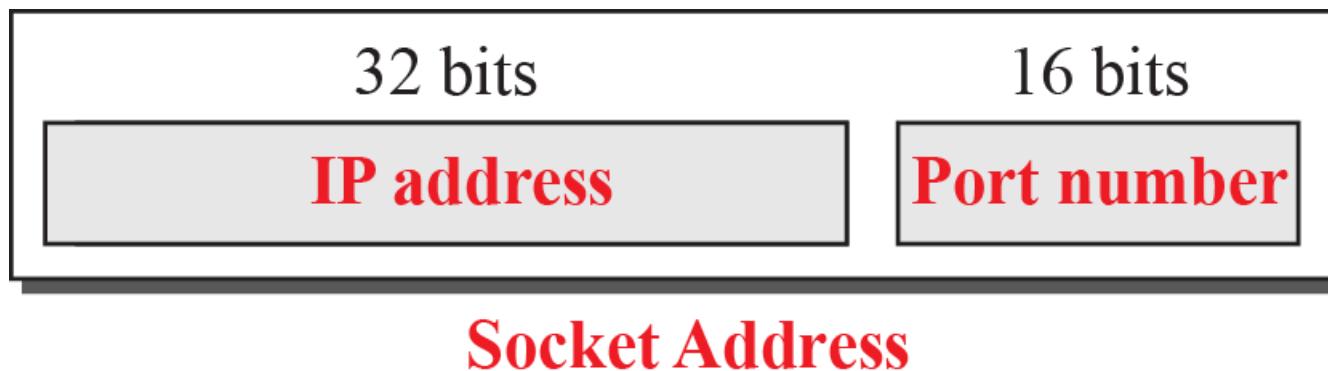
**Figure 2.5: A Sockets used like other sources and sinks**



**Figure 2.6: Use of sockets in process-to-process communication**

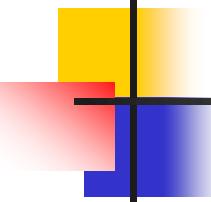


**Figure 2.7: A socket address**



## ***Example 2.1***

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.



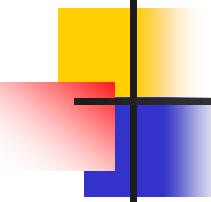
## **2.2.2 Using Services of Transport Layer**

*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. There are three common transport layer protocols in the TCP/IP suite: UDP, TCP, and SCTP.*

- UDP Protocol***
- TCP Protocol***
- SCTP Protocol***

## 2-3 STANDARD CLIENT-SERVER APPLICATIONS

*During the lifetime of the Internet, several 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 can help us to create customized applications in the future.*

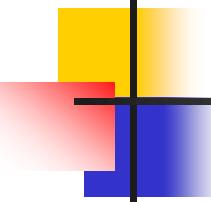


## **2.3.1 World Wide Web and HTTP**

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

## 2.3.1 (*continued*)

- *World Wide Web*
  - ❖ *Architecture*
  - ❖ *Uniform Resource Locator (URL)*
  - ❖ *Web Documents*
  
- *HyperText Transfer Protocol (HTTP)*
  - ❖ *Nonpersistent versus Persistent Connections*
  - ❖ *Message Formats*
  - ❖ *Conditional Request*
  - ❖ *Cookies*



## **2.3.1 (continued)**

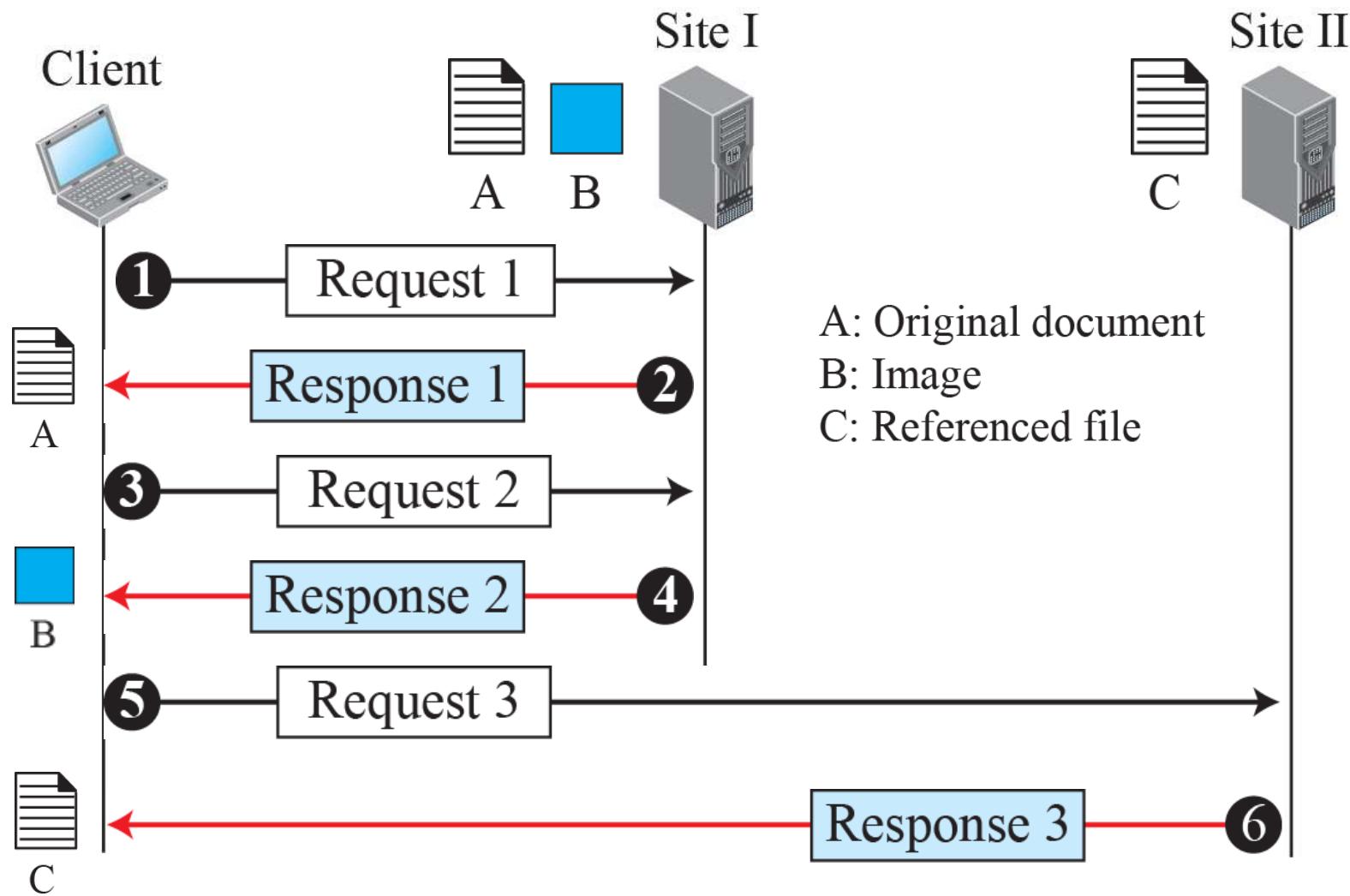
- Web Caching: Proxy Server***
  - ❖ *Proxy Server Location*
  - ❖ *Cache Update*
- HTTP Security***

## *Example 2.2*

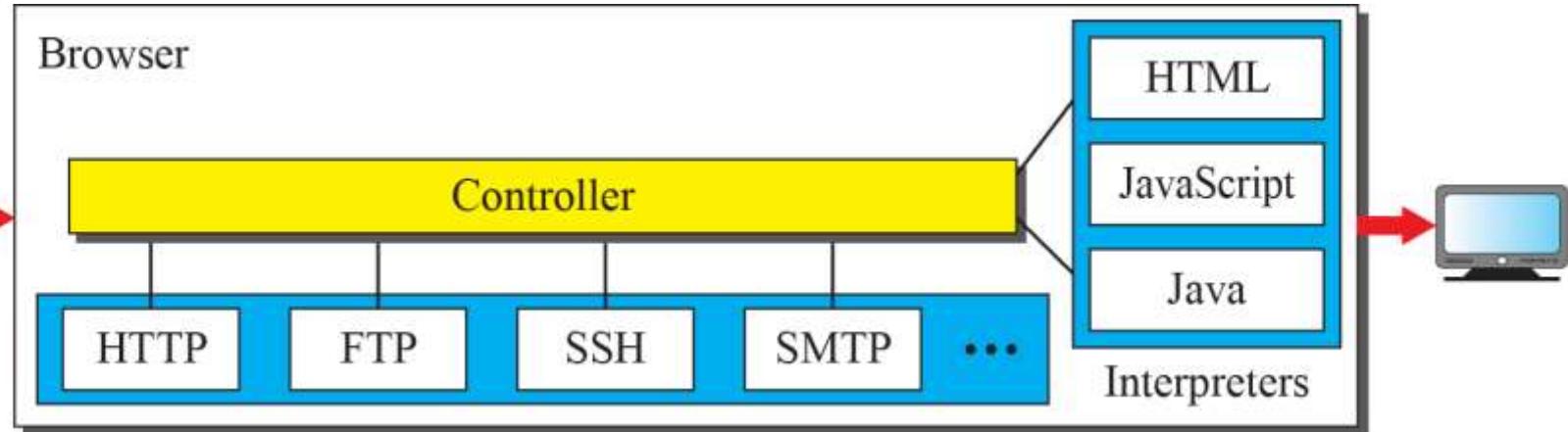
Assume we need to retrieve a scientific document that contains one reference to another text file and one reference to a large image. Figure 2.8 shows the situation.

The main document and the image are stored in two separate files in the same site (file A and file B); the referenced text file is stored in another site (file C). Since we are dealing with three different files, we need three transactions if we want to see the whole document.

**Figure 2.8: Example 2.2 (Retrieving two files and one image)**



**Figure 2.9: Browser**



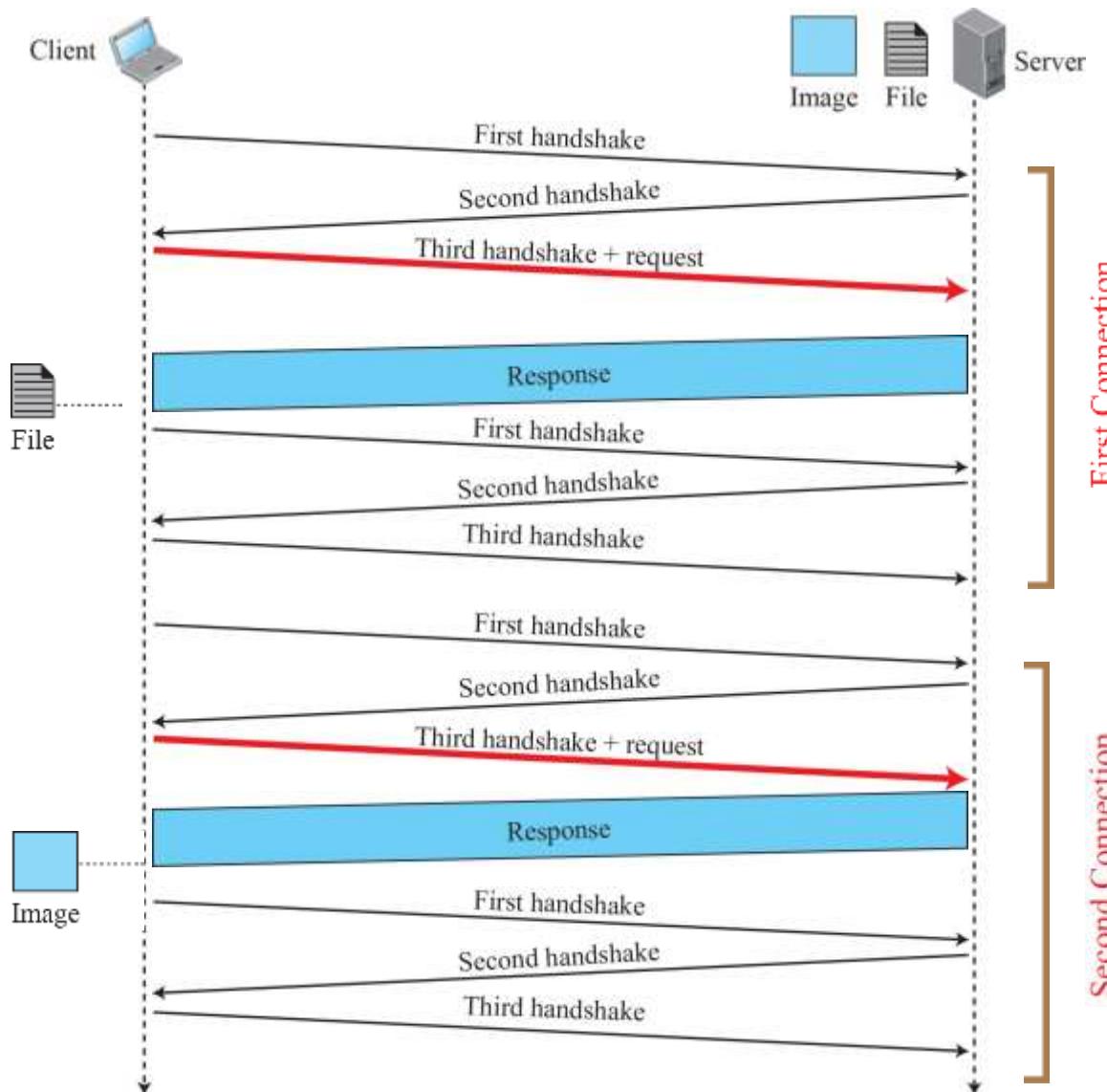
## **Example 2.3**

The URL <http://www.mhhe.com/compsci/forouzan/> defines the web page related to one of the computer 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).

## **Example 2.4**

Figure 2.10 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, as we will see in Chapter 3.

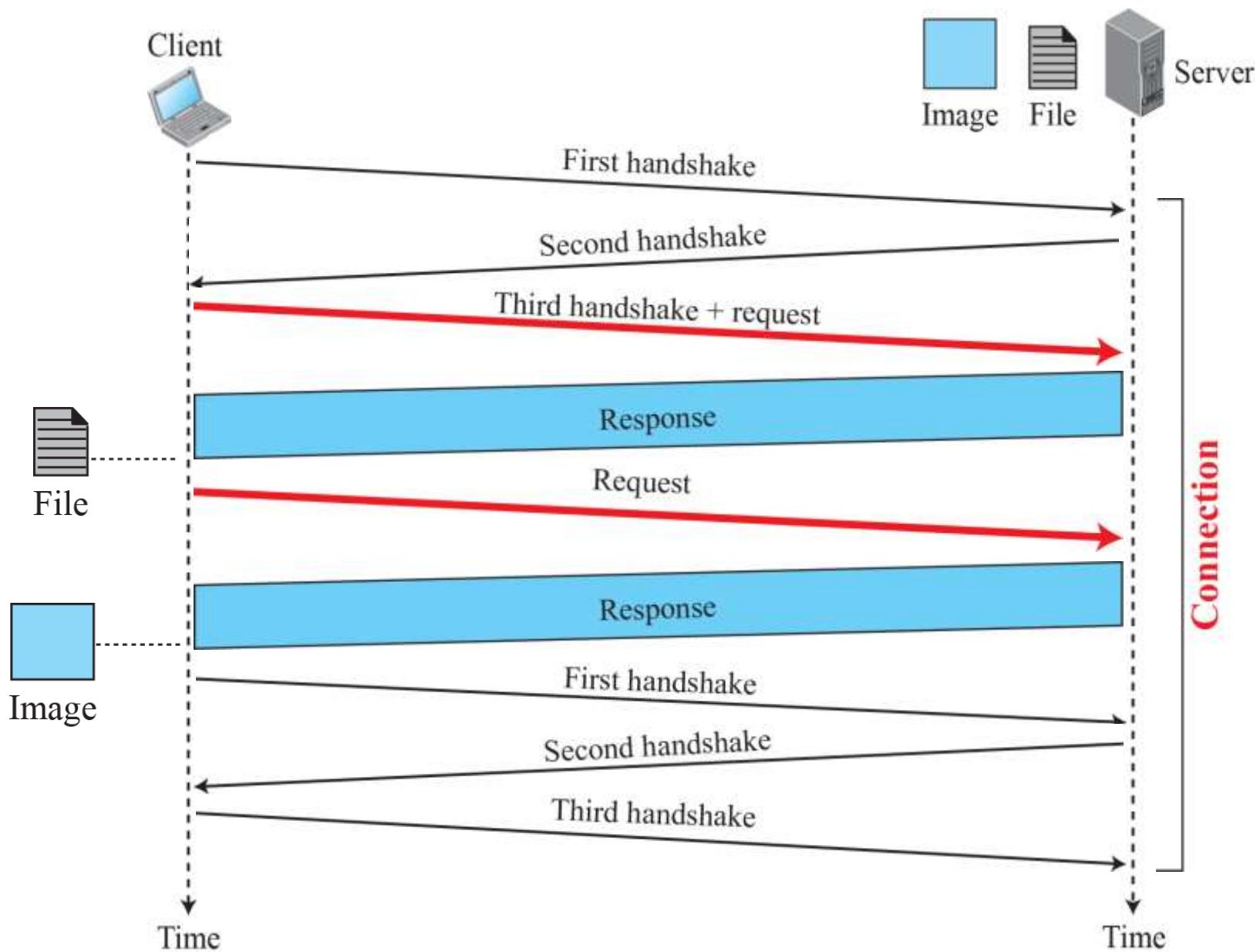
**Figure 2.10: Example 2.4**



## ***Example 2.5***

Figure 2.11 shows the same scenario as in Example 2.4, but using a persistent connection. Only one connection establishment and connection termination is used, but the request for the image is sent separately.

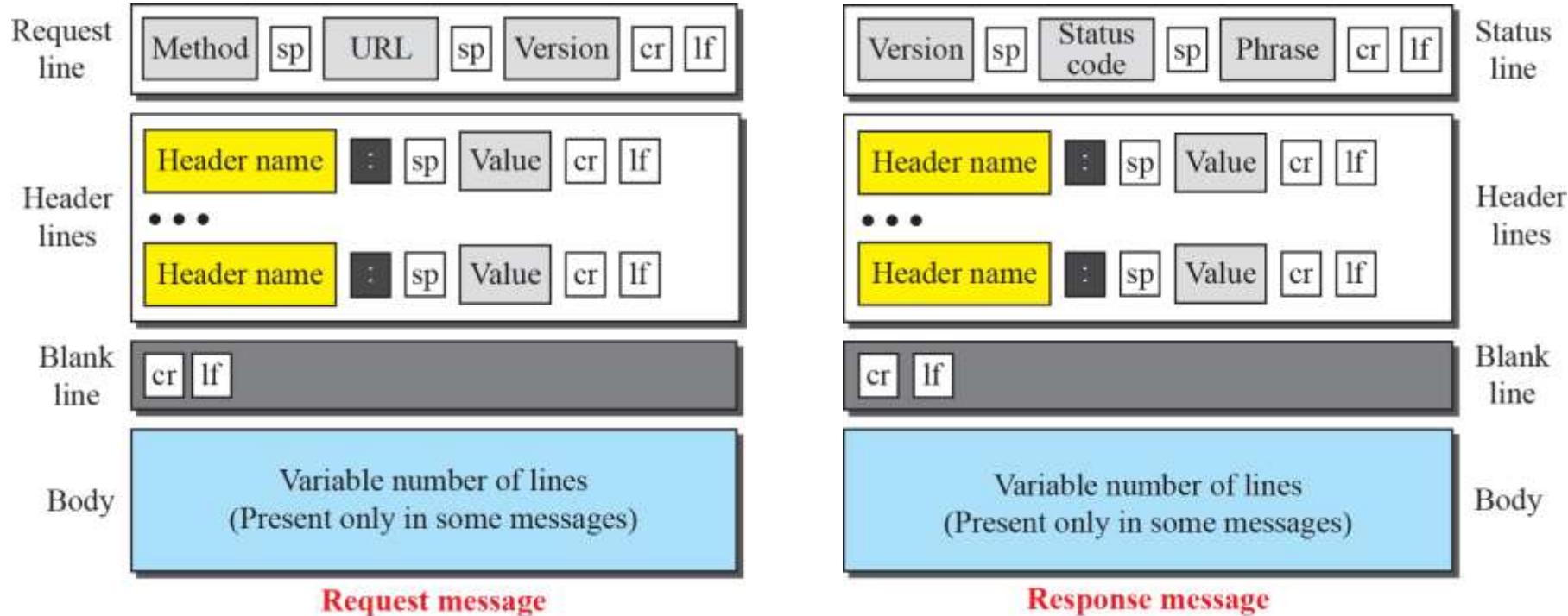
**Figure 2.11: Example 2.5**

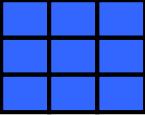


## **Figure 2.12: Formats of the request and response messages**

### **Legend**

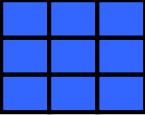
sp: Space cr: Carriage Return lf: Line Feed





