COMPUTER NETWORKS

SEMESTER – V	Subject Code 15CS52
IA Marks 20	Number of Lecture Hours/Week 4
Exam Marks 80	Total Number of Lecture Hours 50
Exam Hours 03	CREDITS – 04

Course objectives: This course will enable students to

• Demonstration of application layer protocols

• Discuss transport layer services and understand UDP and TCP protocols

• Explain routers, IP and Routing Algorithms in network layer

• Disseminate the Wireless and Mobile Networks covering IEEE 802.11 Standard

• Illustrate concepts of Multimedia Networking, Security and Network Management

Module – 1 Application Layer: Principles of Network Applications: Network Application Architectures, Processes Communicating, Transport Services Available to Applications, Transport Services Provided by the Internet, Application-Layer Protocols. The Web and HTTP: Overview of HTTP, Non-persistent and Persistent Connections, HTTP Message Format, User-Server Interaction: Cookies, Web Caching, The Conditional GET, File Transfer: FTP Commands & Replies, Electronic Mail in the Internet: SMTP, Comparison with HTTP, Mail Message Format, Mail Access Protocols, DNS; The Internet's Directory Service: Services Provided by DNS, Overview of How DNS Works, DNS Records and Messages, Peer-to-Peer Applications: P2P File Distribution, Distributed Hash Tables, Socket Programming: creating Network Applications: Socket Programming with UDP, Socket Programming with TCP.

T1: Chap 2 10 Hours

Module – 2 Transport Layer : Introduction and Transport-Layer Services: Relationship Between Transport and Network Layers, Overview of the Transport Layer in the Internet, Multiplexing and Demultiplexing: Connectionless Transport: UDP,UDP Segment Structure, UDP Checksum, Principles of Reliable Data Transfer: Building a Reliable Data Transfer Protocol, Pipelined Reliable Data Transfer Protocols, Go-Back-N, Selective repeat, Connection-Oriented Transport TCP: The TCP Connection, TCP Segment Structure, Round-Trip Time Estimation and Timeout, Reliable Data Transfer, Flow Control, TCP Connection Management, Principles of Congestion Control: The Causes and the Costs of Congestion, Approaches to Congestion Control, Network-assisted congestion-control example, ATM ABR Congestion control, TCP Congestion Control: Fairness.

T1: Chap 3 10 Hours

Module – 3 The Network layer: What's Inside a Router?: Input Processing, Switching, Output Processing, Where Does Queuing Occur? Routing control plane, IPv6,A Brief foray into IP Security, Routing Algorithms: The Link-State (LS) Routing Algorithm, The Distance-Vector (DV) Routing Algorithm, Hierarchical Routing, Routing in the Internet, Intra-AS Routing in the Internet: RIP, Intra-AS Routing in the Internet: OSPF, Inter/AS Routing: BGP, Broadcast and Multicast Routing: Broadcast Routing Algorithms and Multicast.

T1: Chap 4: 4.3-4.7

Module – 4 Mobile and Multimedia Networks: Cellular Internet Access: An Overview of Cellular Network Architecture, 3G Cellular Data Networks: Extending the Internet to Cellular subscribers, On to 4G:LTE,Mobility management: Principles, Addressing, Routing to a mobile node, Mobile IP, Managing mobility in cellular Networks, Routing calls to a Mobile user, Handoffs in GSM, Wireless and Mobility: Impact on Higher-layer protocols.

T1: Chap: 6 : 6.4-6.8 10 Hours

Module – 5 Multimedia Networking Applications: Properties of video, properties of Audio, Types of multimedia Network Applications, Streaming stored video: UDP Streaming, HTTP Streaming, Adaptive streaming and DASH, content distribution Networks, case studies: Netflix, You Tube and Kankan. Network Support for Multimedia: Dimensioning Best-Effort Networks, Providing Multiple Classes of Service, Diffserv, Per-Connection Quality-of-Service (QoS) Guarantees: Resource Reservation and Call Admission

T1: Chap: 7: 7.1,7.2,7.5 10 Hours

Course outcomes: The students should be able to:

• Explain principles of application layer protocols

• Recognize transport layer services and infer UDP and TCP protocols

• Classify routers, IP and Routing Algorithms in network layer

• Understand the Wireless and Mobile Networks covering IEEE 802.11 Standard

• Describe Multimedia Networking and Network Management

Question paper pattern: The question paper will have TEN questions. There will be TWO questions from each module. Each question will have questions covering all the topics under a module. The students will have to answer FIVE full questions, selecting ONE full question from each module.

Text Books:

1. James F Kurose and Keith W Ross, Computer Networking, A Top-Down Approach, Sixth edition, Pearson,2017 .

Reference Books:

1. Behrouz A Forouzan, Data and Communications and Networking, Fifth Edition, McGraw Hill, Indian Edition

2. Larry L Peterson and Brusce S Davie, Computer Networks, fifth edition, ELSEVIER

3. Andrew S Tanenbaum, Computer Networks, fifth edition, Pearson

4. Mayank Dave, Computer Networks, Second edition, Cengage Learning

CONTENTS

Module1:

Application Layer

1. Principles of Network Applications

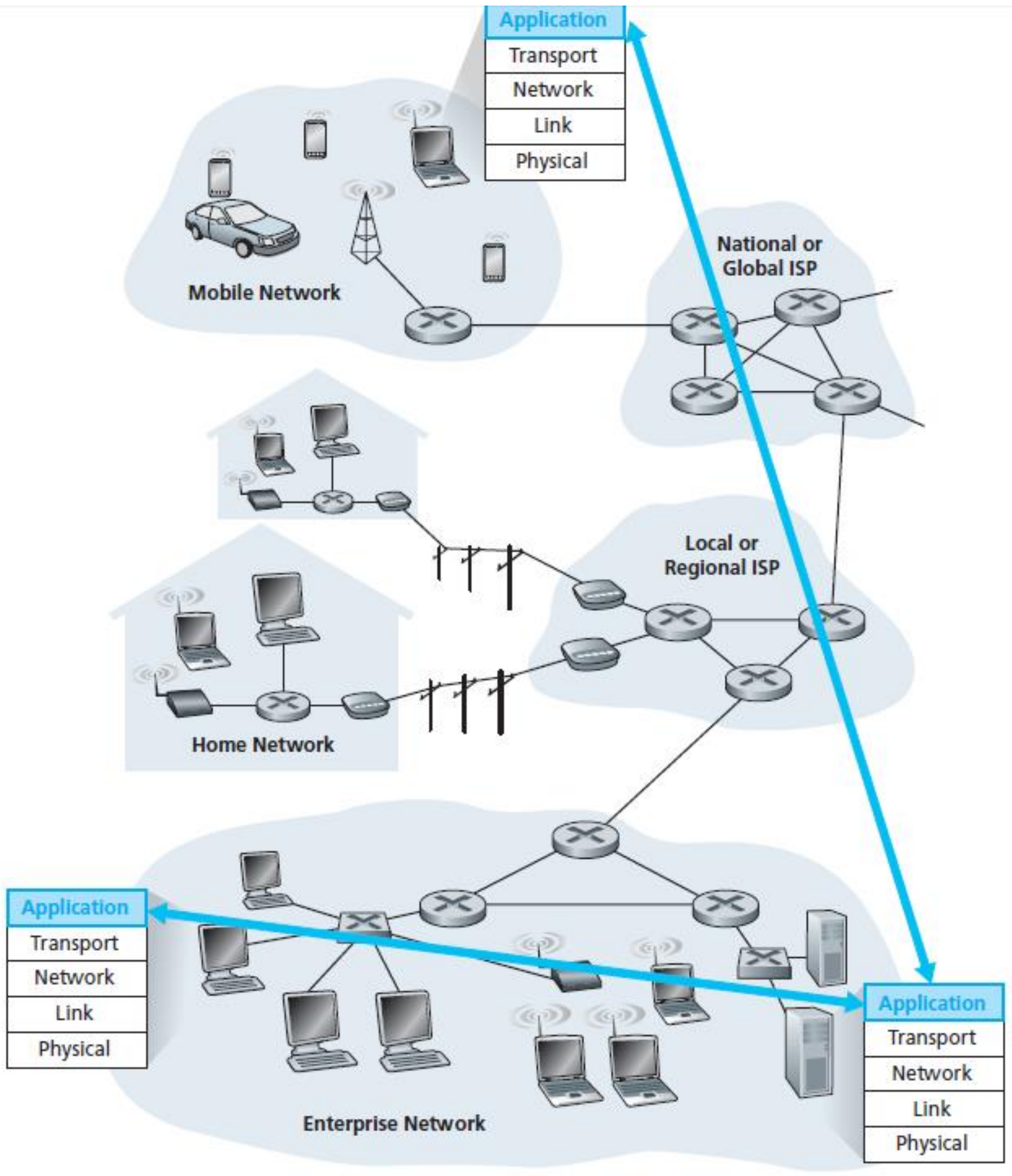


Figure 2.1 ♦ Communication for a network application takes place between end systems at the application layer

Network Application Architectures

The **application architecture** is designed by the application developer and dictates how the application is structured over the various end systems.
Two types: i) the client-server architecture , ii) the peer-to-peer (P2P) architecture
In a **client-server architecture**, there is an always-on host, called the *server*, which services requests from many other hosts, called *clients*. Example: Web browser and Web server.,(Web, FTP, Telnet, and e-mail.)
In a **P2P architecture**, there is minimal (or no) reliance on dedicated servers in data centers. Instead the application exploits direct communication between pairs of intermittently connected hosts, called *peers*. The peers are not owned by the service provider, but are instead desktops and laptops controlled by users, with most of the peers residing in homes, universities, and offices. Because the peers communicate without passing through a dedicated server, the architecture is called peer-to-peer. Example: file sharing (e.g., BitTorrent), peer-assisted download acceleration (e.g., Xunlei), Internet Telephony (e.g., Skype), and IPTV (e.g., Kankan and PPstream).

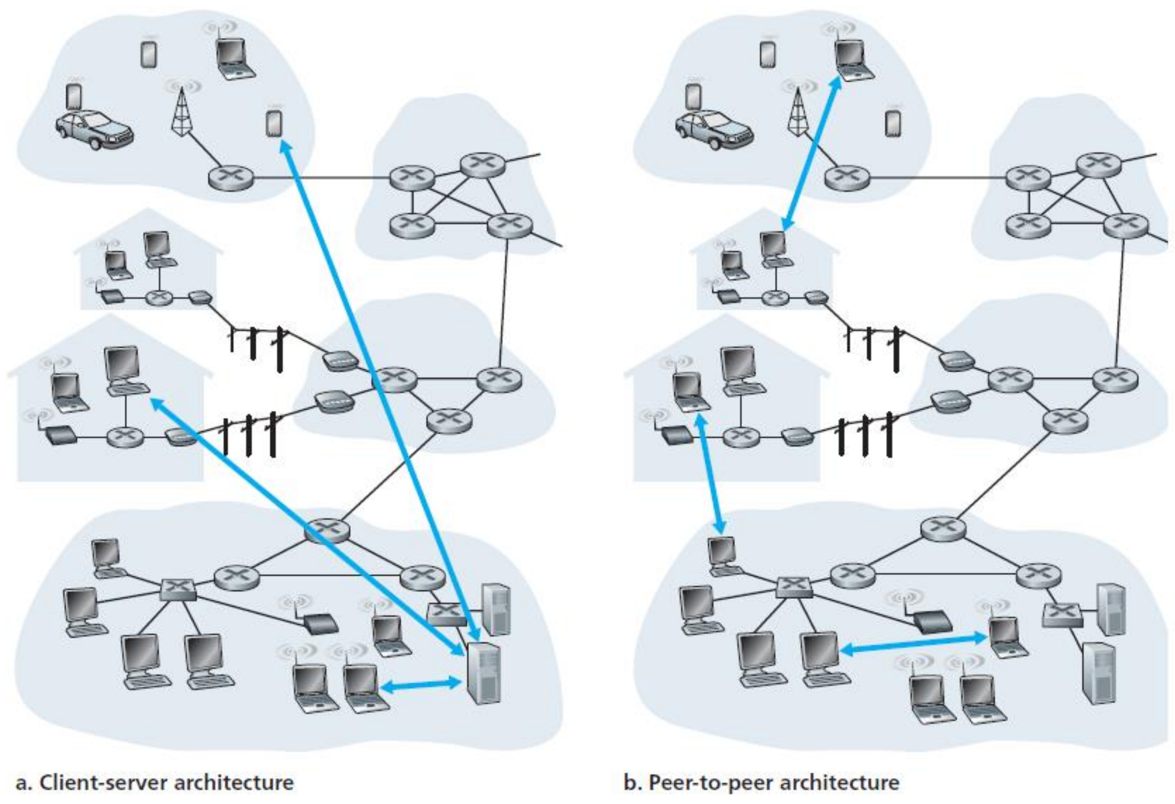


Figure 2.2 ♦ (a) Client-server architecture; (b) P2P architecture

Some applications have hybrid architectures, combining both client-server and P2P elements. For example, for many instant messaging applications, servers are used to track the IP addresses of users, but user-to-user messages are sent directly between user hosts (without passing through intermediate servers).