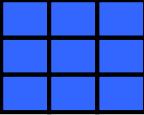
## Table 2.1: Methods

<i>Method</i>	<i>Action</i>
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 2.2: Request Header Names

<i>Header</i>	<i>Description</i>
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)
If-Modified-Since	If the file is modified since a specific date

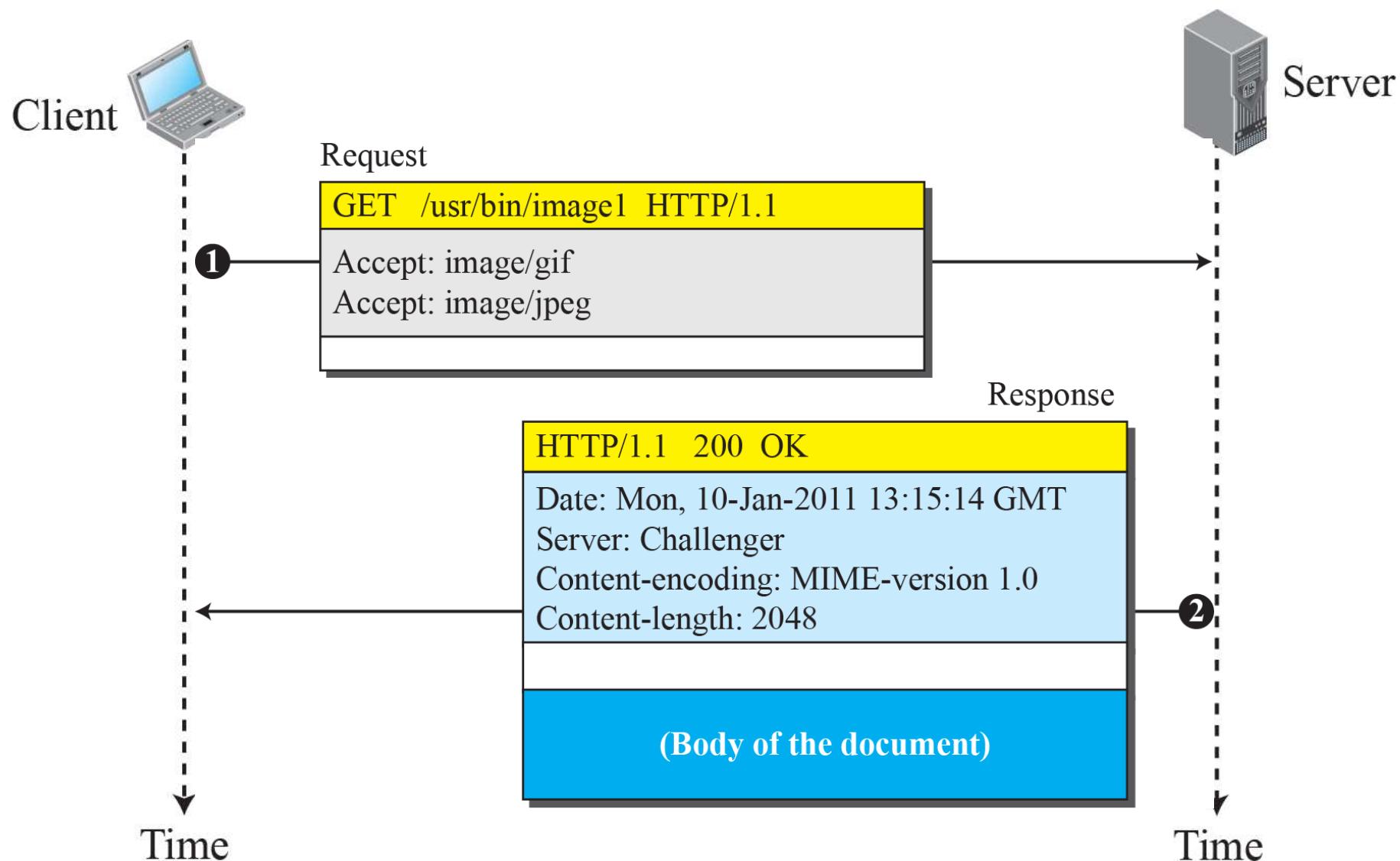


## Table 2.3: Response Header Names

<i>Header</i>	<i>Description</i>
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

This example retrieves a document (see Figure 2.13). 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..

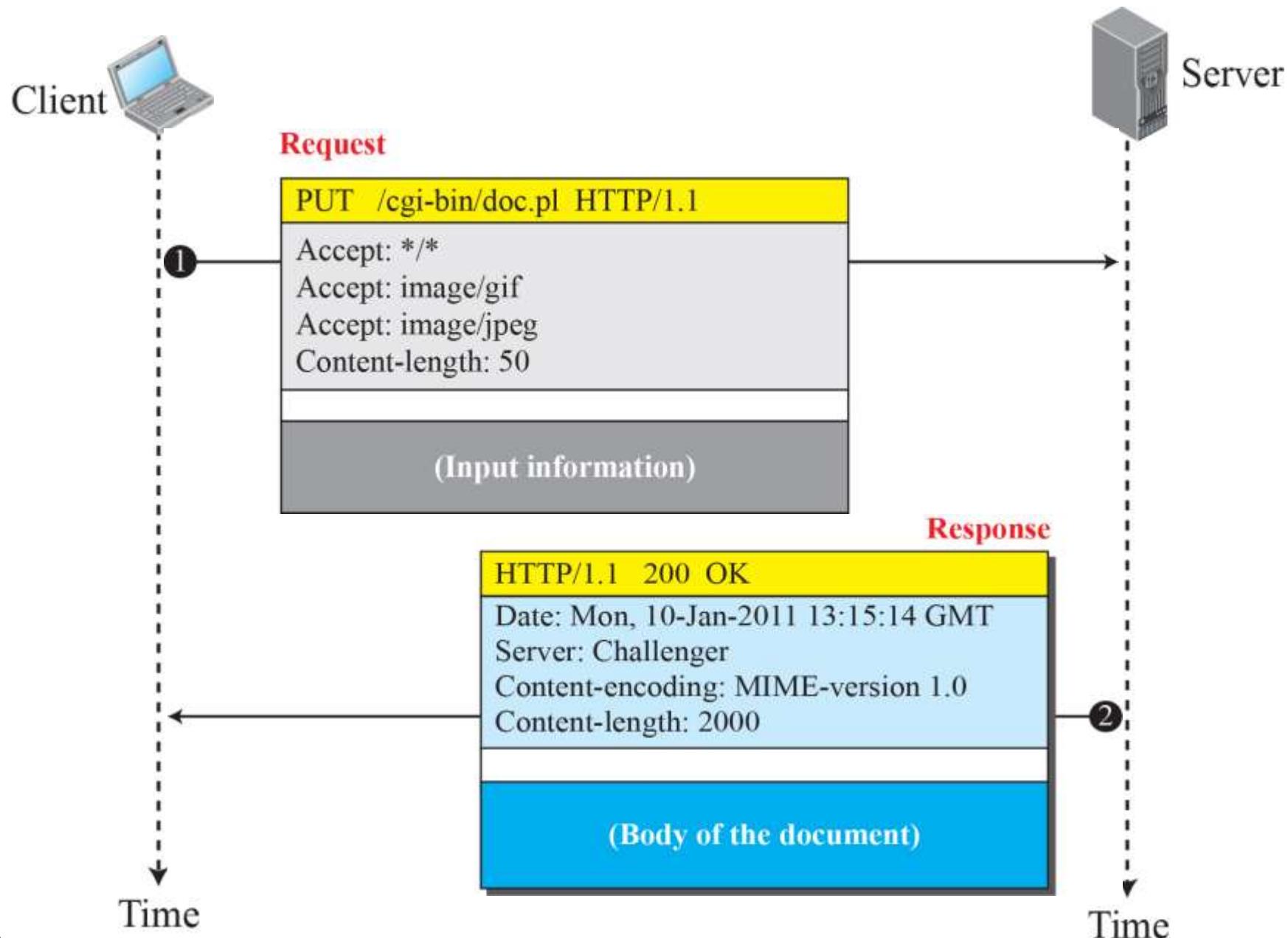
**Figure 2.13: Example 2.6**



## ***Example 2.7***

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 2.14).

**Figure 2.14: Example 2.7**



## *Example 2.8*

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**

(Empty Body)

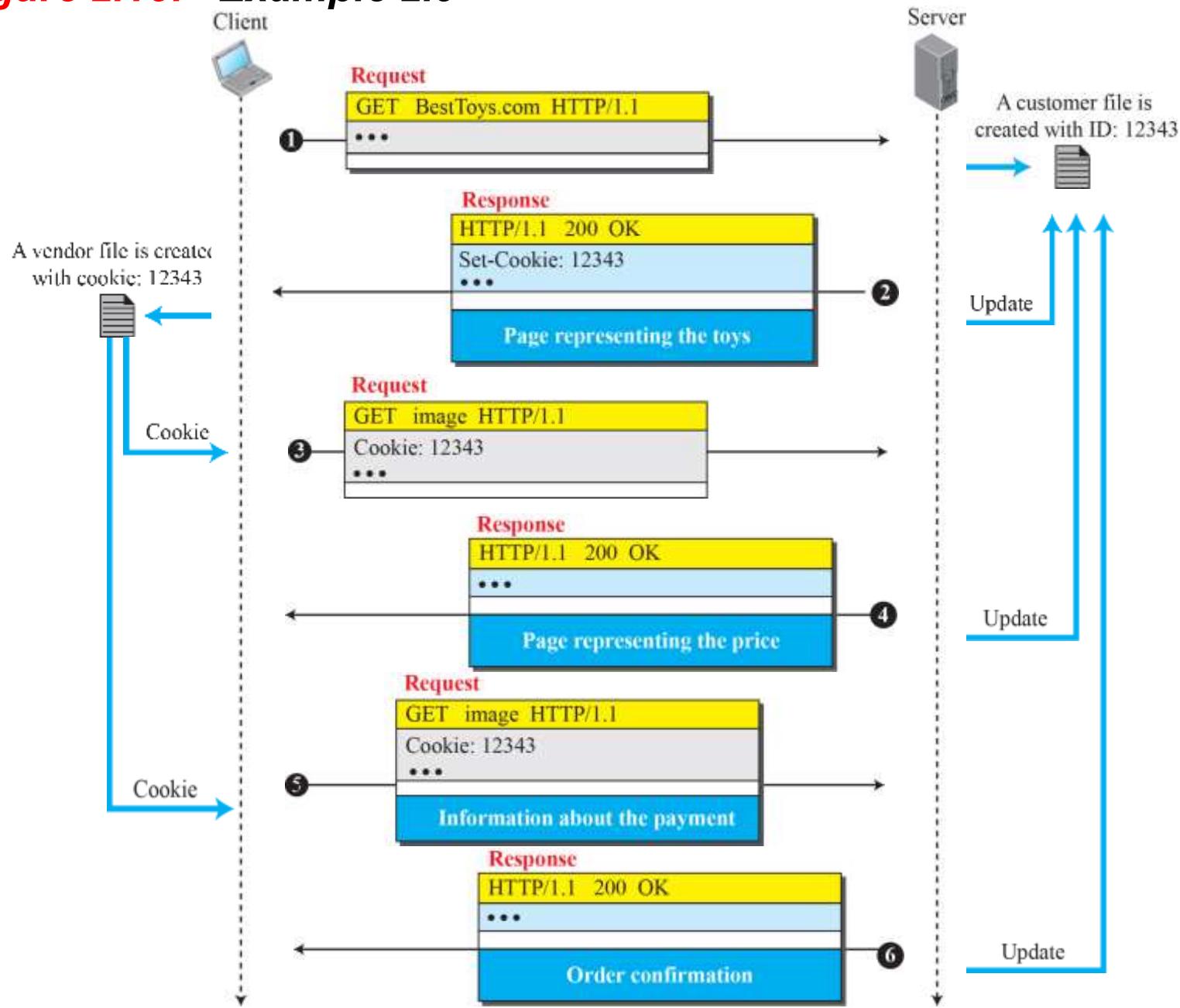
**Blank line**

**Empty body**

## **Example 2.9**

Figure 2.15 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.

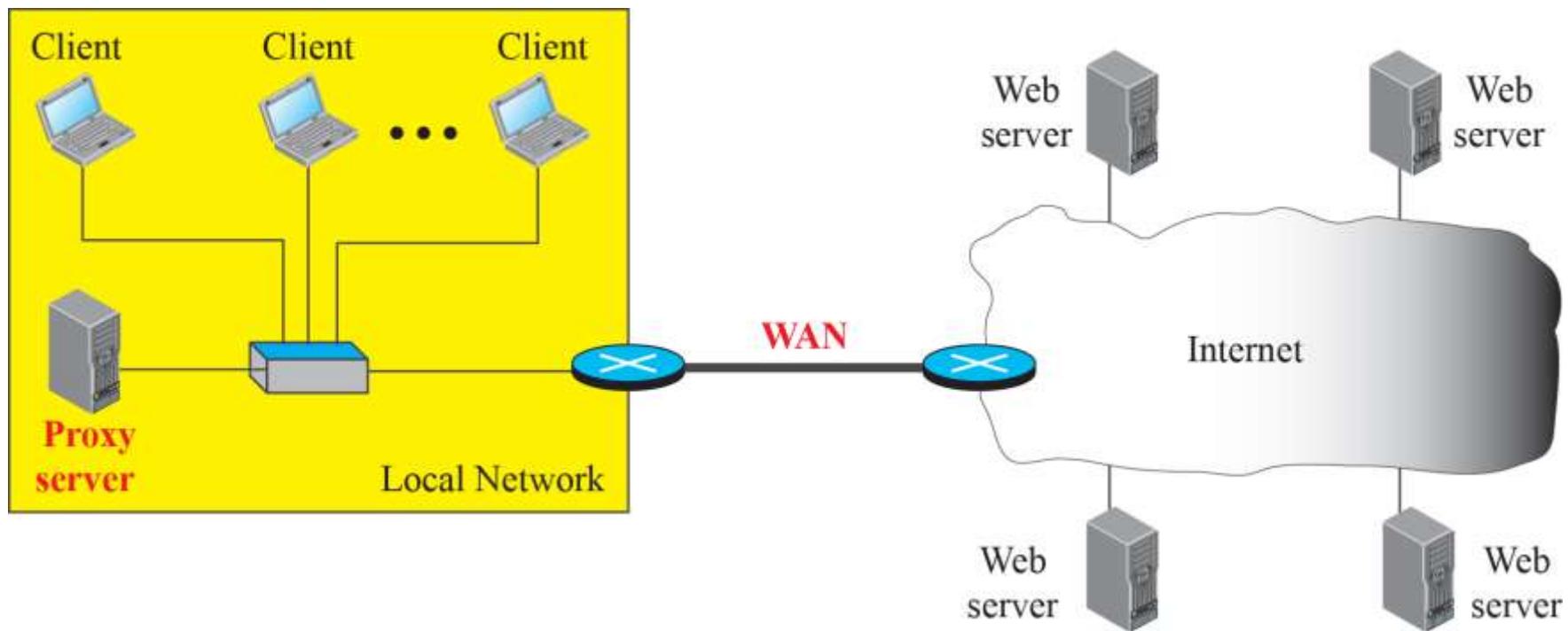
**Figure 2.15: Example 2.9**

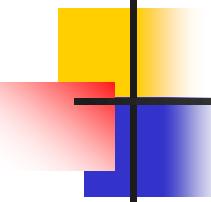


## **Example 2.10**

Figure 2.16 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.

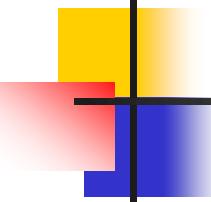
**Figure 2.16:** Example of a proxy server





## **2.3.2 *FTP***

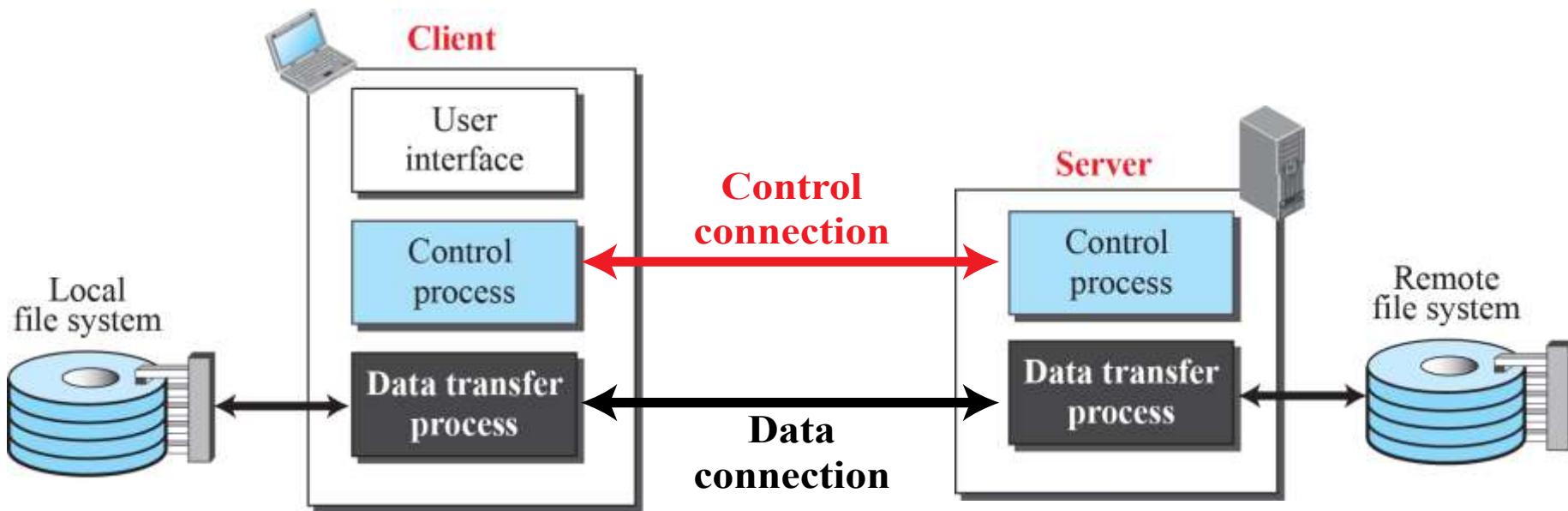
*File Transfer Protocol (FTP) is the standard protocol provided by TCP/IP for copying a file from one host to another. Although transferring files from one system to another seems simple and straightforward, some problems must be dealt with first. For example, two systems may use different file name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

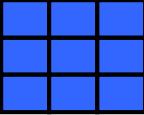


## 2.3.2 (*continued*)

- ❑ *Lifetimes of Two Connections*
- ❑ *Control Connection*
- ❑ *Data Connection*
  - ❖ *Communication over Data Connection*
  - ❖ *File Transfer*
- ❑ *Security for FTP*

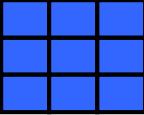
**Figure 2.17: FTP**





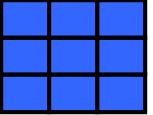
## Table 2.4: Some FTP commands

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



## Table 2.4: Some FTP commands (continued)

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



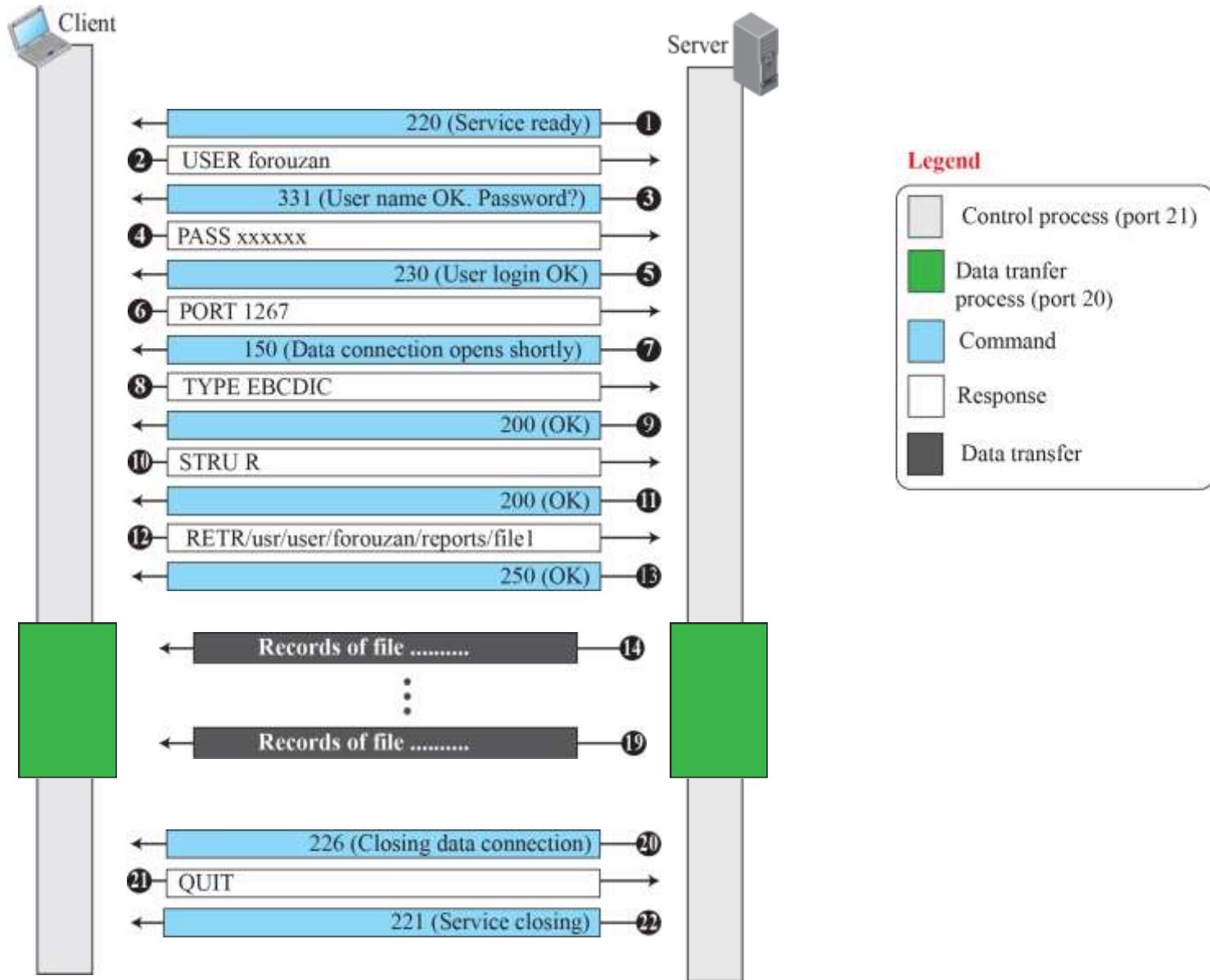
## Table 2.5: Some responses in FTP

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

## ***Example 2.11***

Figure 2.18 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.

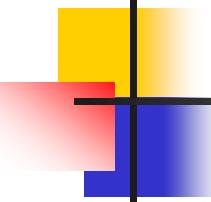
**Figure 2.18: Example 2.11**



## Example 2.12

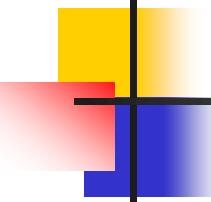
The following shows an actual FTP session that lists the directories.

```
$ 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.
```



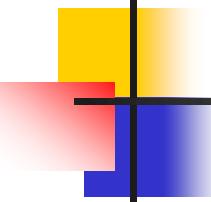
### **2.3.3 Electronic Mail**

*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. In the case of electronic mail, the situation is different.*



## **2.3.3 *Continued***

*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. Bob may or may not respond. If he does respond, it is another one-way transaction. Second, it is neither feasible nor logical for Bob to run a server program and wait until someone sends an e-mail to him. Bob may turn off his computer when he is not using it. This means that the idea of client/ server programming should be implemented in another way: using some intermediate computers (servers).*

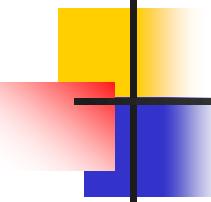


## **2.3.3 (continued)**

### **□ *Architecture***

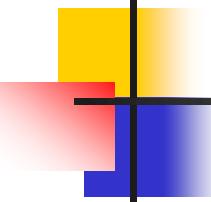
### **□ *User Agent***

- ❖ *Sending Mail*
- ❖ *Receiving Mail*
- ❖ *Addresses*
- ❖ *Mailing List or Group List*



## ***2.3.1 (continued)***

- MIME***
  - ❖ *MIME Headers*
- Web-Based Mail***
  - ❖ *Case I*
  - ❖ *Case II*
- E-Mail Security***



## 2.3.1 (*continued*)

- *Message Transfer Agent: SMTP*

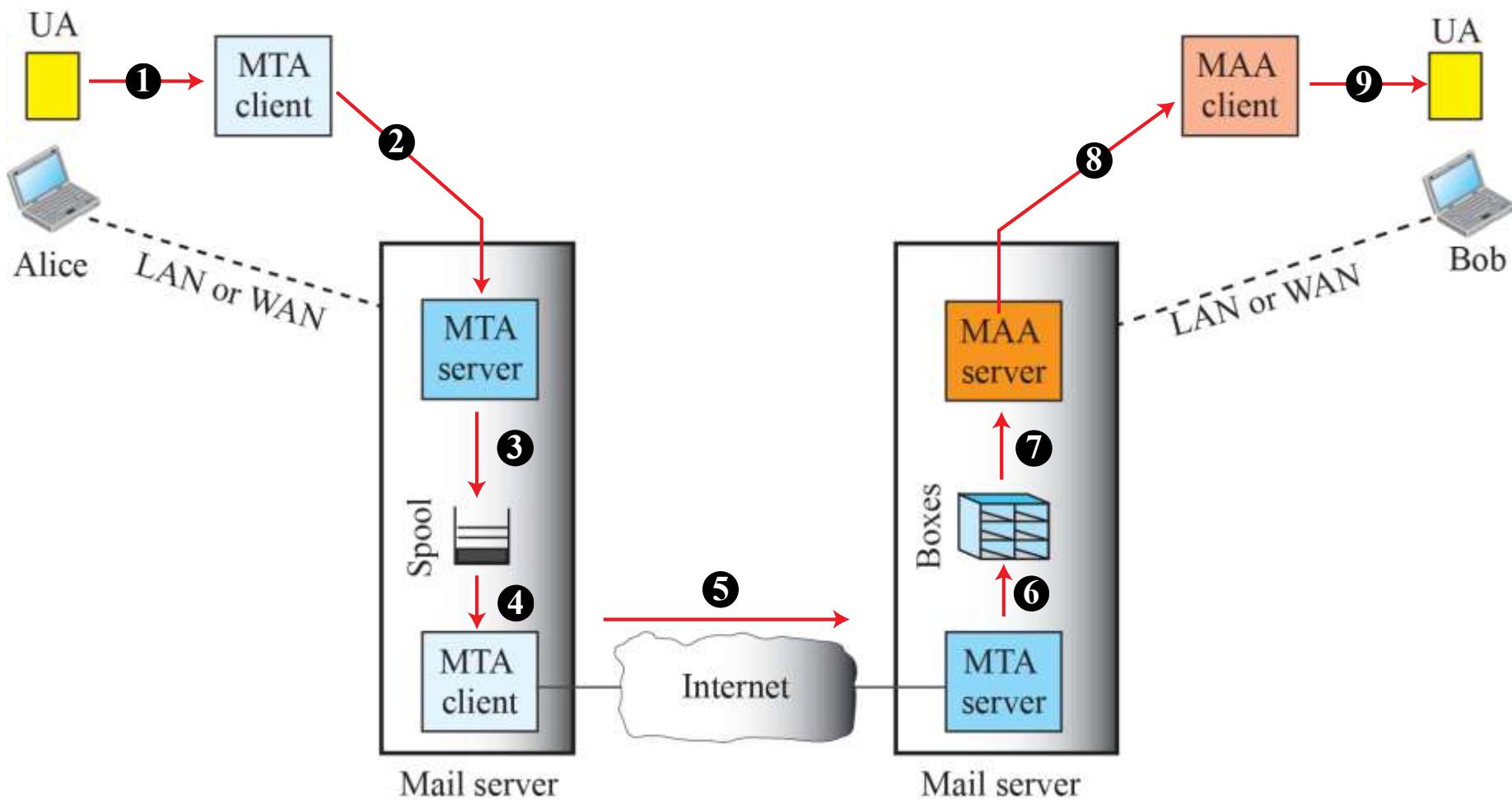
- ❖ *Commands and Responses*
- ❖ *Mail Transfer Phases*

- *Message Access Agent: POP and IMAP*

- ❖ *POP3*
- ❖ *IMAP4*

**Figure 2.19: Common scenario**

UA: user agent  
MTA: message transfer agent  
MAA: message access agent



## **Figure 2.20: Format of an e-mail**



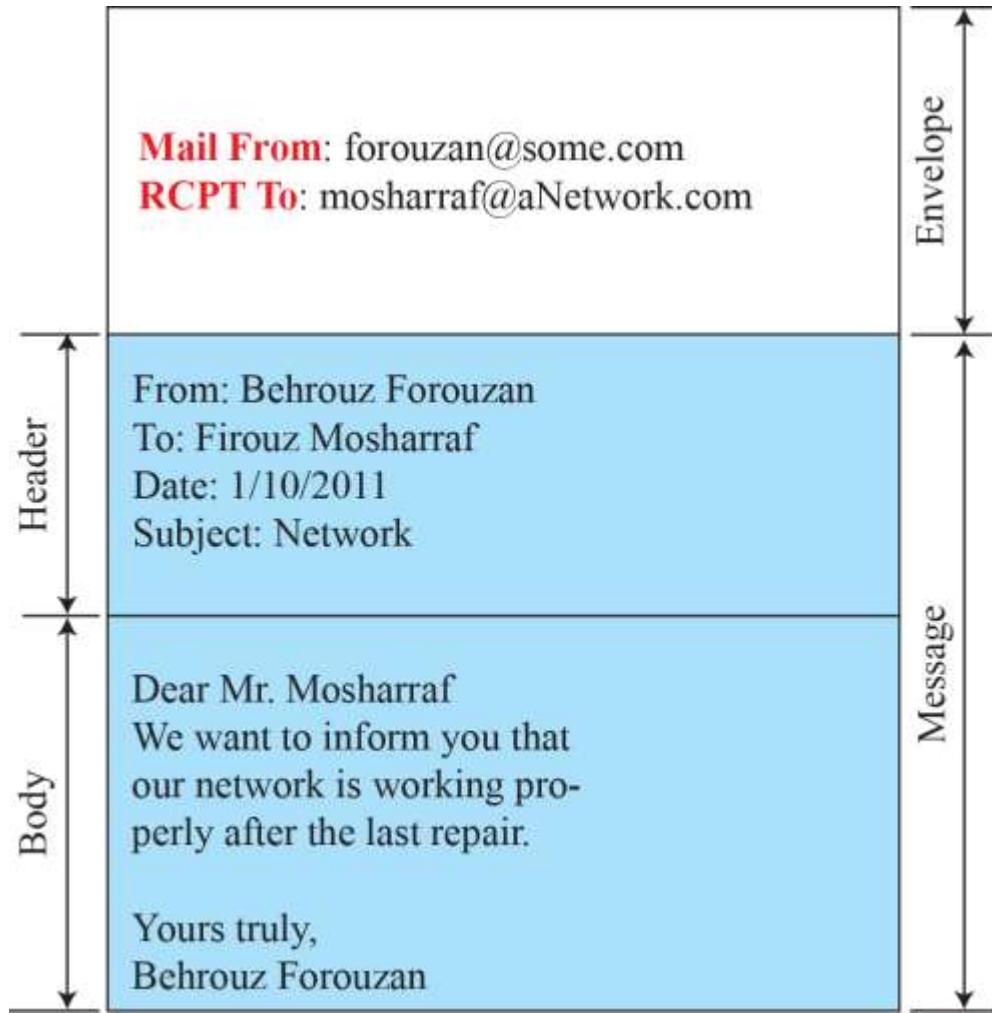
Behrouz Forouzan  
20122 Olive Street  
Bellbury, CA 91000  
Jan. 10, 2011

Subject: Network

Dear Mr. Mosharraf  
We want to inform you that  
our network is working pro-  
perly after the last repair.

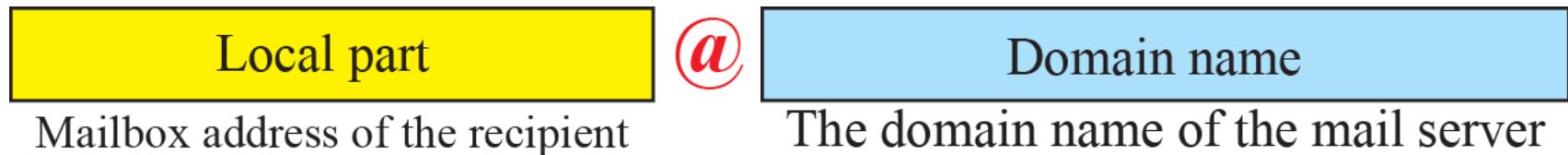
Yours truly,  
Behrouz Forouzan

**Postal mail**

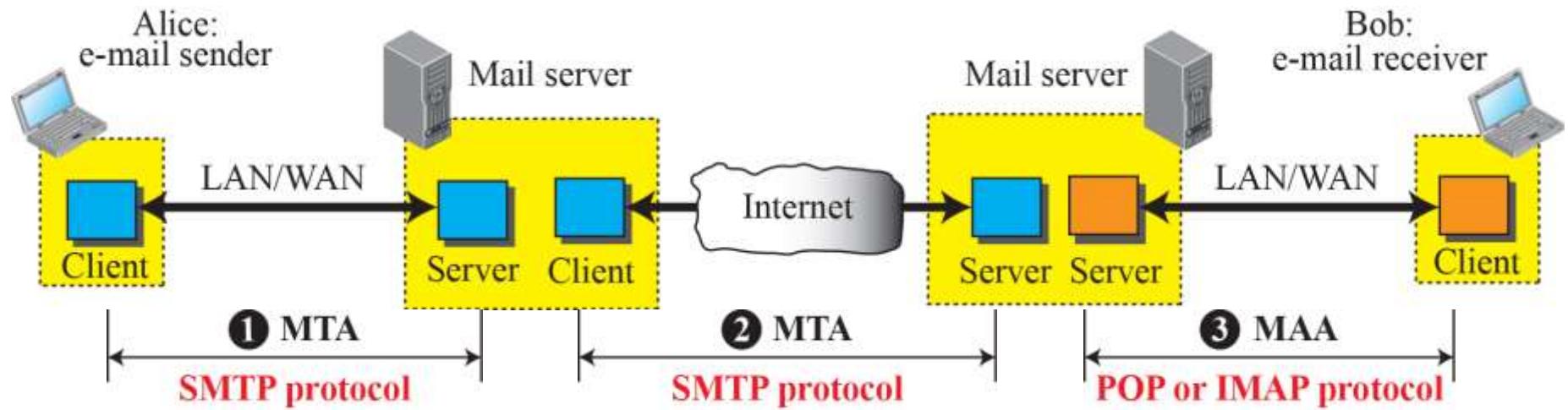


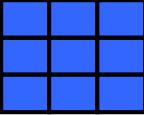
**Electronic mail**

## **Figure 2.21: E-mail address**



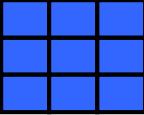
**Figure 2.22: Protocols used in electronic mail**





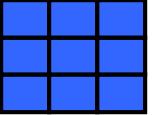
## Table 2.6: SMTP Commands

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 <i>or</i> the mailbox of the recipient
SMAL FROM	Intended recipient	Specifies that the mail be delivered to the terminal <i>and</i> the mailbox of the recipient



## Table 2.7: SMTP responses

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



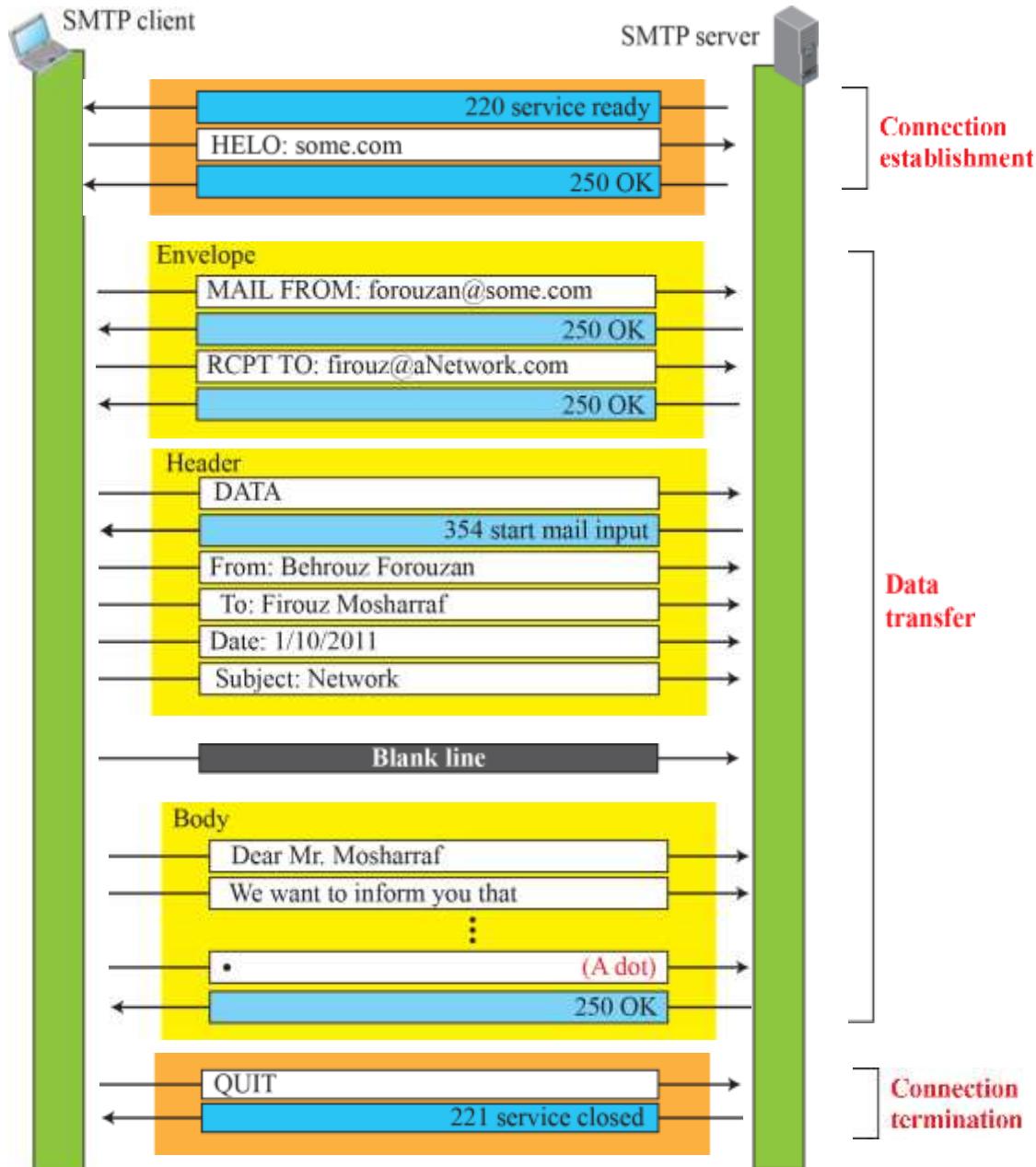
## Table 2.7: SMTP responses (continued)

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

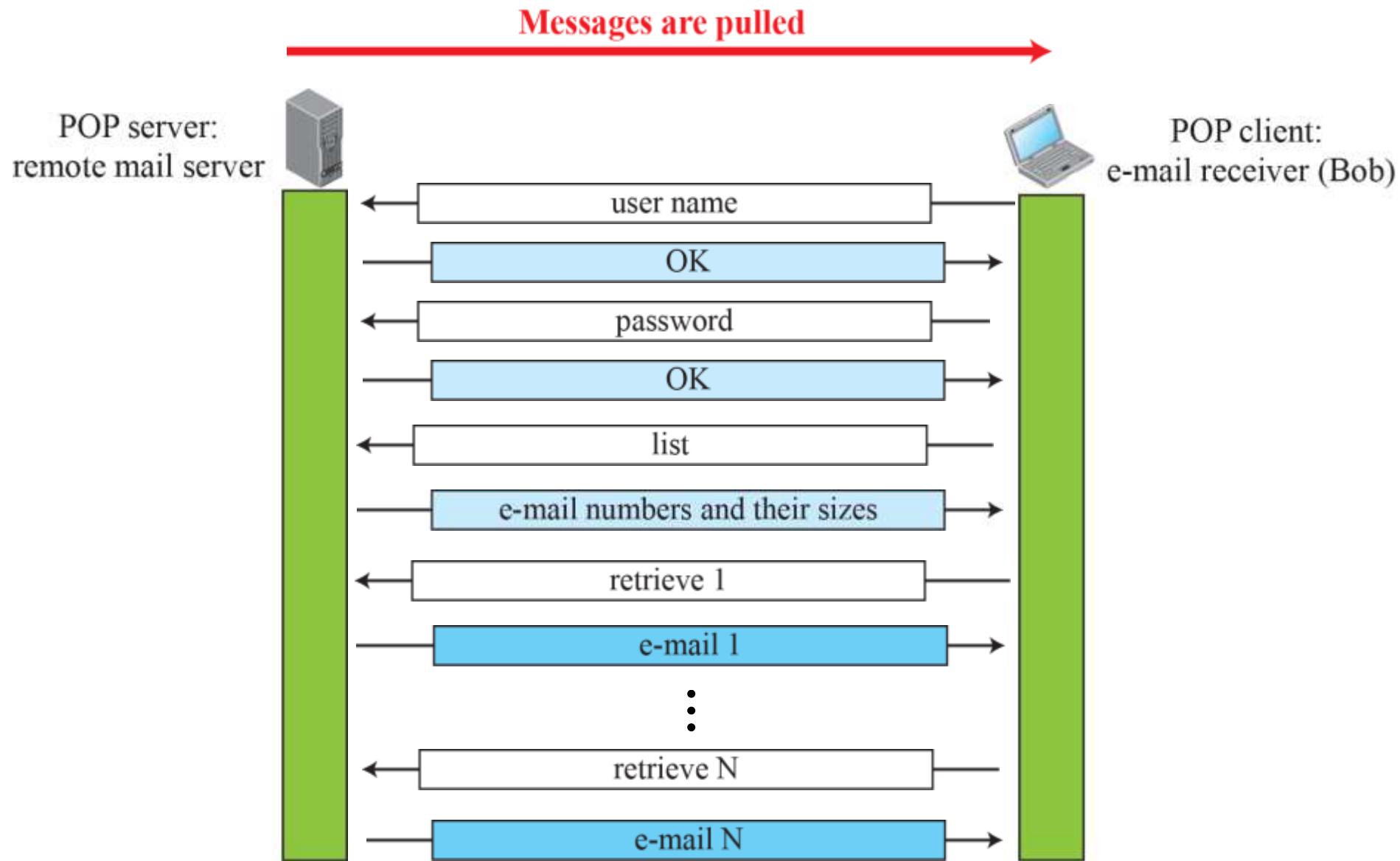
## **Example 2.13**

To show the three mail transfer phases, we show all of the steps described above using the information depicted in Figure 2.23. 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.