2. Processes Communicating

When processes are running on the same end system, they can communicate with each other with inter process communication, using rules that are governed by the end system's operating system. Processes on two different end systems communicate with each other by exchanging messages across the computer network. A sending process creates and sends messages into the network; a receiving process receives these messages and possibly responds by sending messages back.

Client and Server Processes:

A network application consists of pairs of processes that send messages to each other over a network. we typically label one of the two processes as the client and the other process as the server. With the Web, a browser is a client process and a Web server is a server process. With P2P file sharing, the peer that is downloading the file is labeled as the client, and the peer that is uploading the file is labeled as the server.

Definition: In the context of a communication session between a pair of processes, the process that initiates the communication (that is, initially contacts the other process at the beginning of the session) is labeled as the client. The process that waits to be contacted to begin the session is the server.

A process sends messages into, and receives messages from, the network through a software interface called a socket.

Illustration: A process is analogous to a house and its socket is analogous to its door. When a process wants to send a message to another process on another host, it shoves the message out its door (socket). This sending process assumes that there is a transportation infrastructure on the other side of its door that will transport the message to the door of the destination process. Once the message arrives at the destination host, the message passes through the receiving process's door (socket), and the receiving process then acts on the message.

As shown in this figure 2.3, a socket is the interface between the application layer and the transport layer within a host. It is also referred to as the Application Programming Interface (API) between the application and the network.

Addressing Processes:

In order for a process running on one host to send packets to a process running on another host, the receiving process needs to have an address. In the Internet, the host is identified by its IP address. To identify the receiving process, two pieces of information need to be specified: (1) the address of the host (IP Address) and (2) an identifier that specifies the receiving process in the destination host (Port No.). For example, a Web server is identified by port number 80. A mail server process (using the SMTP protocol) is identified by port number 25. IP address is a 32-bit quantity that we can think of as uniquely identifying the host.

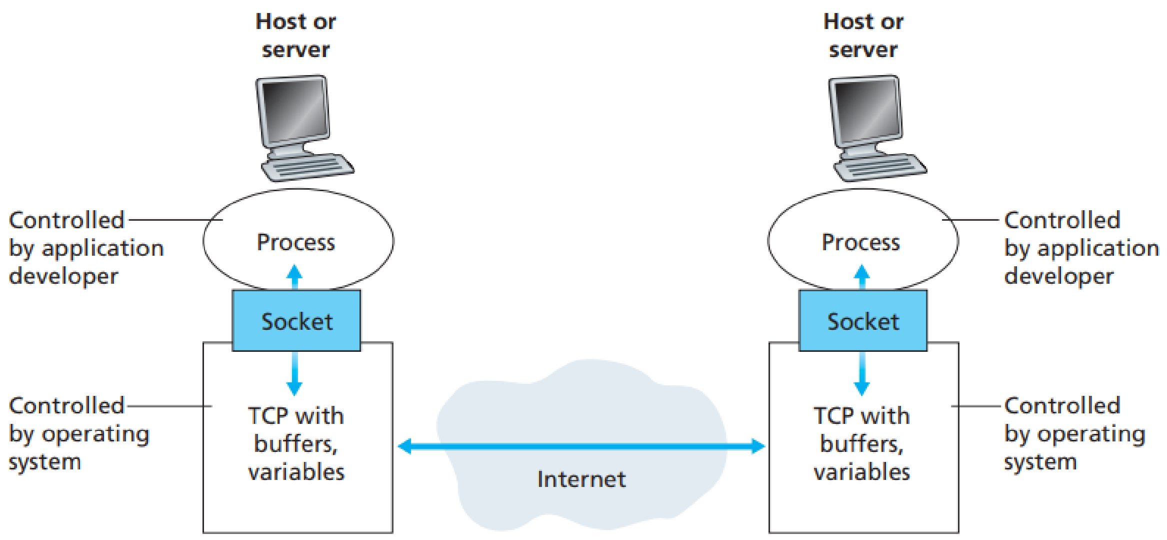

Host or server
Host or server
Controlled by application developer
Process
Socket
Controlled by operating system
TCP with buffers, variables
Internet
Process
Socket
TCP with buffers, variables
Controlled by application developer
Controlled by operating system

Figure 2.3 ♦ Application processes, sockets, and underlying transport protocol

3. Transport Services Available to Applications

Reliable Data Transfer : guaranteed data delivery service

Throughput: guaranteed throughput of *r* bits/sec, and the transport protocol would then ensure that the available throughput is always at least *r* bits/sec.

Timing: interactive real-time applications, such as Internet telephony, virtual environments, teleconferencing, and multiplayer games, all of which require tight timing constraints on data delivery in order to be effective.
Security: A transport protocol can also provide security services such as confidentiality, data integrity and end-point authentication.

4 **Transport Services Provided by the Internet**
When an application developer create a new network application for the Internet, one of the first decisions you have to make is whether to use UDP or TCP.

Application	Data Loss	Throughput	Time-Sensitive
File transfer/download	No loss	Elastic	No
E-mail	No loss	Elastic	No
Web documents	No loss	Elastic (few kbps)	No
Internet telephony/ Video conferencing	Loss-tolerant	Audio: few kbps–1Mbps Video: 10 kbps–5 Mbps	Yes: 100s of msec
Streaming stored audio/video	Loss-tolerant	Same as above	Yes: few seconds
Interactive games	Loss-tolerant	Few kbps–10 kbps	Yes: 100s of msec
Instant messaging	No loss	Elastic	Yes and no

Figure 2.4 ♦ Requirements of selected network applications

Application	Application-Layer Protocol	Underlying Transport Protocol
Electronic mail	SMTP [RFC 5321]	TCP
Remote terminal access	Telnet [RFC 854]	TCP
Web	HTTP [RFC 2616]	TCP
File transfer	FTP [RFC 959]	TCP
Streaming multimedia	HTTP (e.g., YouTube)	TCP
Internet telephony	SIP [RFC 3261], RTP [RFC 3550], or proprietary (e.g., Skype)	UDP or TCP

Figure 2.5 ♦ Popular Internet applications, their application-layer protocols, and their underlying transport protocols

5 **Application-Layer Protocols**

An **application-layer protocol** defines how an application's processes, running on different end systems, pass messages to each other. In particular, an application-layer protocol defines:

• The types of messages exchanged, for example, request messages and response messages

• The syntax of the various message types, such as the fields in the message and how the fields are delineated.