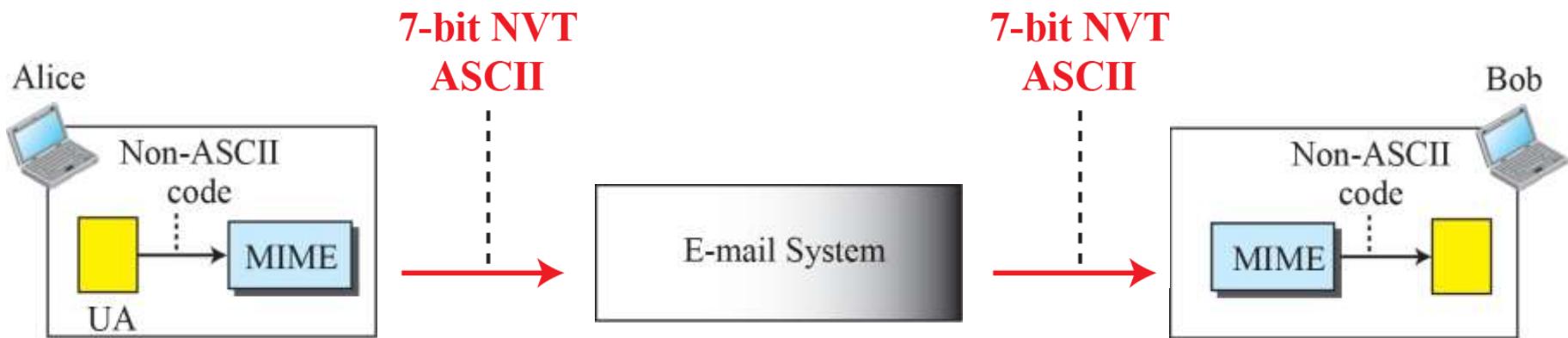
**Figure 2.23: Example 2.13**



**Figure 2.24: POP3**



**Figure 2.25: MIME**

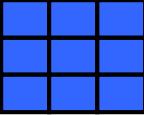


**Figure 2.26: MIME header**

**MIME headers**

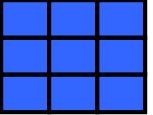
MIME-Version: 1.1  
Content-Type: type/subtype  
Content-Transfer-Encoding: encoding type  
Content-ID: message ID  
Content-Description: textual explanation of nontextual contents

**E-mail body**



## Table 2.8: Data Types and Subtypes in MIME

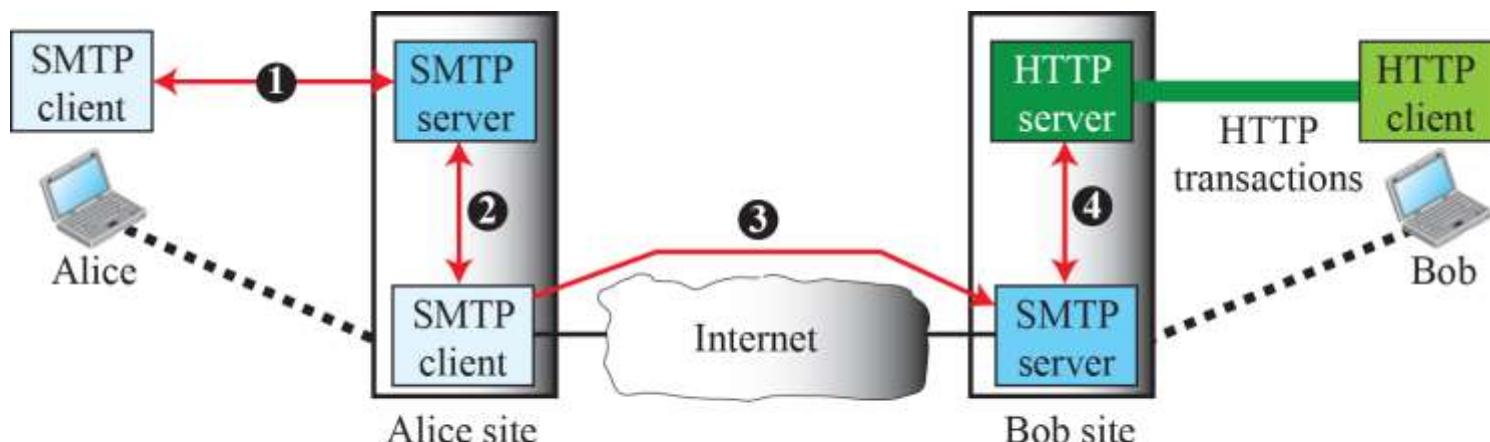
Type	Subtype	Description
Text	Plain	Unformatted
	HTML	HTML format (see Appendix C)
Multipart	Mixed	Body contains ordered parts of different data types
	Parallel	Same as above, but no order
	Digest	Similar to Mixed, but the default is message/RFC822
	Alternative	Parts are different versions of the same 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 (eight-bit bytes)



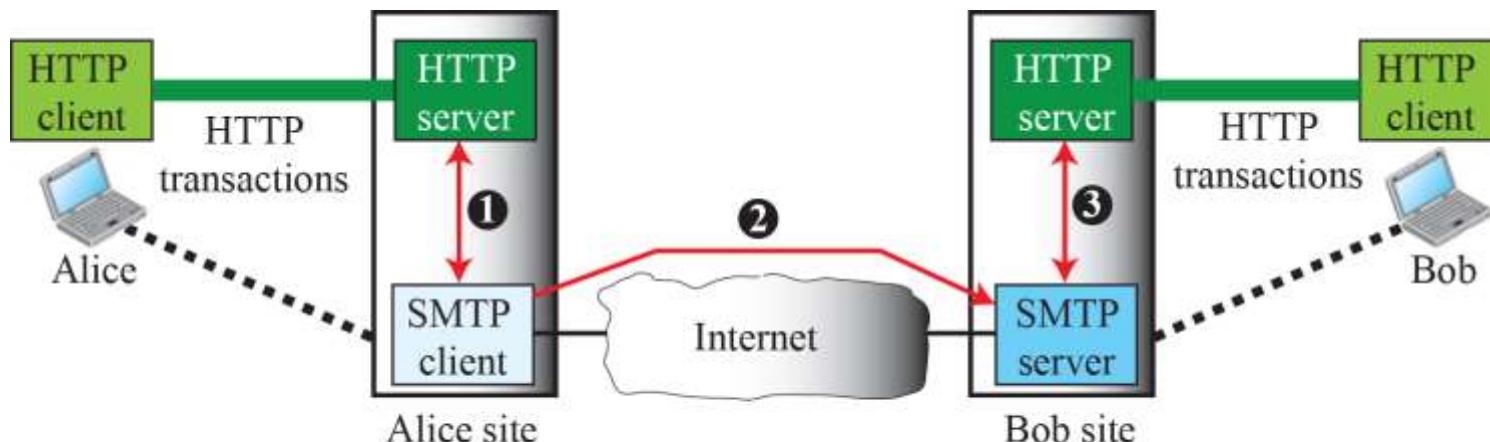
## Table 2.9: Methods for Content-Transfer-Encoding

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

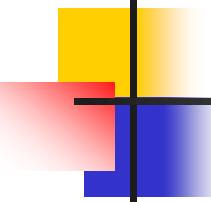
**Figure 2.29: Web-based e-mail, cases I and II**



Case 1: Only receiver uses HTTP

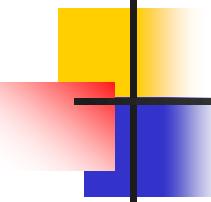


Case 2: Both sender and receiver use HTTP



## 2.3.4 TELNET

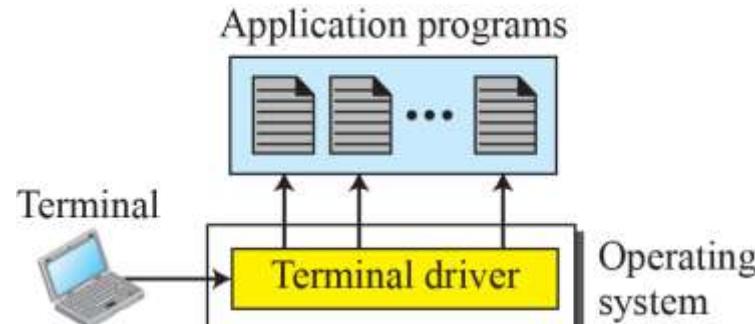
*A server program can provide a specific service to its corresponding client program. However, it is impossible to have a client/server pair for each type of service we need. Another solution is to have a specific client/server program for a set of common scenarios, but to have some generic client/server programs that allow a user on the client site to log into the computer at the server site and use the services available there. We refer to these generic client/server pairs as remote logging applications. One of the original remote logging protocols is TELNET.*



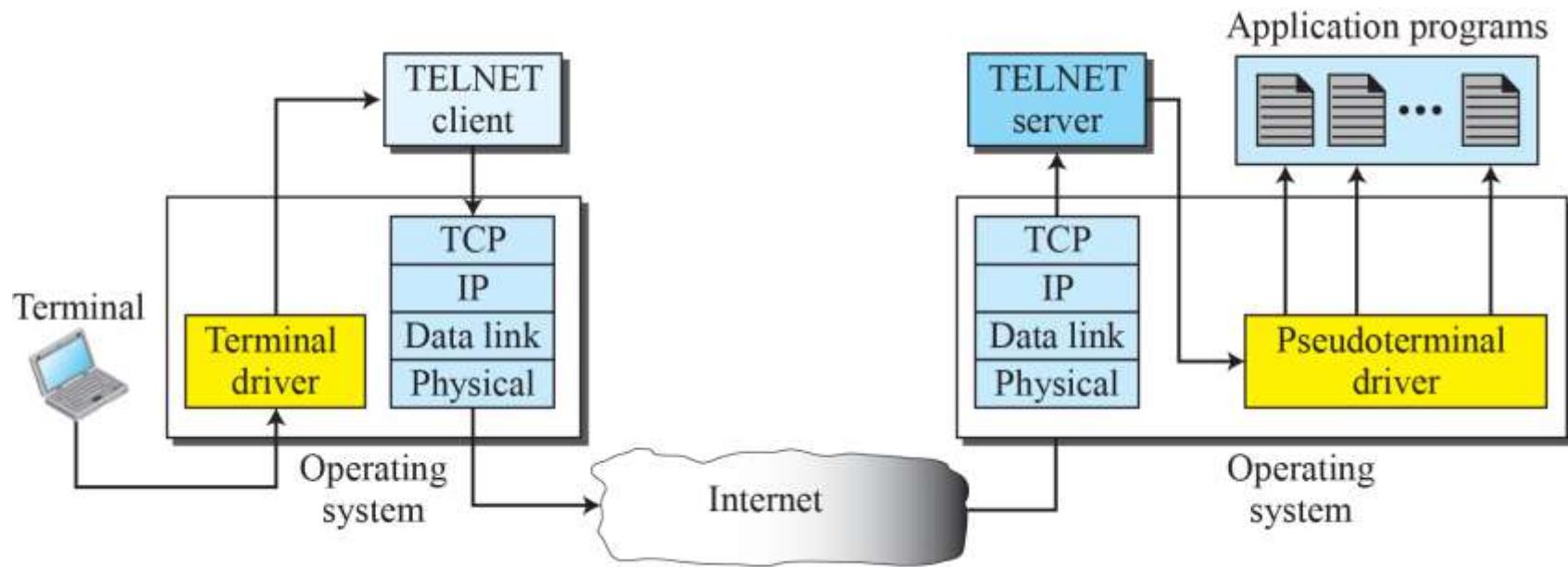
## **2.3.4 (continued)**

- Local versus Remote Logging*
- Network Virtual Terminal (NVT)*
- Options*
- User Interface*

**Figure 2.30: Local versus remote logging**

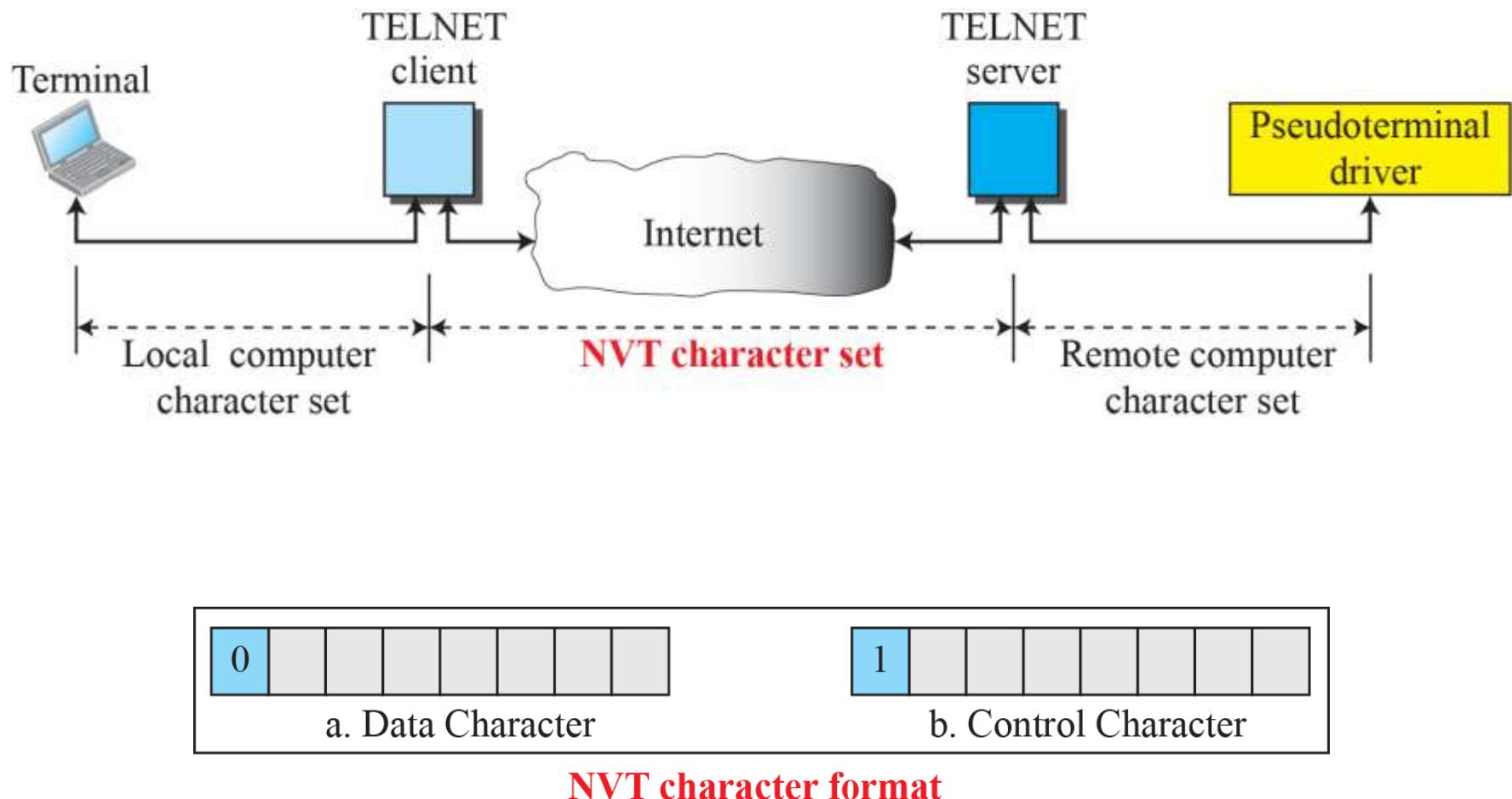


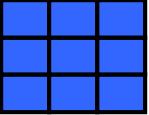
**a. Local logging**



**b. Remote logging**

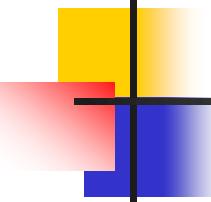
**Figure 2.31: Concept of NVT**





## Table 2.11: Examples of interface commands

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



## **2.3.5 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.*

## 2.3.5 (*continued*)

### ❑ *Components*

- ❖ *SSH Transport-Layer Protocol (SSH-TRANS)*
- ❖ *SSH Authentication Protocol (SSH-AUTH)*
- ❖ *SSH Connection Protocol (SSH-CONN)*