• The semantics of the fields, that is, the meaning of the information in the fields

• Rules for determining when and how a process sends messages and responds to messages

The Web and HTTP

HyperText Transfer Protocol (HTTP) is implemented in two programs: a client program and a server program. A **Web page** (also called a document) consists of objects. An **object** is simply a file—such as an HTML file, a JPEG image, a Java applet, or a video clip—that is addressable by a single URL. HTTP defines how Web clients request Web pages from Web servers and how servers transfer Web pages to clients. Figure 2.6. When a user requests a Web page (for example, clicks on a hyperlink), the browser sends HTTP request messages for the objects in the page to the server. The server receives the requests and responds with HTTP response messages that contain the objects.

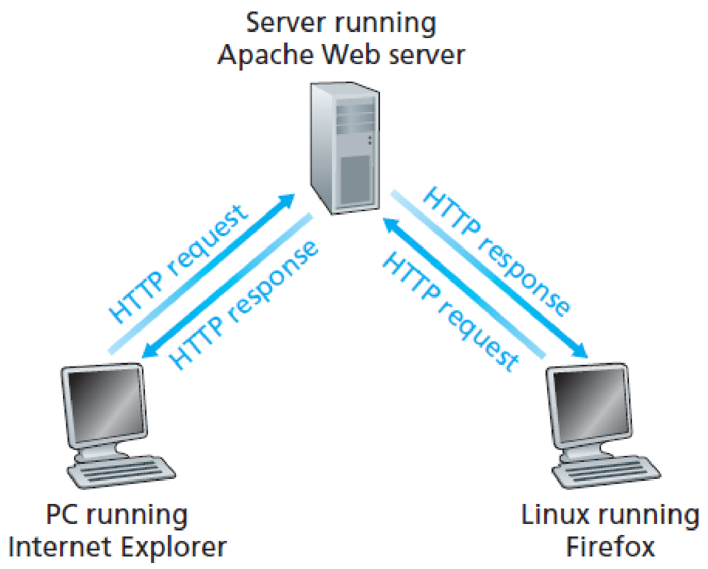


Figure 2.6 ♦ HTTP request-response behavior

HTTP uses TCP as its underlying transport protocol
The client sends HTTP request messages into its socket interface and receives HTTP response messages from its socket interface. Similarly, the HTTP server receives request messages from its socket interface and sends response messages into its socket interface.

Non-Persistent and Persistent Connections

When this client-server interaction is taking place over TCP, the application developer needs to make an important decision—should each request/response pair be sent over a *separate* TCP connection(**non-persistent connections**), or should all of the requests and their corresponding responses be sent over the *same* TCP connection (**persistent connections**)?

HTTP with Non-Persistent Connections

What happens when a user clicks on a hyperlink. As shown in Figure 2.7, this causes the browser to initiate a TCP connection between the browser and the Web server; this involves a "three-way handshake"—the client sends a small TCP segment to the server, the server acknowledges and responds with a small TCP segment, and, finally, the client acknowledges back to the server. The first two parts of the three way handshake take one RTT. After completing the first two parts of the handshake, the client sends the HTTP request message combined with the third part of the three-way handshake (the acknowledgment) into the TCP connection. Once the request message arrives at the server, the server sends the HTML file into the

TCP connection. This HTTP request/response eats up another RTT. Thus, roughly, the total response time is two RTTs plus the transmission time at the server of the HTML file.

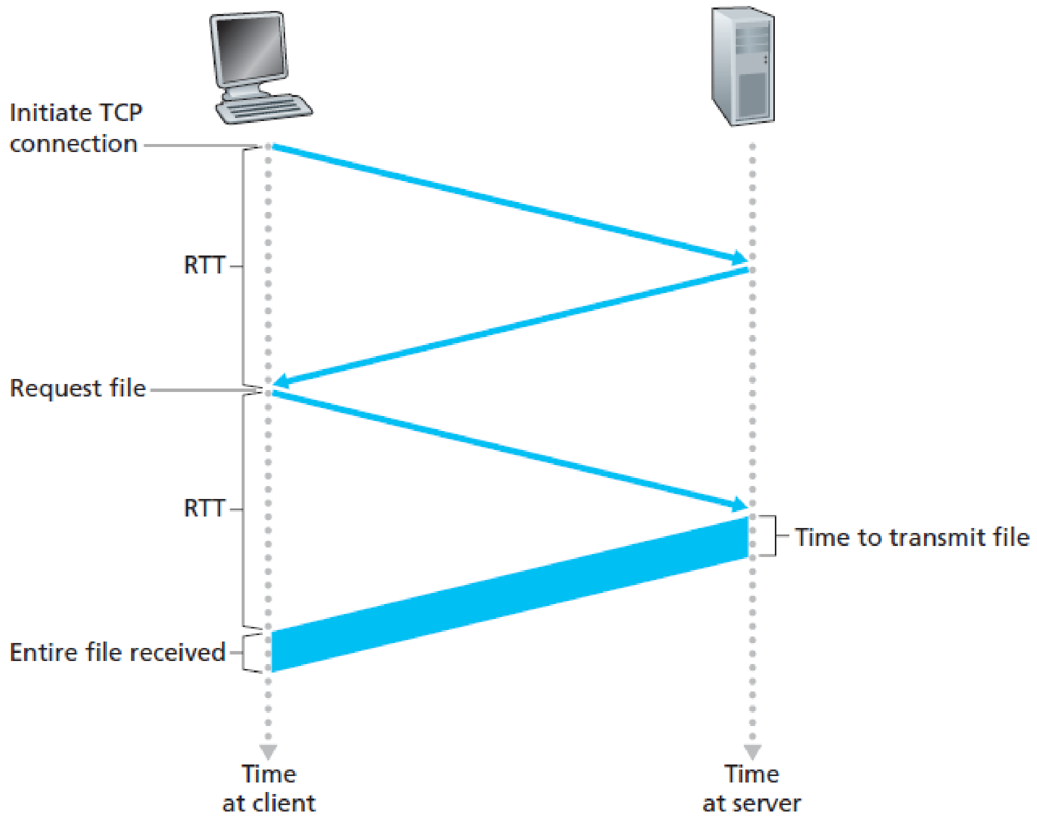


Figure 2.7 ♦ Back-of-the-envelope calculation for the time needed to request and receive an HTML file

HTTP with Persistent Connections

With persistent connections, the server leaves the TCP connection open after sending a response. Multiple Web pages residing on the same server can be sent from the server to the same client over a single persistent TCP connection. These requests for objects can be made back-to-back, without waiting for replies to pending requests (pipelining). Typically, the HTTP server closes a connection when it isn't used for a certain time. When the server receives the back-to-back requests, it sends the objects back-to-back.