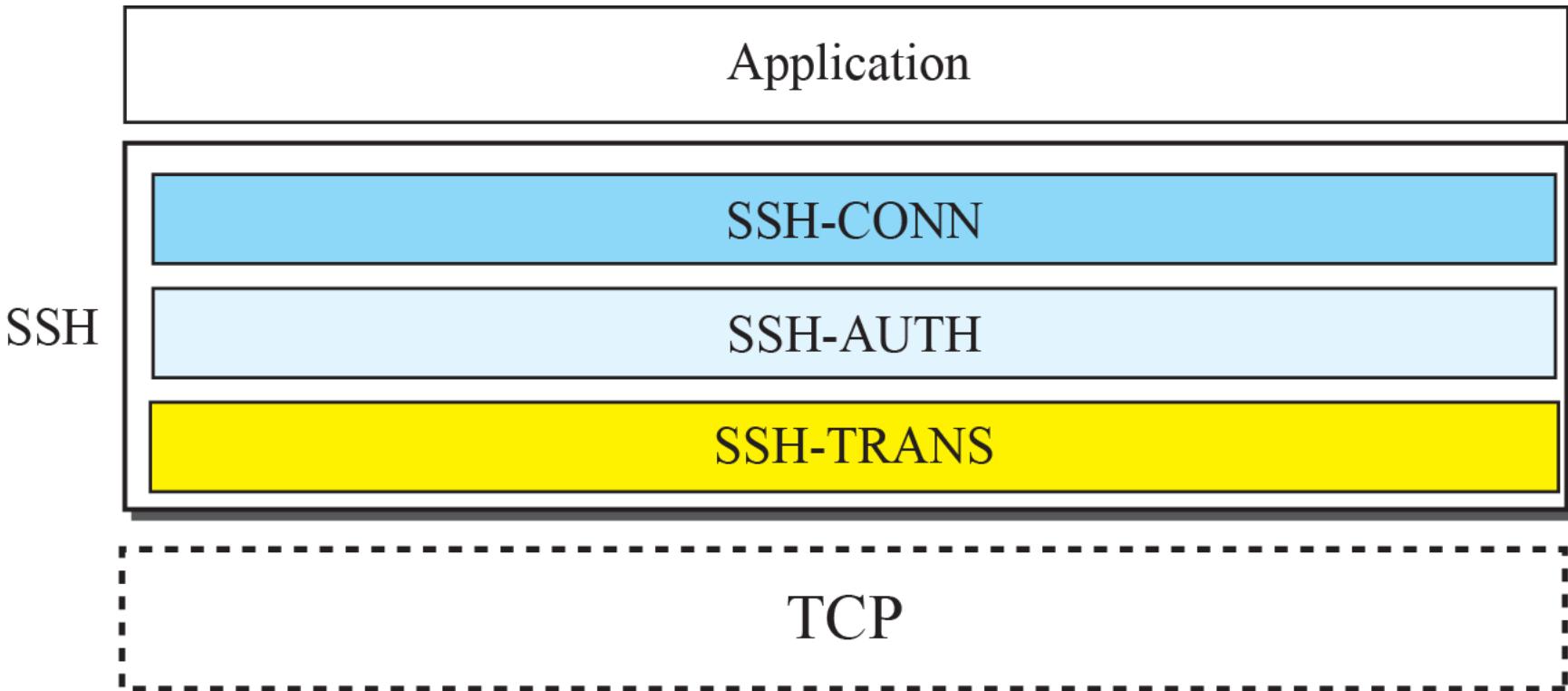
### ❑ *Applications*

- ❖ *SSH for Remote Logging*
- ❖ *SSH for File Transfer*

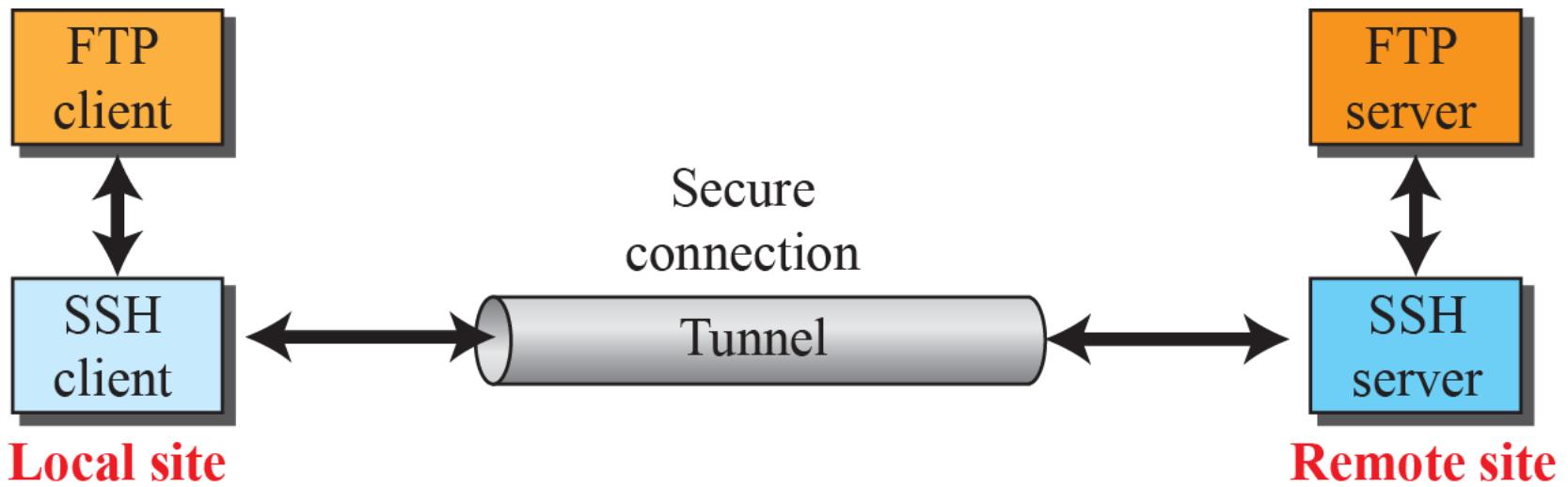
### ❑ *Port Forwarding*

### ❑ *Format of the SSH Packets*

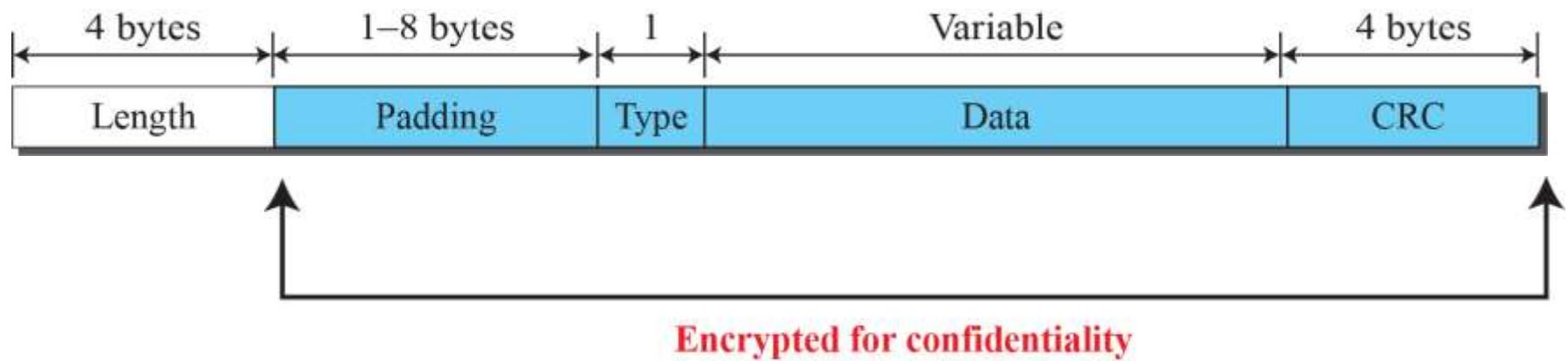
**Figure 2.32: Components of SSH**

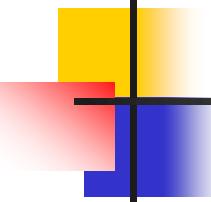


**Figure 2.33: Port Forwarding**



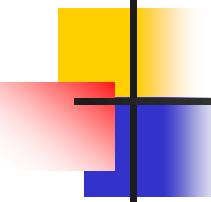
**Figure 2.34: SSH Packet Format**





## **2.3.6 Domain Name System (DNS)**

*To identify an entity, TCP/IP protocols use the IP address, which uniquely identifies the connection of a host to the Internet. However, people prefer to use names instead of numeric addresses. Therefore, the Internet needs to have a directory system that can map a name to an address. This is analogous to the telephone network. A telephone network is designed to use telephone numbers, not names. People can either keep a private file to map a name to the corresponding telephone number or can call the telephone directory to do so.*



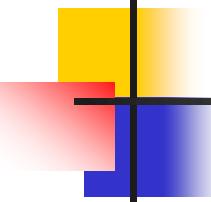
## 2.3.6 (*continued*)

### *Name Space*

- ❖ *Domain Name Space*
- ❖ *Domain*
- ❖ *Distribution of Name Space*
- ❖ *Zone*
- ❖ *Root Server*

### *DNS in the Internet*

- ❖ *Generic Domains*
- ❖ *Country Domains*



## **2.3.1 (continued)**

### ***Resolution***

- ❖ ***Recursive Resolution***
- ❖ ***Iterative Resolution***
- ❖ ***Caching***

### ***Resource Records***

### ***DNS Messages***

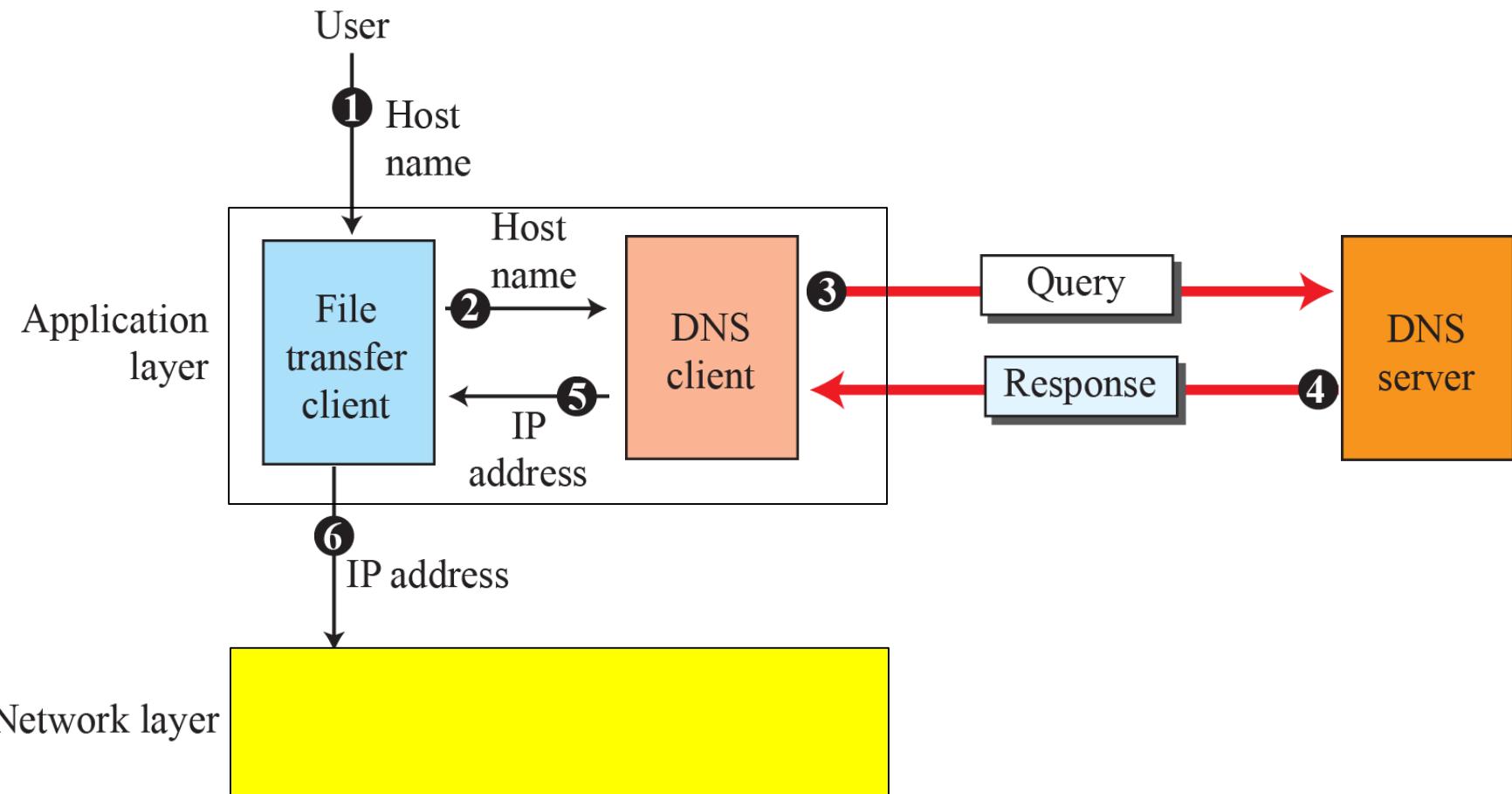
### ***Encapsulation***

### ***Registrars***

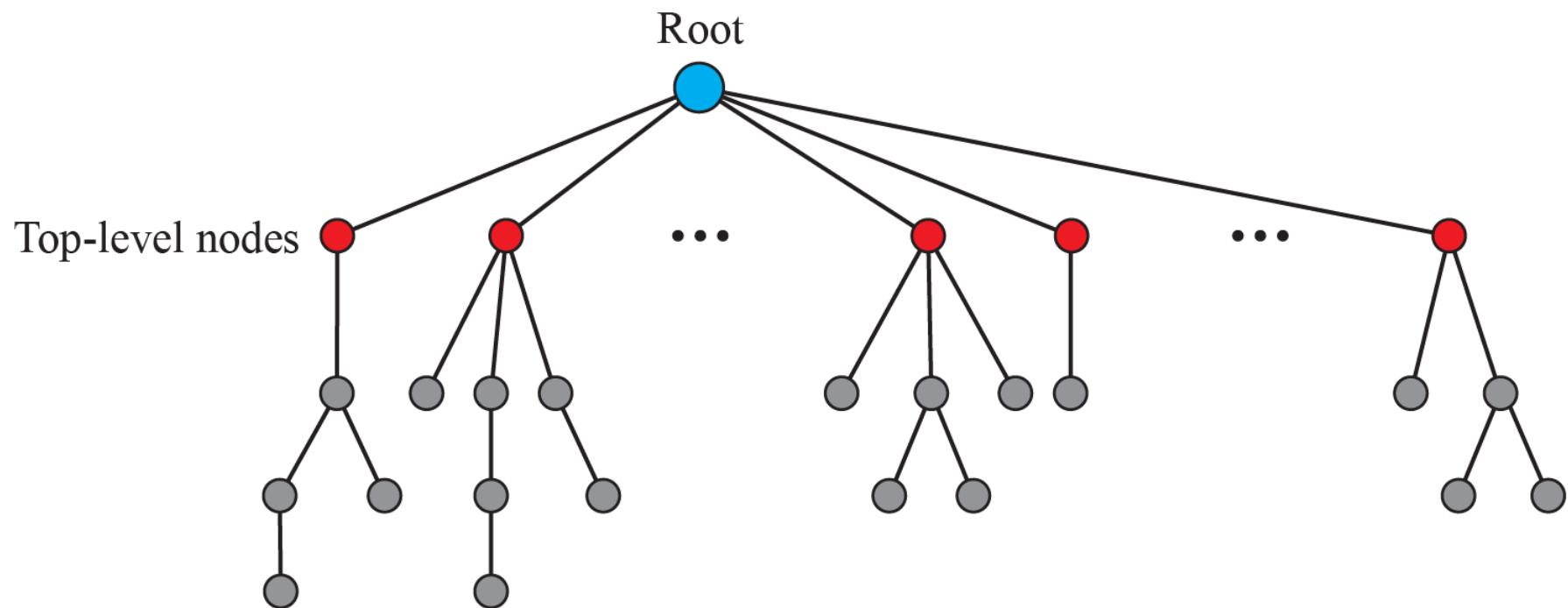
### ***DDNS***

### ***Security of DNS***

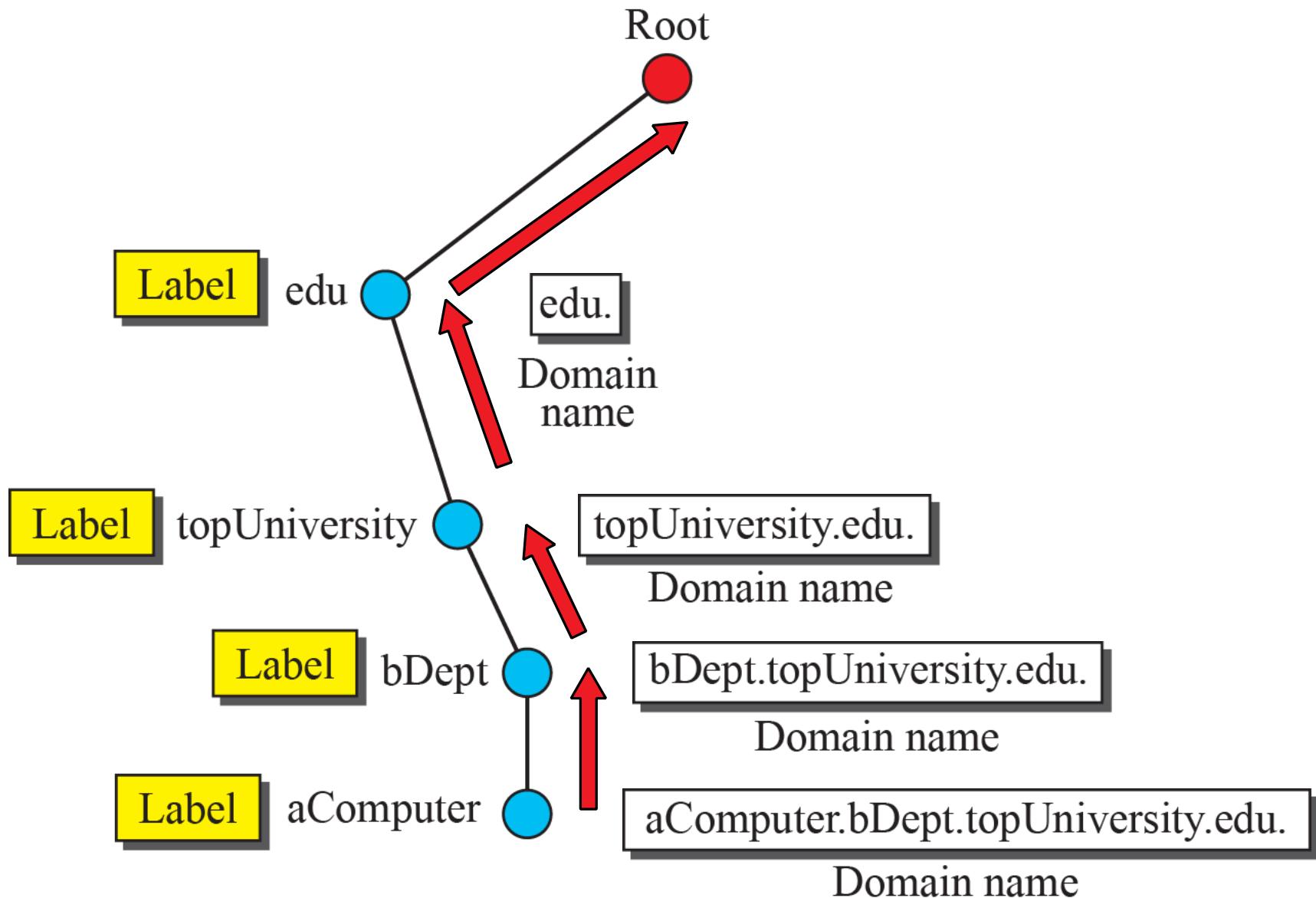
**Figure 2.35: Purpose of DNS**



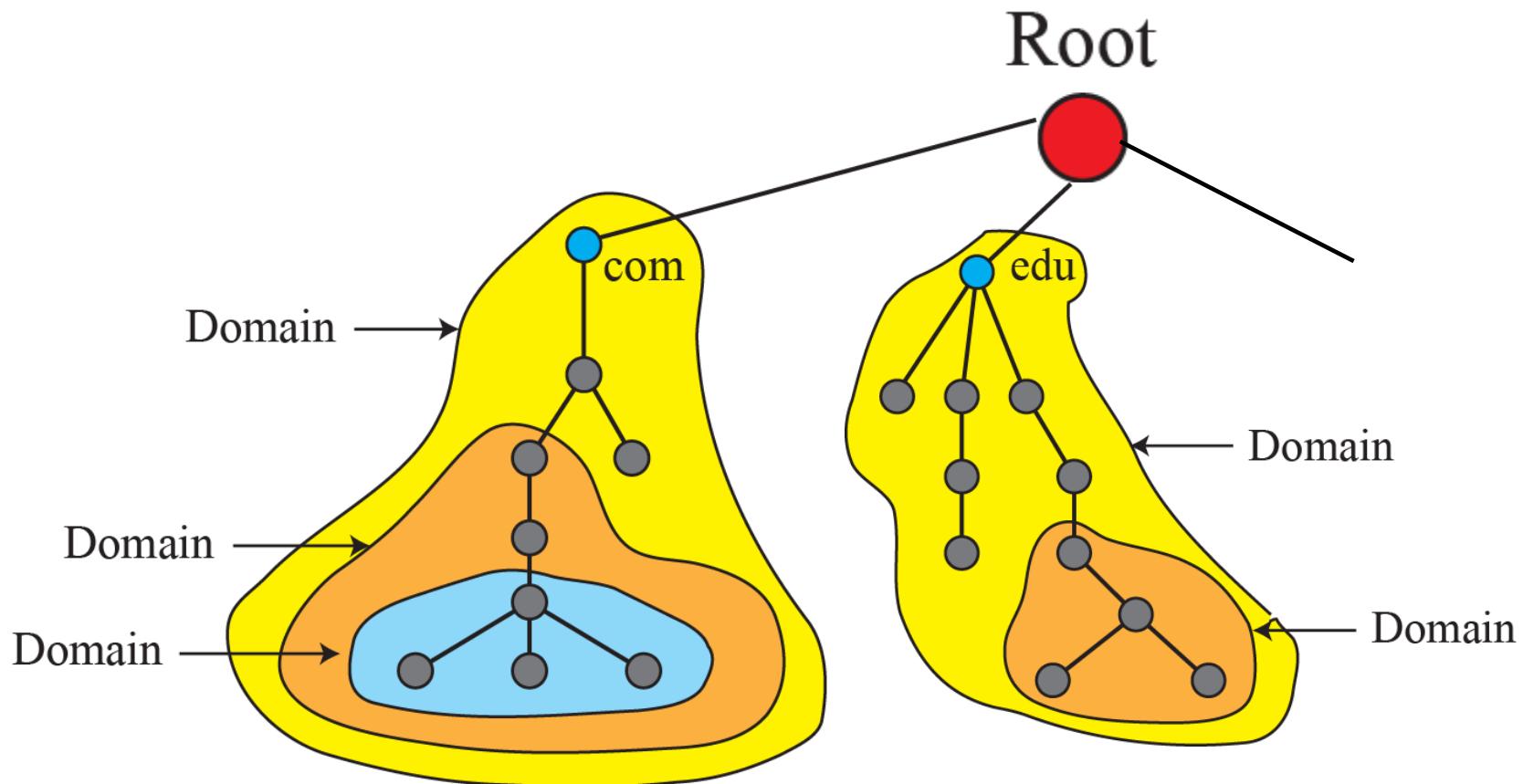
**Figure 2.36: Domain name space**



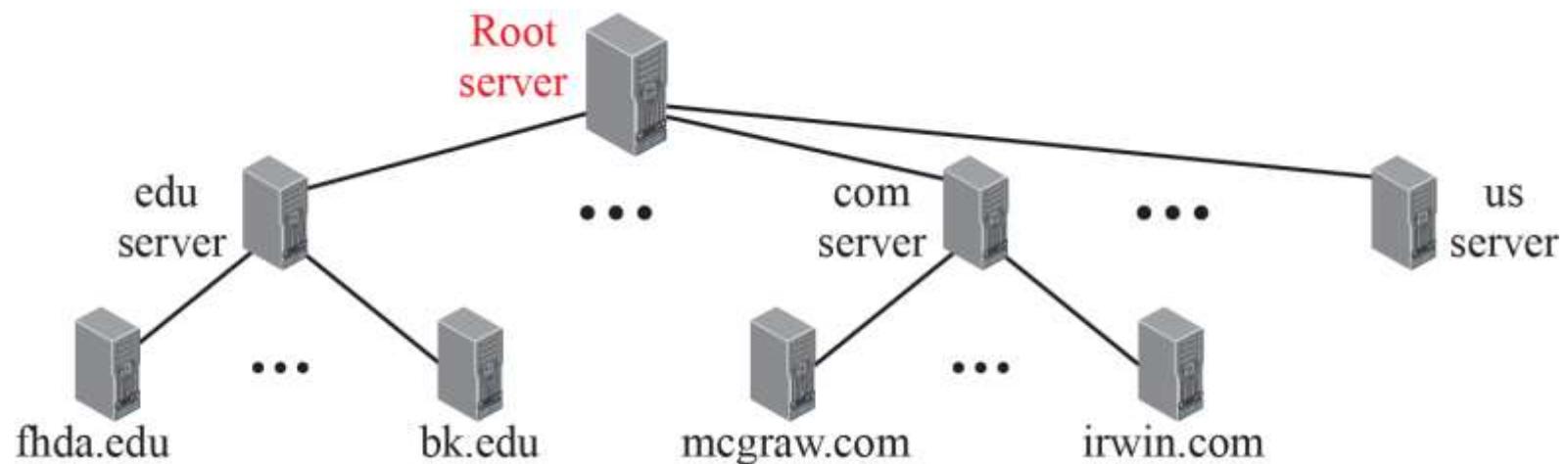
**Figure 2.37: Domain names and labels**