HTTP Message Format

There are two types of HTTP messages: request messages and response messages.

HTTP Request Message

A typical HTTP request message:

GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu
2.2 • THE WEB AND HTTP **103**
Connection: close
User-agent: Mozilla/5.0
Accept-language: fr

The first line of an HTTP request message is called the **request line**; the subsequent lines are called the **header lines**. The request line has three fields: the method field, the URL field, and the HTTP version field. The method field can take on several different values, including GET, POST, HEAD, PUT, and DELETE. The great majority of HTTP request messages use the GET method.
The header line Host: www.someschool.edu specifies the host on which the object resides.
Connection: it wants the server to close the connection after sending the requested object.
The User-agent: header line specifies the user agent, that is, the browser type that is making the request to the server. Here the user agent is Mozilla/5.0, a Firefox browser.
Accept language: header indicates that the user prefers to receive a French version of the object, if such an object exists on the server; otherwise, the server should send its default version.

General format of a request message, as shown in Figure 2.8.

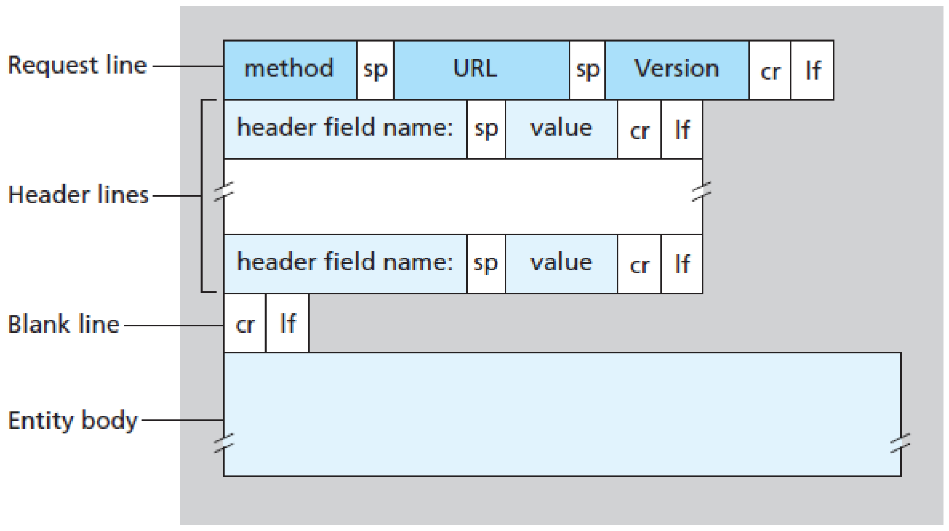


Figure 2.8 ♦ General format of an HTTP request message

HTTP Response Message

A typical HTTP response message:

HTTP/1.1 200 OK
Connection: close
Date: Tue, 09 Aug 2011 15:44:04 GMT
Server: Apache/2.2.3 (CentOS)
Last-Modified: Tue, 09 Aug 2011 15:11:03 GMT
Content-Length: 6821
Content-Type: text/html
(data data data data data ...)

It has three sections: an initial **status line**, six **header lines**, and then the **entity body**.

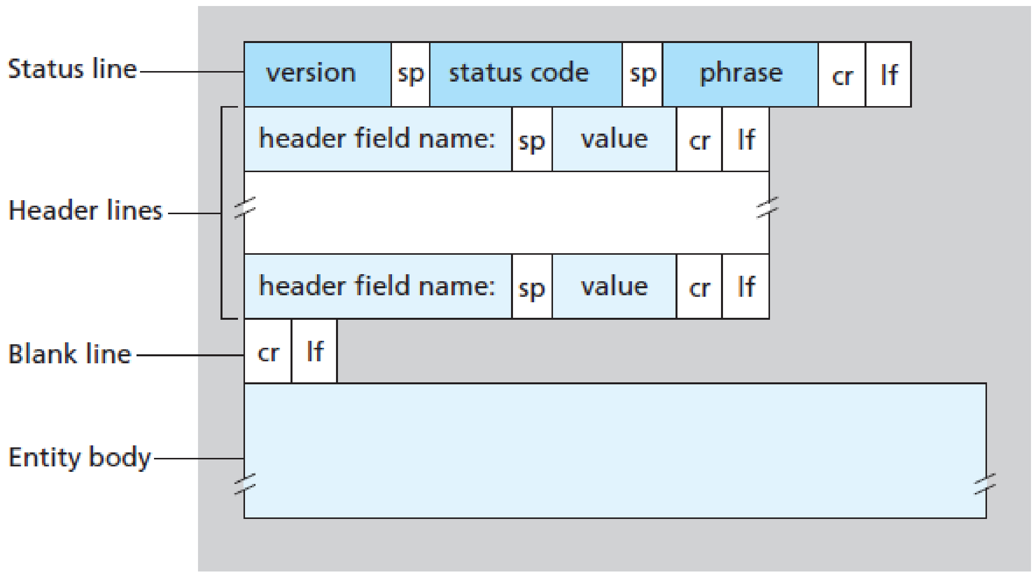


Figure 2.9 ♦ General format of an HTTP response message

The status line has three fields: the protocol version field, a status code, and a corresponding status message. The status code and associated phrase indicate the result of the request. Some common status codes and associated phrases include:

- 200 OK: Request succeeded and the information is returned in the response.
- 301 Moved Permanently: Requested object has been permanently moved; the new URL is specified in Location: header of the response message. The client software will automatically retrieve the new URL.
- 400 Bad Request: This is a generic error code indicating that the request could not be understood by the server.
- 404 Not Found: The requested document does not exist on this server.
- 505 HTTP Version Not Supported: The requested HTTP protocol version is not supported by the server.

User-Server Interaction: Cookies

Cookies allow sites to keep track of users. Most major commercial Web sites use cookies today. cookie technology has four components: (1) a cookie header line in the HTTP response message; (2) a cookie header line in the HTTP request message; (3) a cookie file kept on the user’s end system and managed by the user’s browser; and (4) a back-end database at the Web site.

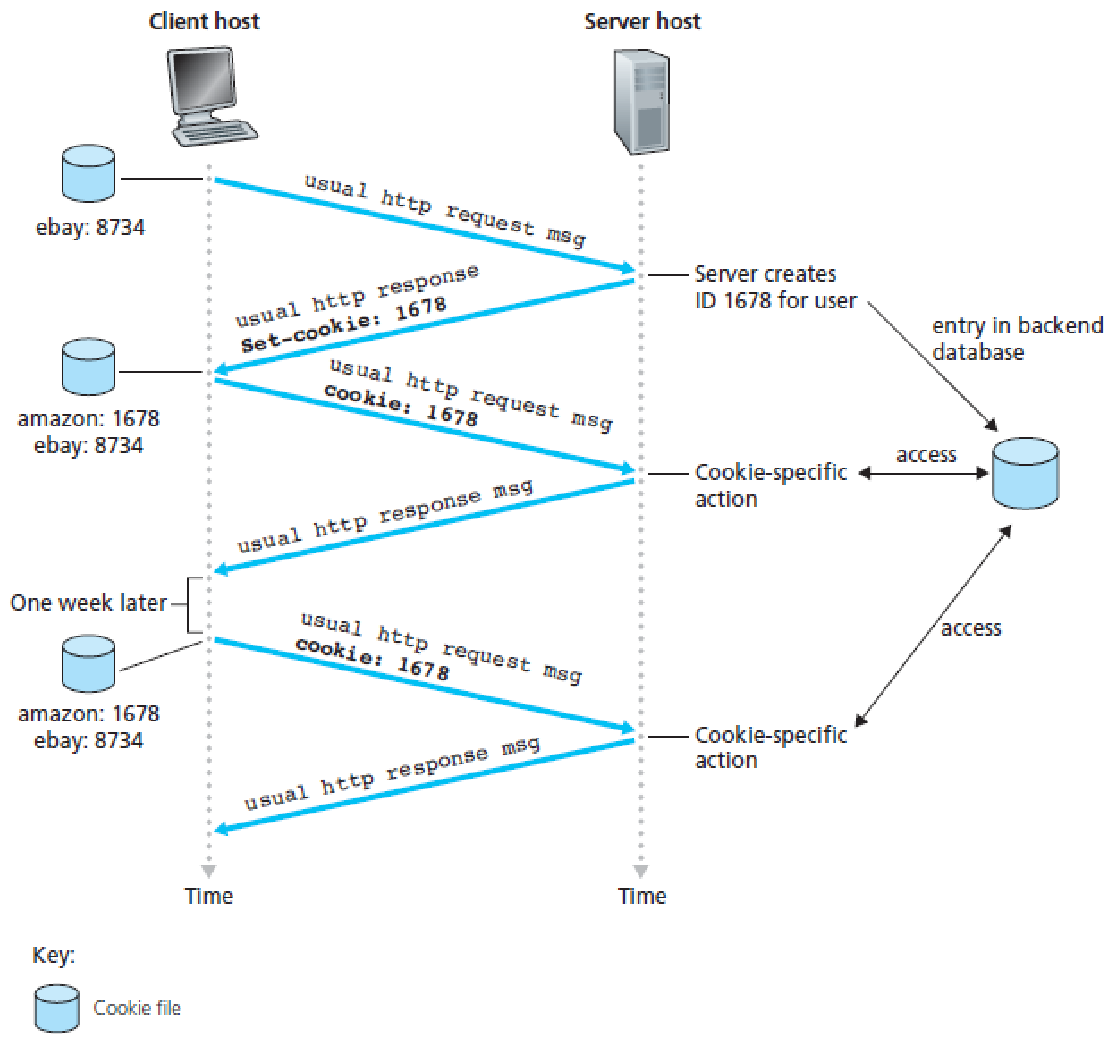


Figure 2.10 ♦ Keeping user state with cookies

Web Caching

A **Web cache**—also called a **proxy server**—is a network entity that satisfies HTTP requests on the behalf of an origin Web server. The Web cache has its own disk storage and keeps copies of recently requested objects in this storage. As shown in Figure 2.11, a user's browser can be configured so that all of the user's HTTP requests are first directed to the Web cache. Once a browser is configured, each browser request for an object is first directed to the Web cache.

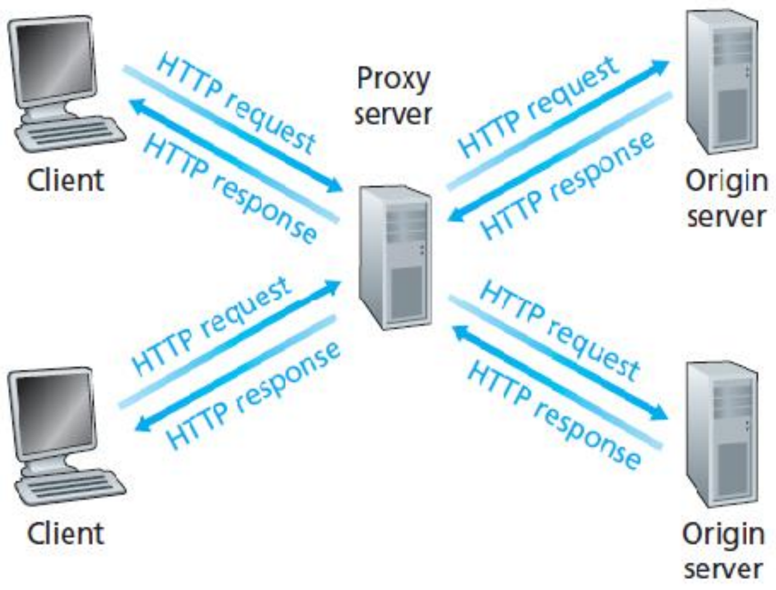


Figure 2.11 ♦ Clients requesting objects through a Web cache

3 **File Transfer: FTP**

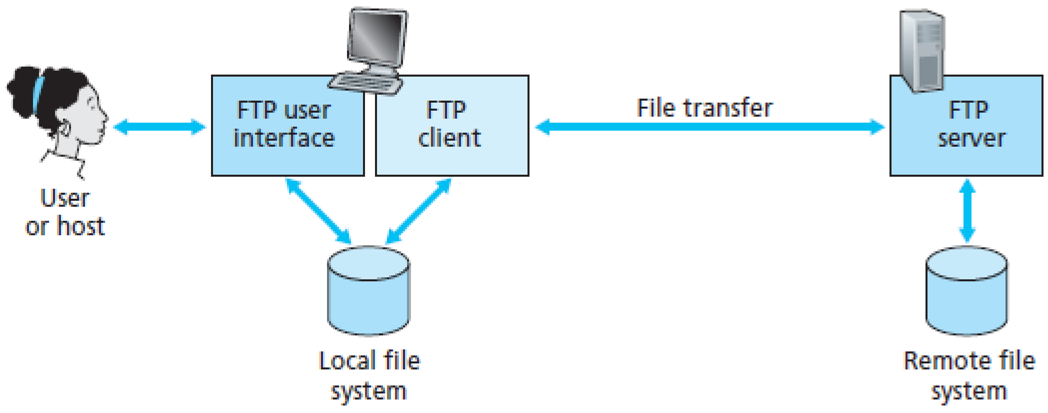


Figure 2.14 ♦ FTP moves files between local and remote file systems

The user must provide a user identification and a password. After providing this authorization information, the user can transfer files from the local file system to the remote file system and vice versa.
The user first provides the hostname of the remote host, causing the FTP client process in the local host to establish a TCP connection with the FTP server process in the remote host. The user then provides the user identification and password, which are sent over the TCP connection as part of FTP commands. Once the server has authorized the user, the user copies one or more files stored in the local file system into the remote file system (or vice versa).