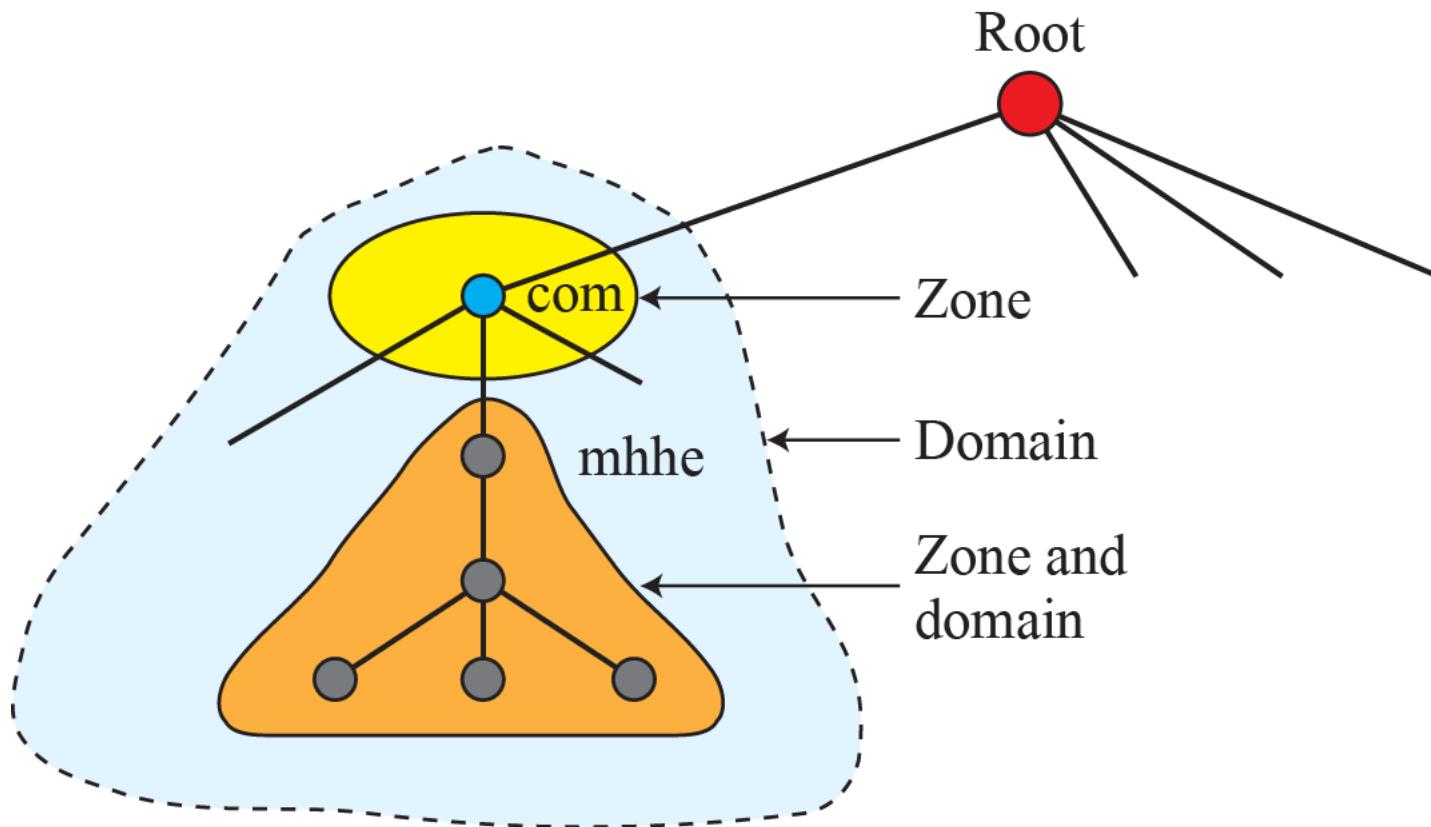
**Figure 2.38: Domains**



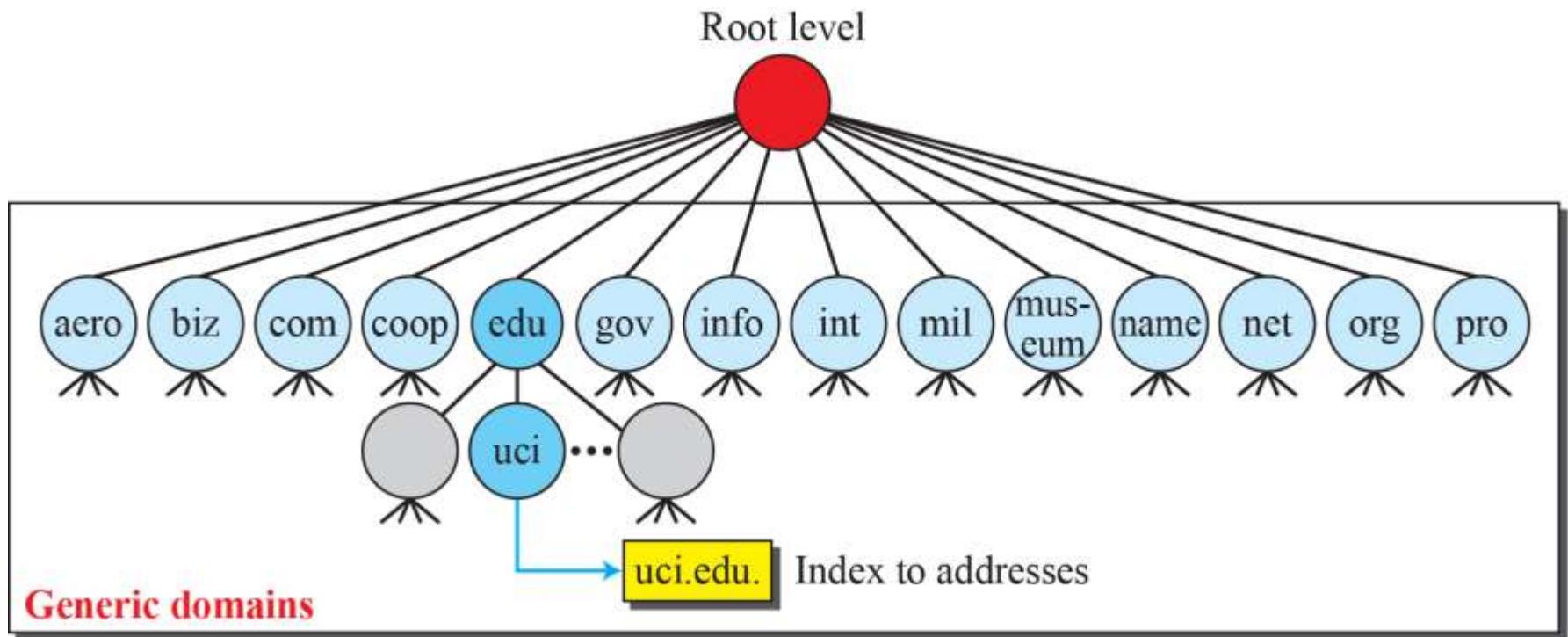
**Figure 2.39: Hierarchy of name servers**

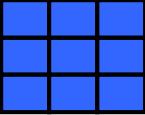


**Figure 2.40:** Zone



**Figure 2.41: Generic domains**

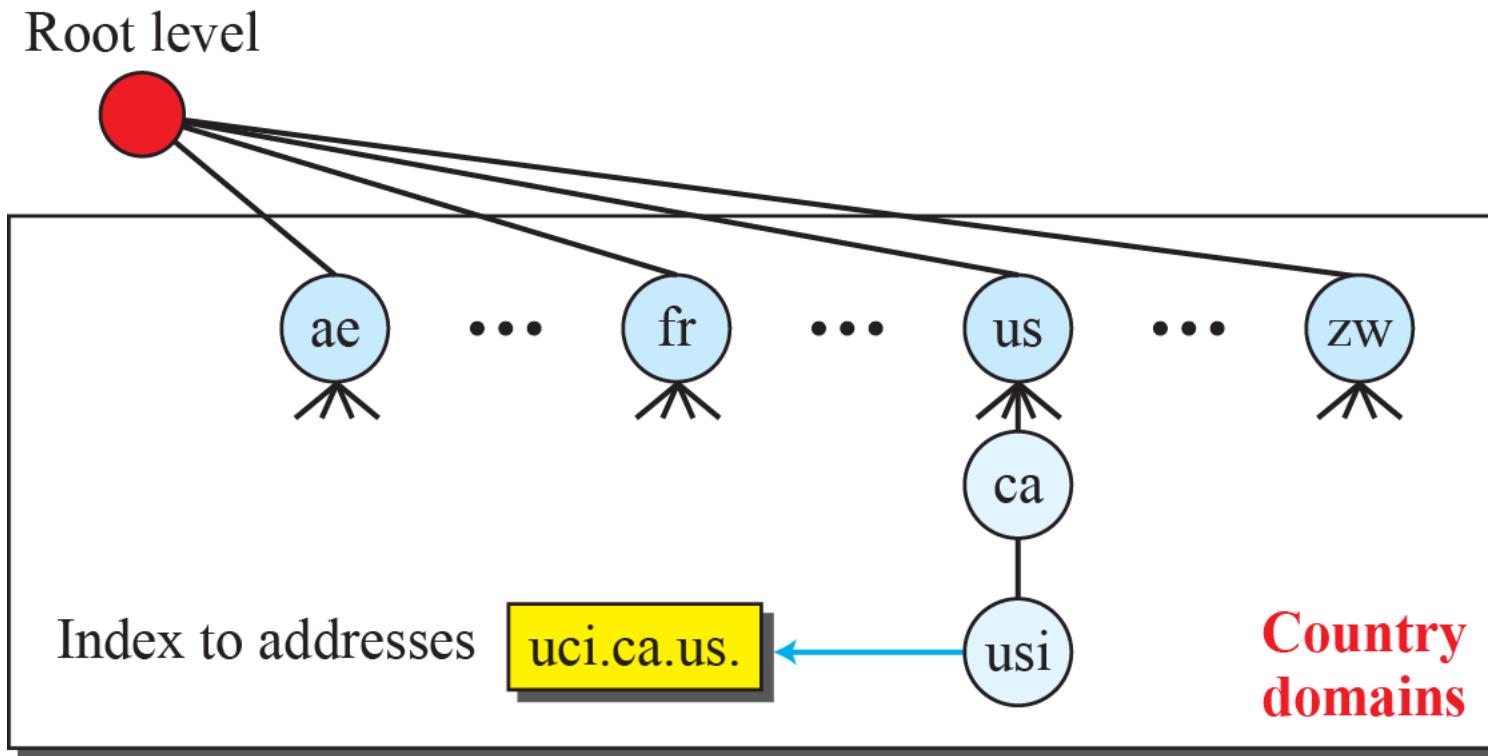




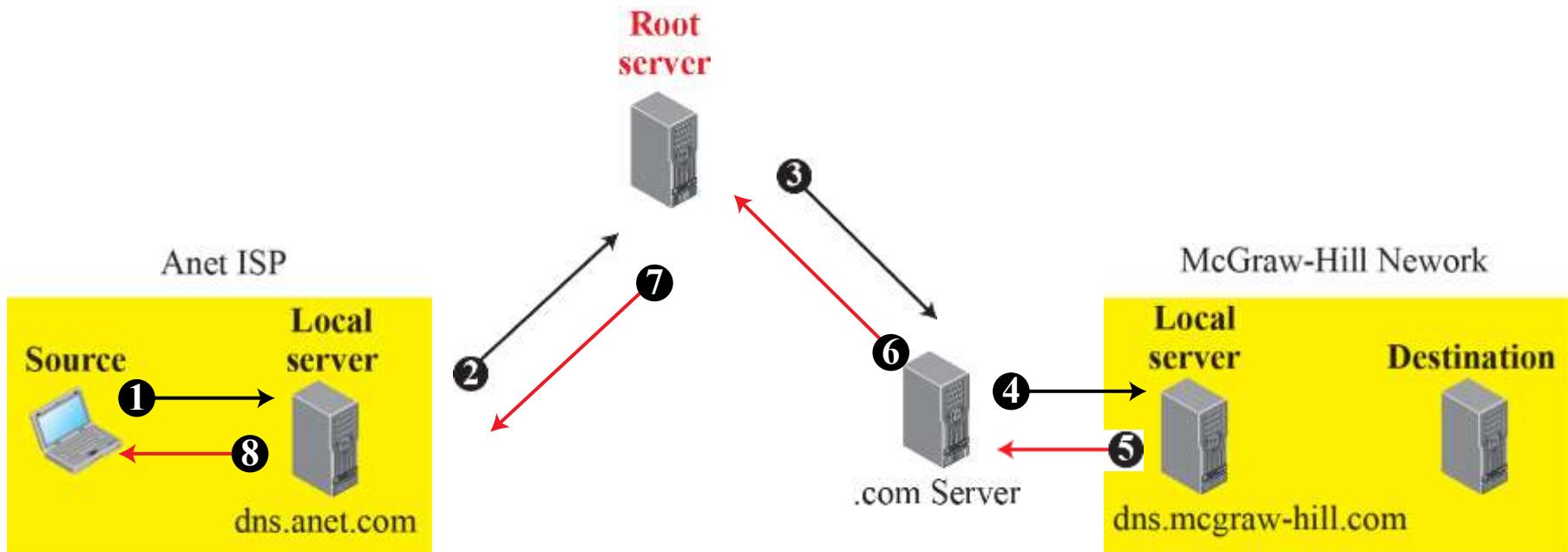
## Table 2.12: Generic domain labels

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

**Figure 2.42: Country domains**



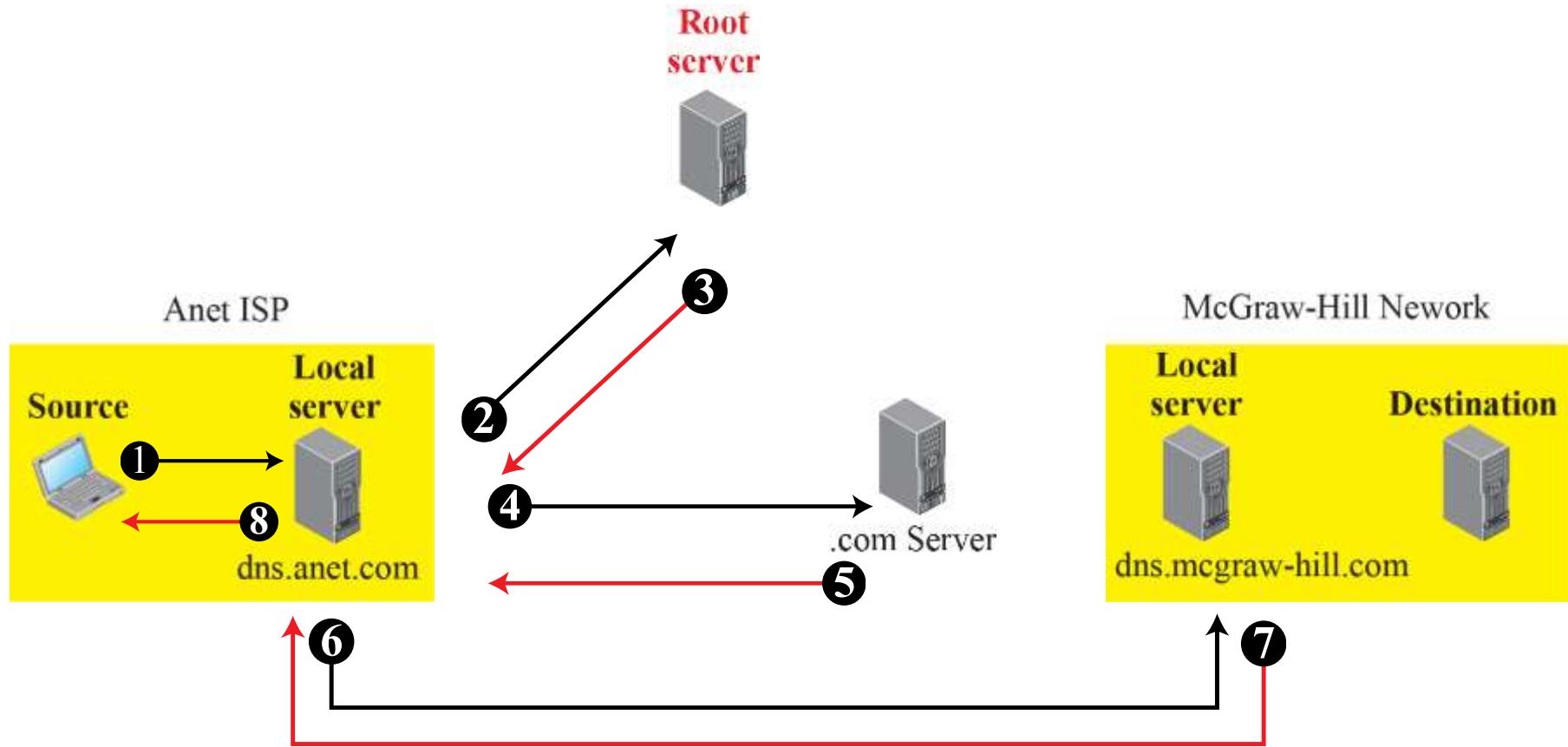
**Figure 2.43: Recursive resolution**



**Source:** some.anet.com

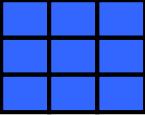
**Destination:** engineering.mcgraw-hill.com

**Figure 2.44: Iterative resolution**



**Source:** some.anet.com

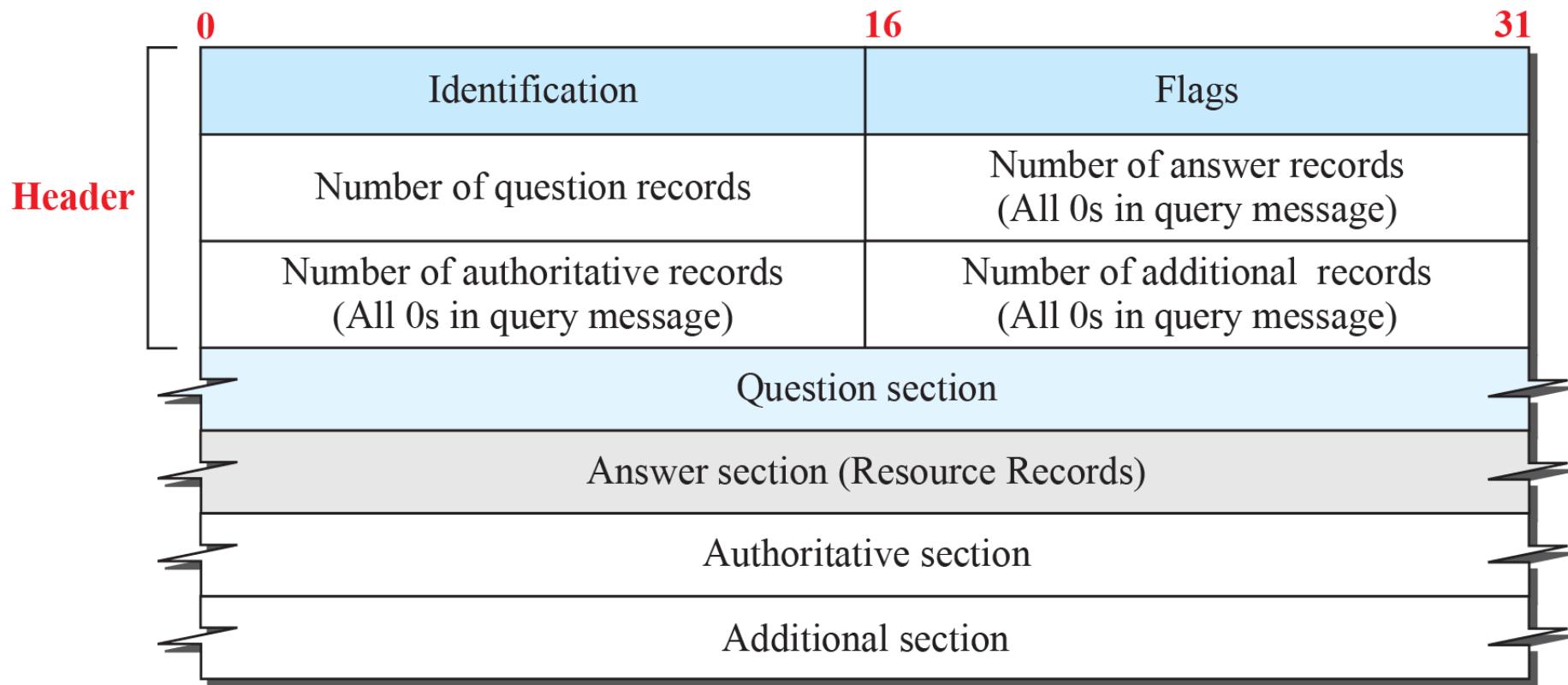
**Destination:** engineering.mcgraw-hill.com



## Table 2.13: DNS types

Type	<i>Interpretation of value</i>
A	A 32-bit IPv4 address (see Chapter 4)
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 4)

**Figure 2.45: 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.

## **Example 2.14**

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.