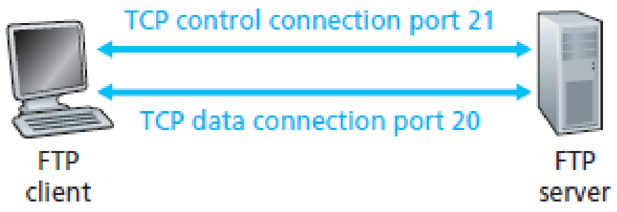


Figure 2.15 ♦ Control and data connections

The FTP control and data connections are illustrated in Figure 2.15.

FTP uses two parallel TCP connections to transfer a file, a **control connection** and a **data connection**. The control connection is used for sending control information between the two hosts—information such as user identification, password, commands to change remote directory, and commands to "put" and "get" files. The data connection is used to actually send a file.

When a user starts an FTP session with a remote host, the client side of FTP (user) first initiates a control TCP connection with the server side (remote host) on server port number 21. The client side of FTP sends the user identification and password over this control connection. The client side of FTP also sends, over the control connection, commands to change the remote directory. When the server side receives a command for a file transfer over the control connection (either to, or from, the remote host), the server side initiates a TCP data connection to the client side. FTP sends exactly one file over the data connection and then closes the data connection. If, during the same session, the user wants to transfer another file, FTP opens another data connection. Thus, with FTP, the control connection remains open throughout the duration of the user session, but a new data connection is created for each file transferred within a session (that is, the data connections are non-persistent).

FTP Commands and Replies

Each command consists of four uppercase ASCII characters, some with optional arguments. Some of the more common commands are given below:

- `USER` username: Used to send the user identification to the server.
- `PASS` password: Used to send the user password to the server.
- `LIST`: Used to ask the server to send back a list of all the files in the current remote directory. The list of files is sent over a (new and non-persistent) data connection rather than the control TCP connection.
- `RETR` filename: Used to retrieve (that is, get) a file from the current directory of the remote host. This command causes the remote host to initiate a data connection and to send the requested file over the data connection.
- `STOR` filename: Used to store (that is, put) a file into the current directory of the remote host.

There is typically a one-to-one correspondence between the command that the user issues and the FTP command sent across the control connection. Each command is followed by a reply, sent from server to client. The replies are three-digit numbers, with an optional message following the number. This is similar in structure to the status code and phrase in the status line of the HTTP response message.

Some typical replies, along with their possible messages, are as follows:

- `331 Username OK, password required`
- `125 Data connection already open; transfer starting`
- `425 Can't open data connection`
- `452 Error writing file`

4. Electronic Mail in the Internet

It has three major components: **user agents**, **mail servers**, and the **Simple Mail Transfer Protocol (SMTP)**.