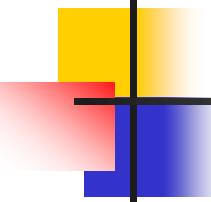
```
$nslookup www.forouzan.biz
```

```
Name: www.forouzan.biz
```

```
Address: 198.170.240.179
```

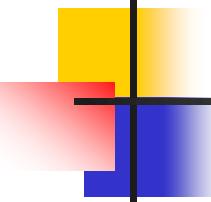
## 2-4 PEER-TO-PEER PARADIGM

*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.*



## 2.4.1 P2P Networks

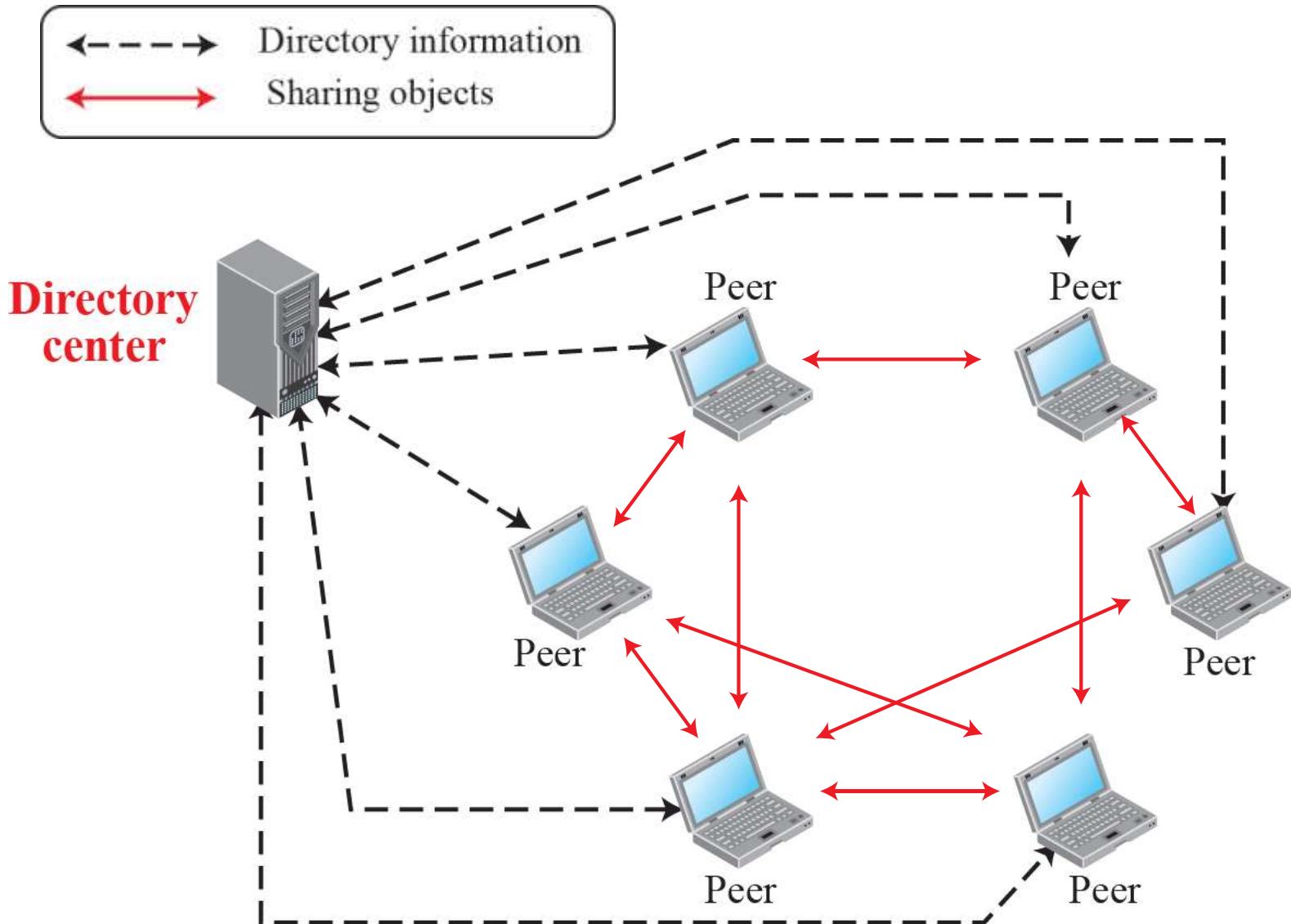
*Internet users that are ready to share their resources become peers and form a network. When a peer in the network has a file) to share, it makes it available to the rest of the peers. An interested peer can connect itself to the computer where the file is stored and download it. After a peer downloads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



## 2.4.1 (*continued*)

- *Centralized Networks*
  
- *Decentralized Network*
  - ❖ *Unstructured Networks*
  
  - ❖ *Structured Networks*

**Figure 2.46: Centralized network**



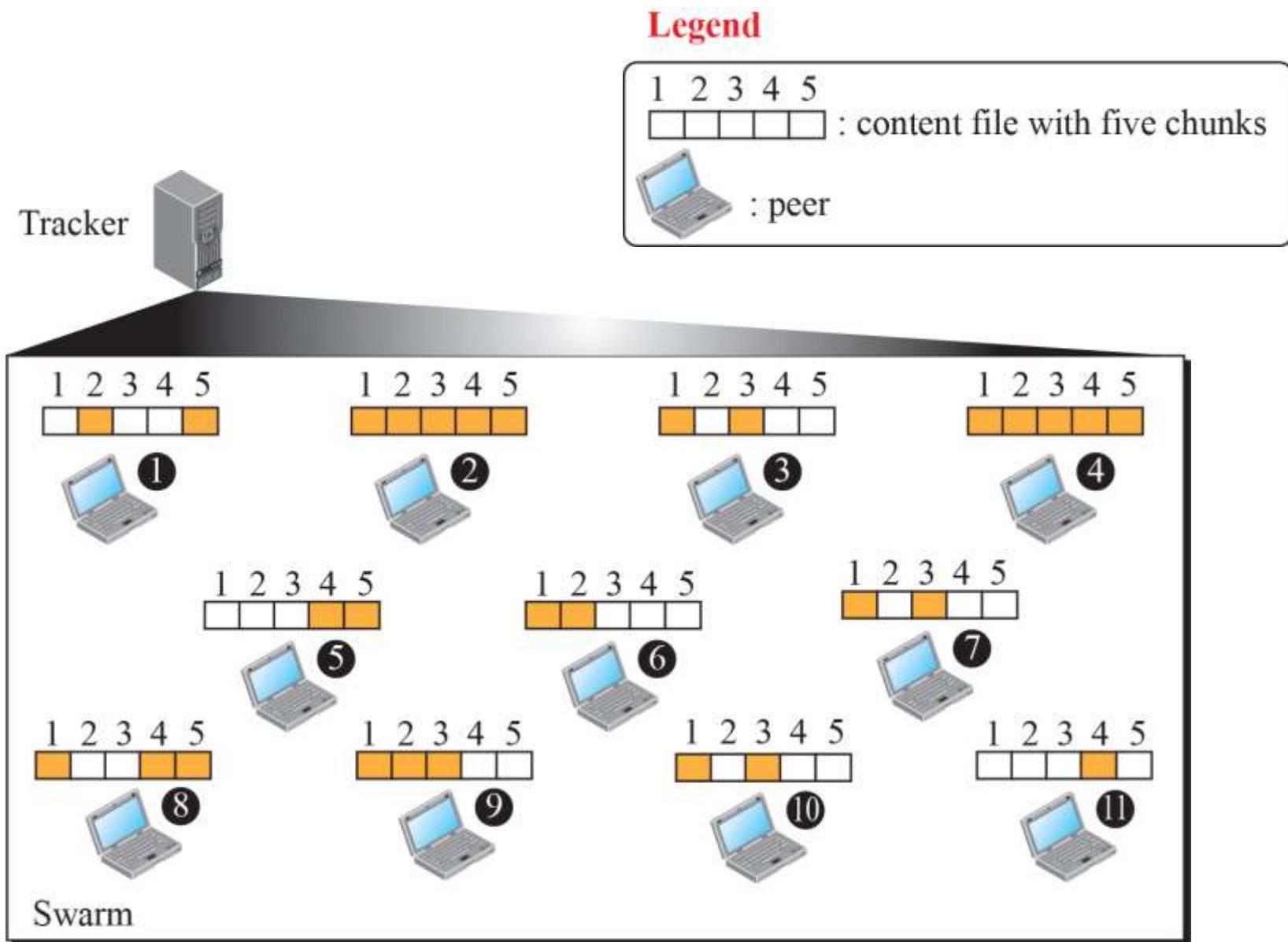
## **2.4.6 BitTorrent**

*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**

### **❑ Trackerless BitTorrent**

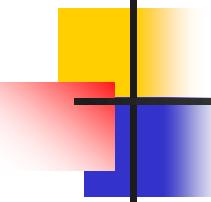
**Figure 2.57:** Example of a torrent



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

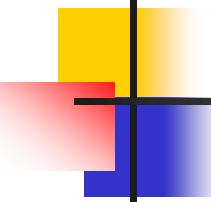
## 2-5 SOCKET-INTERFACE PROGRAMMING

*In this section, we show how to write some simple client-server programs using C, a procedural programming language. We chose the C language in this section; In Chapter 11, we expand this idea in Java, which provides a more compact version.*



## **2.5.1 *Socket Interface in C***

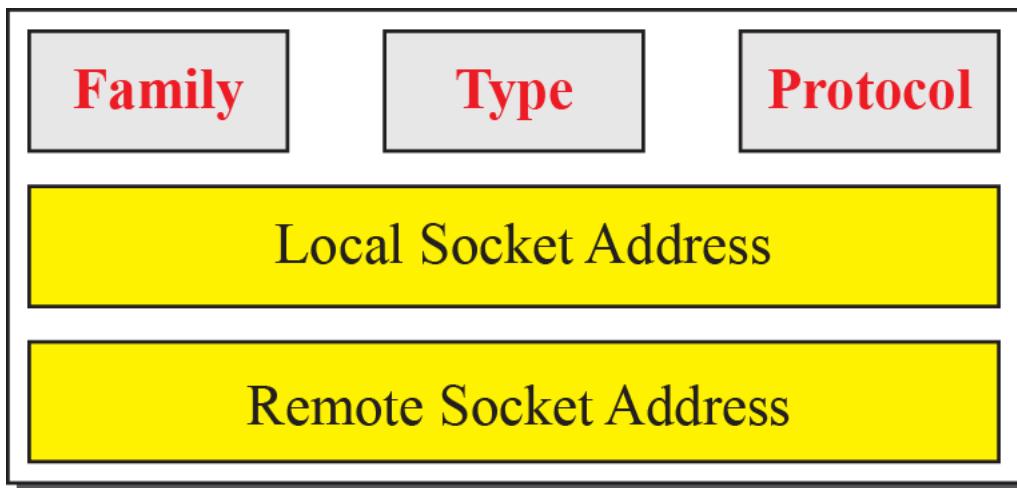
*In this section, we show how this interface is implemented in the C language. The important issue in socket interface is to understand the role of a socket in communication. The socket has no buffer to store data to be sent or received. It is capable of neither sending nor receiving data. The socket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



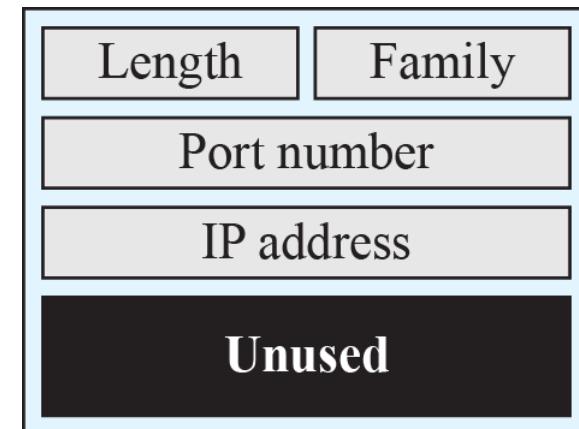
## **2.5.1 (continued)**

- Data Structure for Socket*
- Header Files*
- Communication Using UDP*
  - ❖ *Sockets Used for UDP*
  - ❖ *Communication Flow Diagram*
  - ❖ *Programming Examples*
- Communication Using TCP*
  - ❖ *Sockets Used in TCP*
  - ❖ *Communication Flow Diagram*

**Figure 2.58: Socket data structure**

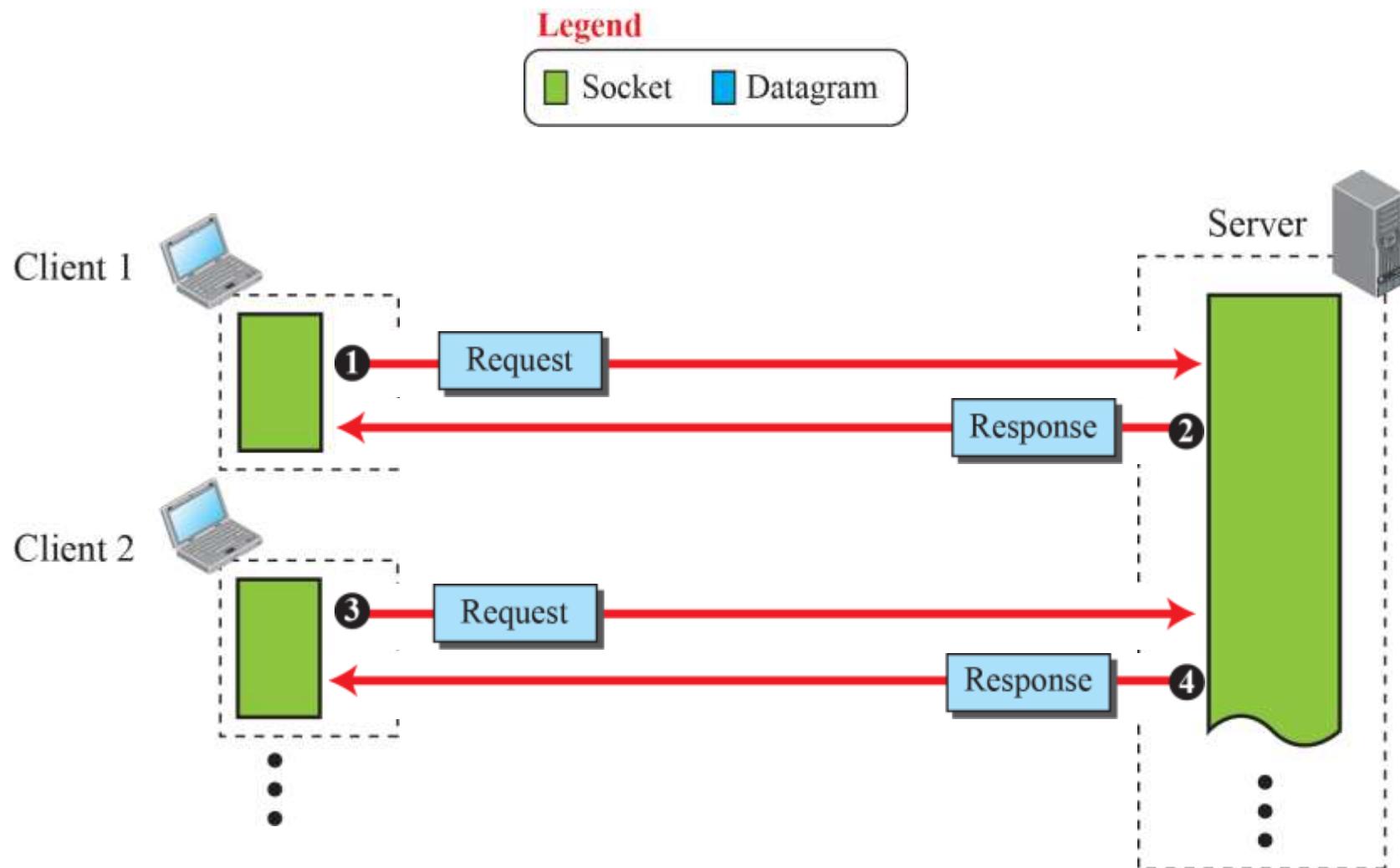


**Socket**

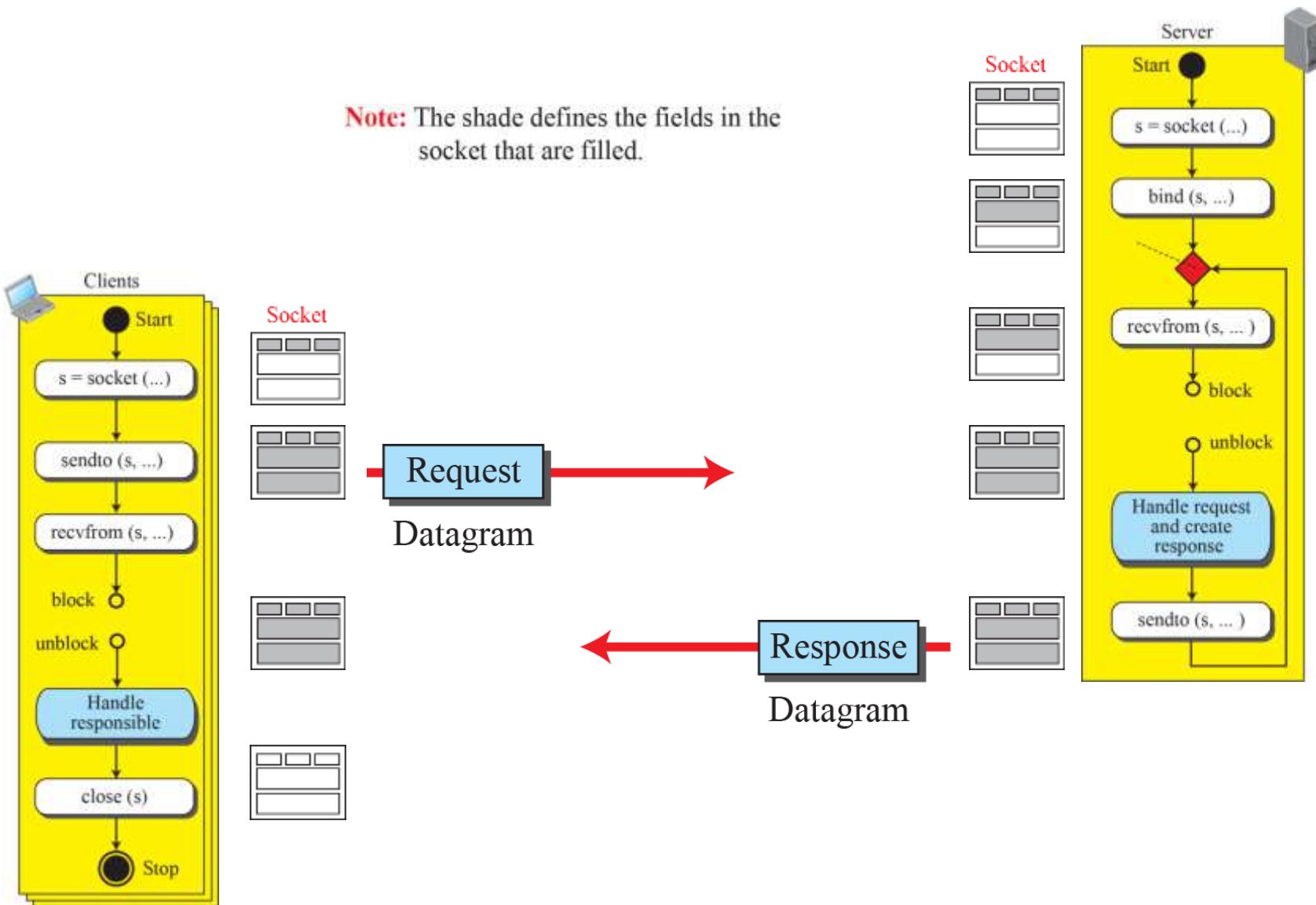


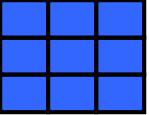
**Socket address**

**Figure 2.59: Sockets for UDP communication**



**Figure 2.60: Flow diagram for iterative UDP communication**





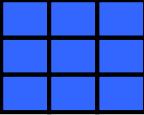
## Table 2.22: Echo server program using UDP

```
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);
```



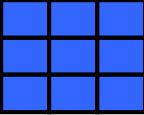
## Table 2.22: Echo server program using UDP (continued)

```
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*)&cIntAddr, &cIntAddrLen);
34      // Send String
35      sendto (s, buffer, len, 0, (struct sockaddr*)&cIntAddr, sizeof(cIntAddr));
36  } // End of for loop
37 } // End of echo server program
```



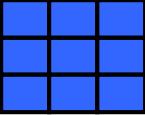
## Table 2.23: Echo client program using UDP

```
1 // UDP echo client program
2 #include "headerFiles.h"
3 int main (int argc, char* argv[ ])
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
14    // Check and set program arguments
15    if (argc != 3)
16    {
17        printf ("Error: three arguments are needed!");
18        exit(1);
19    }
```



## Table 2.23: Echo client program using UDP (continued)

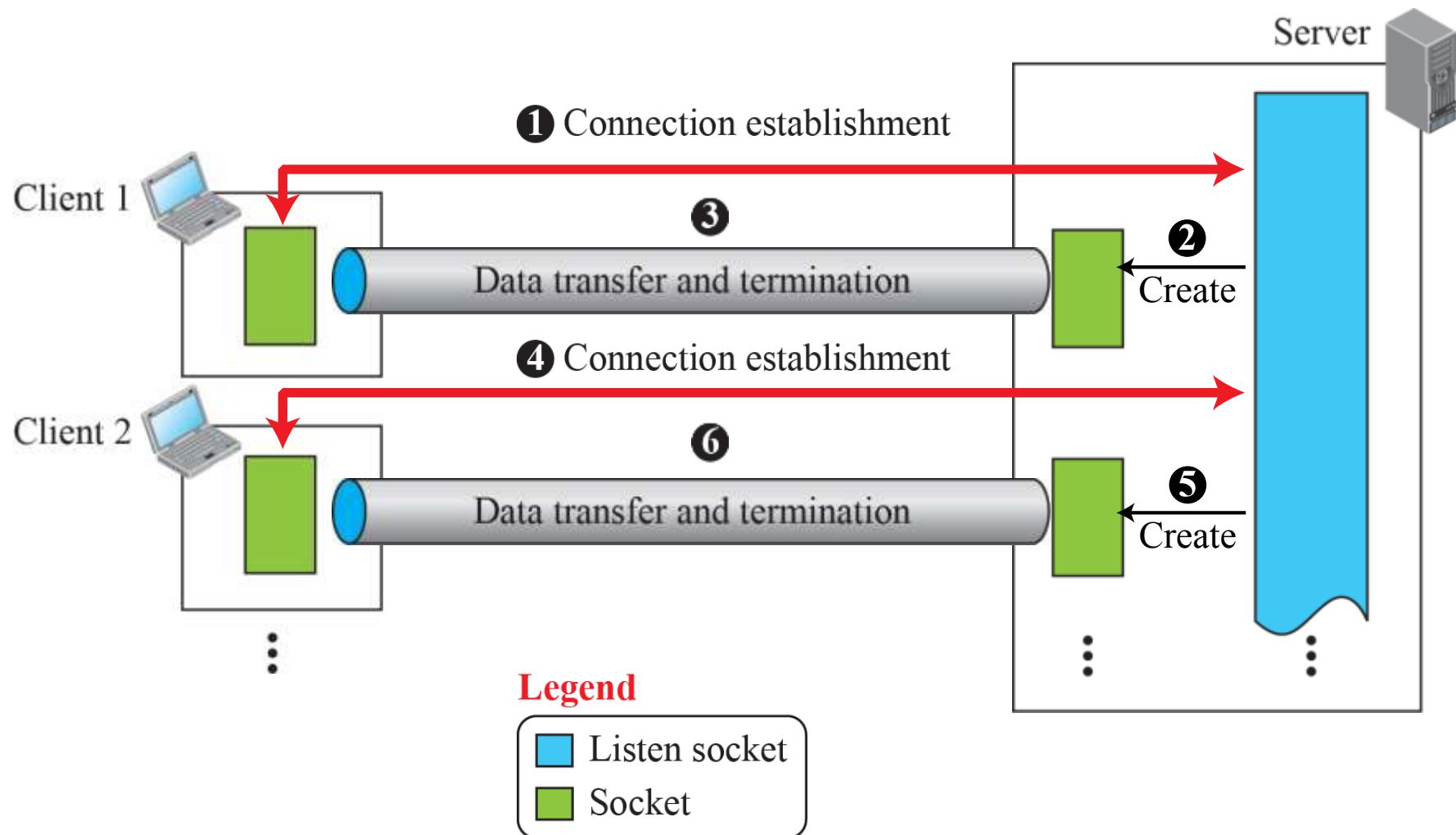
```
19    servName = argv[1];
20    servPort = atoi (argv[2]);
21    string = argv[3];
22    // Build server socket address
23    memset (&servAddr, 0, sizeof (servAddr));
24    servAddr.sin_family = AF_INET;
25    inet_pton (AF_INET, servName, &servAddr.sin_addr);
26    servAddr.sin_port = htons (servPort);
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
```



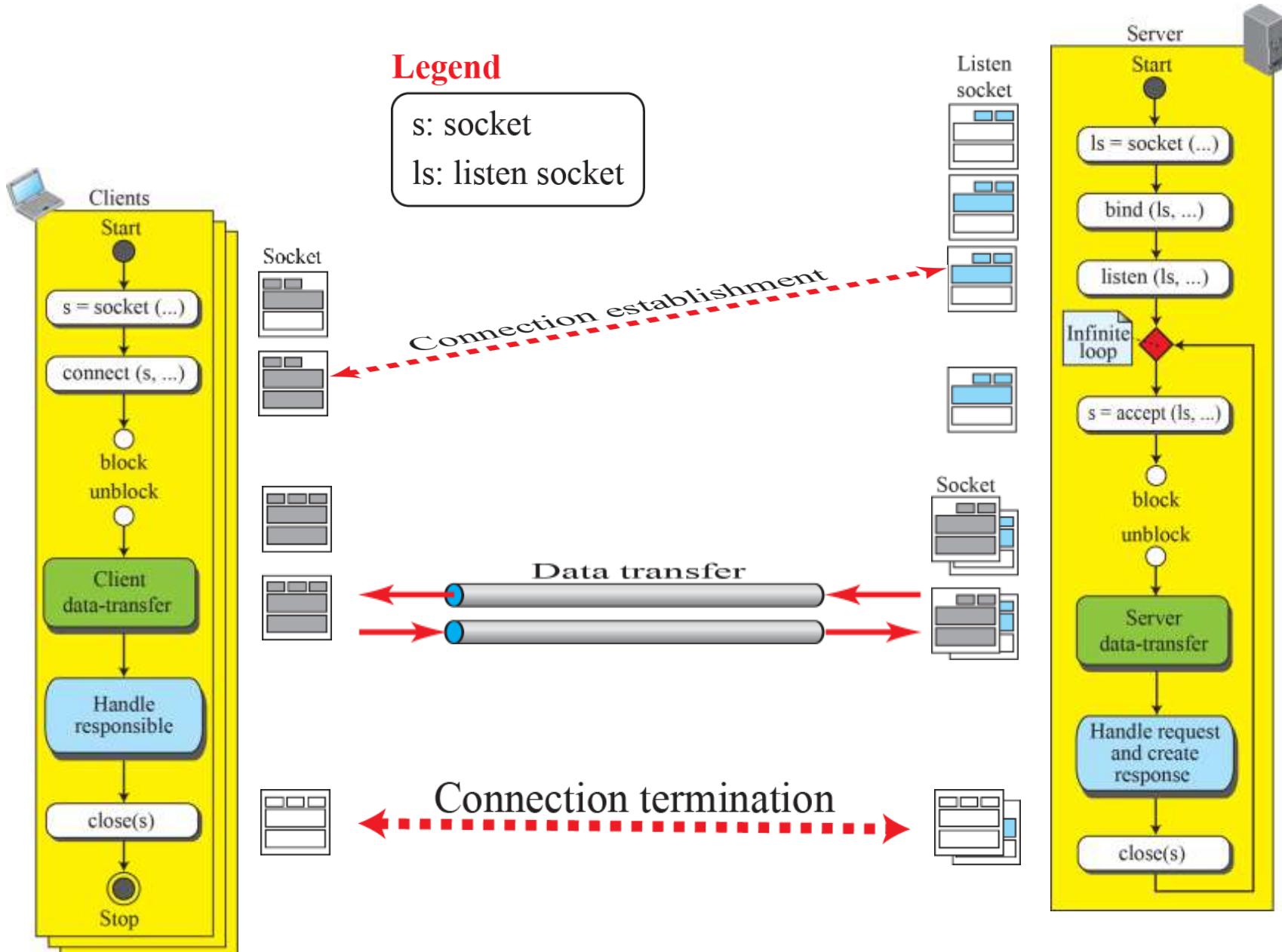
## Table 2.23: Echo client program using UDP (continued)

```
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
```

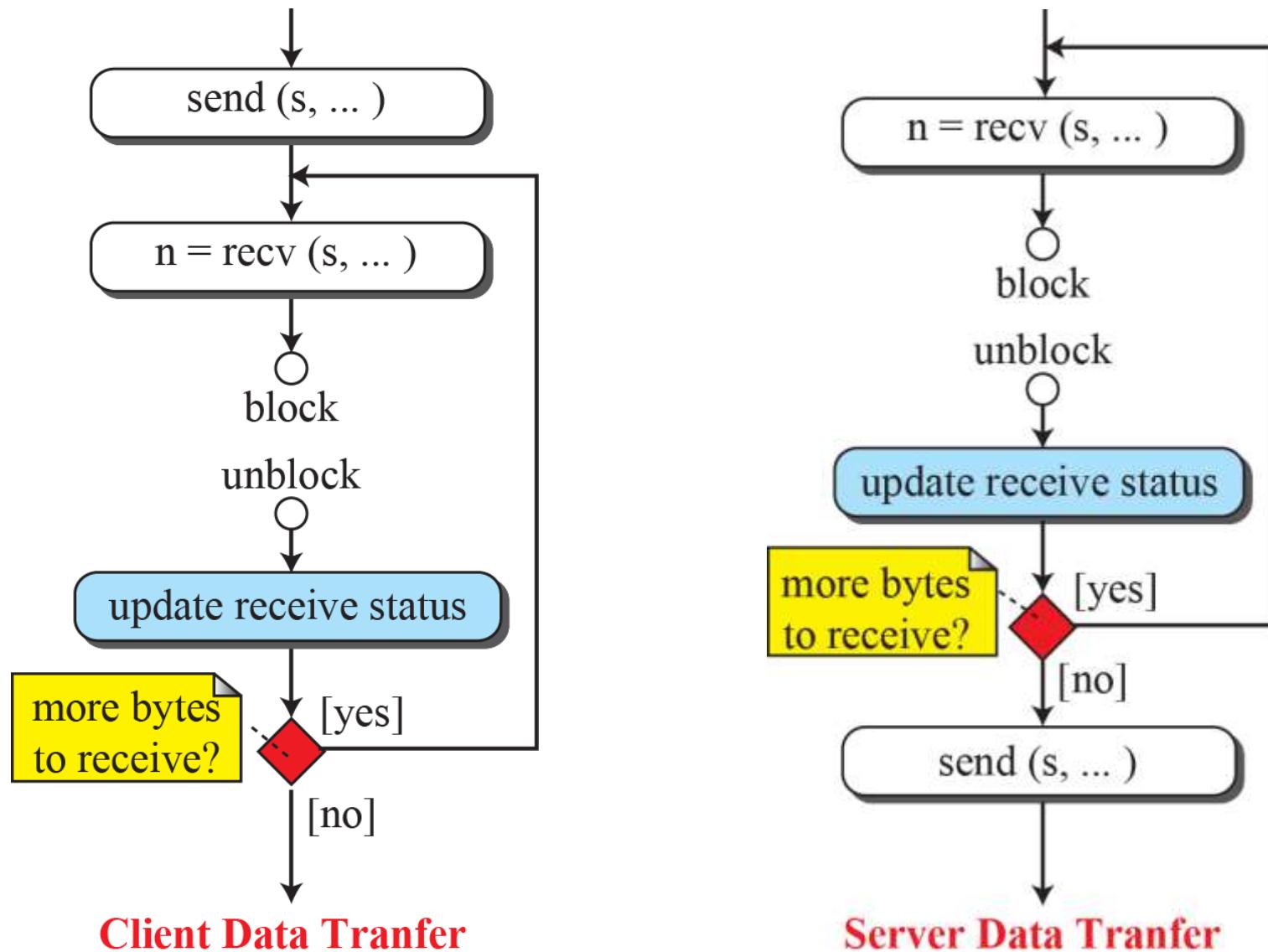
**Figure 2.61: Sockets used in TCP communication**



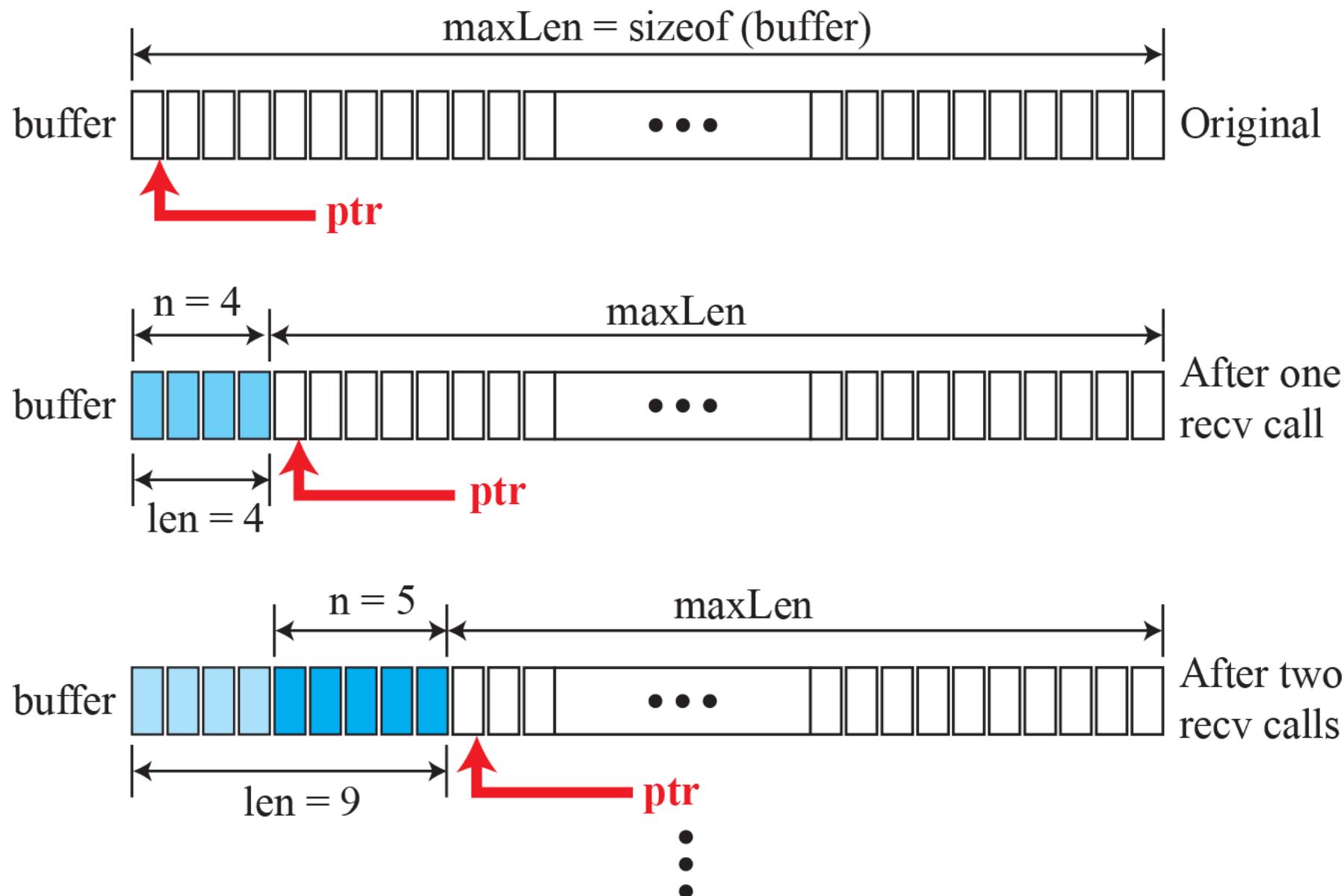
**Figure 2.62: Flow diagram for iterative TCP communication**

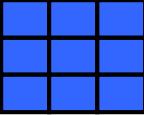


**Figure 2.63: Flow diagram for data-transfer boxes**



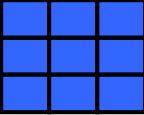
**Figure 2.64: Buffer used for receiving**





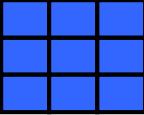
## Table 2.24: Echo server program using the services of TCP

```
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
```



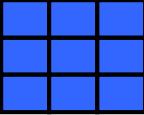
## Table 2.24: TCP Echo server program(continued)

```
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
```



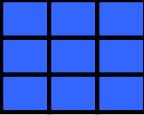
## Table 2.24: TCP Echo server program (continued)

```
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
```



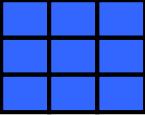
## Table 2.25: Echo client program using the services of TCP

```
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 2.25: TCP Echo client program (continued)

```
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 2.24: TCP Echo client program(continued)

```
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
```

# Chapter 2: Summary

- *Applications in the Internet are designed using either a client-server paradigm or a peer-to-peer paradigm. In a client-server paradigm, an application program, called a server, provides services and another application program, called a client, receives services. A server program is an infinite program; a client program is finite. In a peer-to-peer paradigm, a peer can be both a client and a server.*
- *The World Wide Web (WWW) is a repository of information linked together from points all over the world. Hypertext and hypermedia documents are linked to one another through pointers. The HyperText Transfer Protocol (HTTP) is the main protocol used to access data on the World Wide Web (WWW).*

# Chapter 2: Summary (continued)

- ❑ *File Transfer Protocol (FTP) is a TCP/IP client-server application for copying files from one host to another. FTP requires two connections for data transfer: a control connection and a data connection. FTP employs NVT ASCII for communication between dissimilar systems.*
  
- ❑ *Electronic mail is one of the most common applications on the Internet. The e-mail architecture consists of several components such as user agent (UA), main transfer agent (MTA), and main access agent (MAA). The protocol that implements MTA is called Simple Main Transfer Protocol (SMTP). Two protocols are used to implement MAA: Post Office Protocol, version 3 (POP3) and Internet Mail Access Protocol, version 4 (IMAP4).*

# Chapter 2: Summary (continued)

- ❑ *File Transfer Protocol (FTP) is a TCP/IP client-server application for copying files from one host to another. FTP requires two connections for data transfer: a control connection and a data connection. FTP employs NVT ASCII for communication between dissimilar systems.*
  
- ❑ *TELNET is a client-server application that allows a user to log into a remote machine, giving the user access to the remote system. When a user accesses a remote system via the TELNET process, this is comparable to a time-sharing environment.*

# Chapter 2: Summary (continued)

- ❑ *The Domain Name System (DNS) is a client-server application that identifies each host on the Internet with a unique name. DNS organizes the name space in a hierarchical structure to decentralize the responsibilities involved in naming. TELNET is a client-server application that allows a user to log into a remote machine, giving the user access to the remote system.*
  
- ❑ *In a peer-to-peer network, Internet users that are ready to share their resources become peers and form a network. Peer-to-peer networks are divided into centralized and decentralized.*