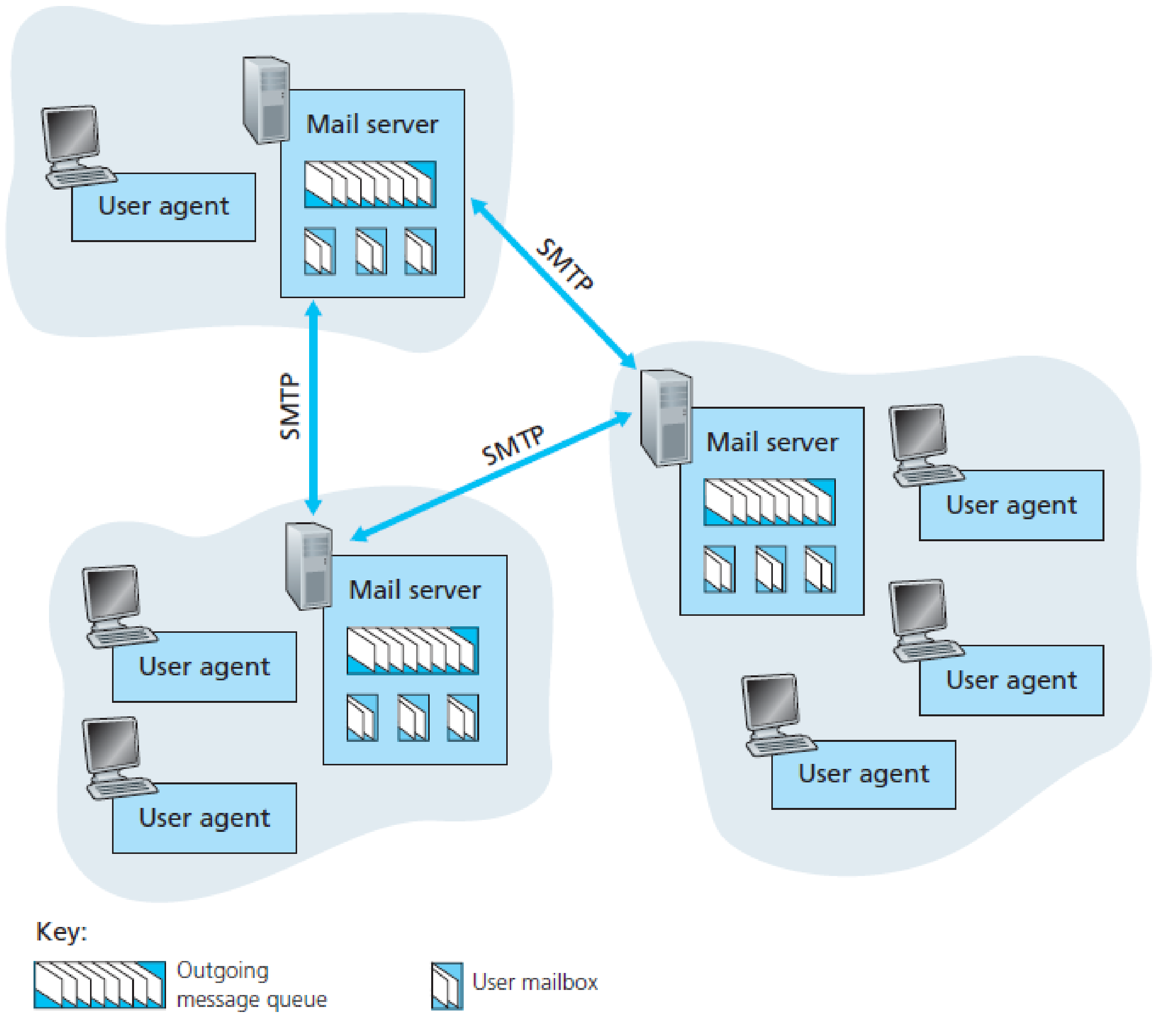


Figure 2.16 ♦ A high-level view of the Internet e-mail system

User agents allow users to read, reply to, forward, save, and compose messages. Microsoft Outlook and Apple Mail are examples of user agents for e-mail. When Alice is finished composing her message, her user agent sends the message to her mail server, where the message is placed in the mail server's outgoing message queue. When Bob wants to read a message, his user agent retrieves the message from his mailbox in his mail server. A typical message starts its journey in the sender's user agent, travels to the sender's mail server, and travels to the recipient's mail server, where it is deposited in the recipient's mailbox. Alice's server holds the message in a **message queue** and attempts to transfer the message later. Reattempts are often done every 30 minutes or so; if there is no success after several days, the server removes the message and notifies the sender (Alice) with an e-mail message.

SMTP uses the reliable data transfer service of TCP to transfer mail from the sender's mail server to the recipient's mail server.

4.1 **SMTP**

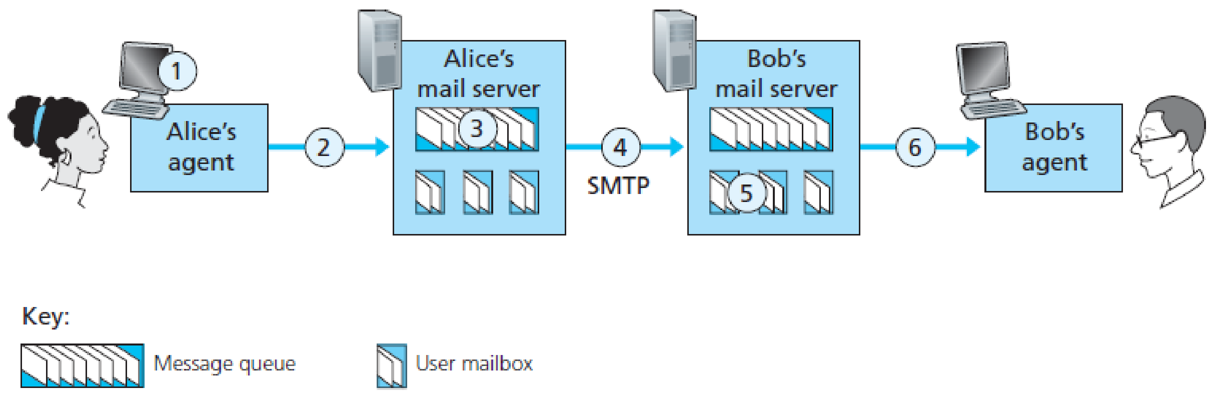


Figure 2.17 ♦ Alice sends a message to Bob

Suppose Alice wants to send Bob a simple ASCII message.

1. Alice invokes her user agent for e-mail, provides Bob's e-mail address (for example, bob@someschool.edu), composes a message, and instructs the user agent to send the message.
2. Alice's user agent sends the message to her mail server, where it is placed in a message queue.
3. The client side of SMTP, running on Alice's mail server, sees the message in the message queue. It opens a TCP connection to an SMTP server, running on Bob's mail server.
4. After some initial SMTP handshaking, the SMTP client sends Alice's message into the TCP connection.
5. At Bob's mail server, the server side of SMTP receives the message. Bob's mail server then places the message in Bob's mailbox.
6. Bob invokes his user agent to read the message at his convenience.

POP3: POP3 is an extremely simple mail access protocol. It is defined in [RFC 1939], which is short and quite readable. Because the protocol is so simple, its functionality is rather limited. POP3 begins when the user agent (the client) opens a TCP connection to the mail server (the server) on port 110. With the TCP connection established, POP3 progresses through three phases: authorization, transaction, and update.

During the first phase, authorization, the user agent sends a username and a password to authenticate the user. During the second phase, transaction, the user agent retrieves messages; also during this phase, the user agent can mark messages for deletion, remove deletion marks, and obtain mail statistics. Update: the mail server deletes the messages that were marked for deletion.

IMAP: An IMAP server will associate each message with a folder; when a message first arrives at the server, it is associated with the recipient's INBOX folder. The recipient can then move the message into a new, user-created folder, read the message, delete the message, and so on. The IMAP protocol provides commands to allow users to create folders and move messages from one folder to another. IMAP also provides commands that allow users to search remote folders for messages matching specific criteria.

5. **DNS—The Internet's Directory Service**

The main task of the Internet's **domain name system (DNS)** is to translate hostnames to IP addresses. In order to deal with the issue of scale, the DNS uses a large number of servers, organized in a hierarchical fashion and distributed around the world. No single DNS server has all of the mappings for all of the hosts in the Internet. Instead, the mappings are distributed across the DNS servers. There are three classes of DNS servers—root DNS servers, top-level domain (TLD) DNS servers, and authoritative DNS servers

Root DNS servers. In the Internet there are 13 root DNS servers (labeled A through M), most of which are located in North America.

Top-level domain (TLD) servers. These servers are responsible for top-level domains such as com, org, net, edu, and gov, and all of the country top-level domains such as uk, fr, ca, and jp. The company Verisign Global Registry Services maintains the TLD servers for the com top-level domain, and the company Educause maintains the TLD servers for the edu top-level domain.

Authoritative DNS servers. Every organization with publicly accessible hosts (such as Web servers and mail servers) on the Internet must provide publicly accessible DNS records that map the names of those hosts to IP addresses. An organization's authoritative DNS server houses these DNS records.

The example shown in Figure 2.21 makes use of both **recursive queries** and **iterative queries**. The query sent from cis.poly.edu to dns.poly.edu is a recursive query, since the query asks dns.poly.edu to obtain the mapping on its behalf. But the subsequent three queries are iterative since all of the replies are directly returned to dns.poly.edu.

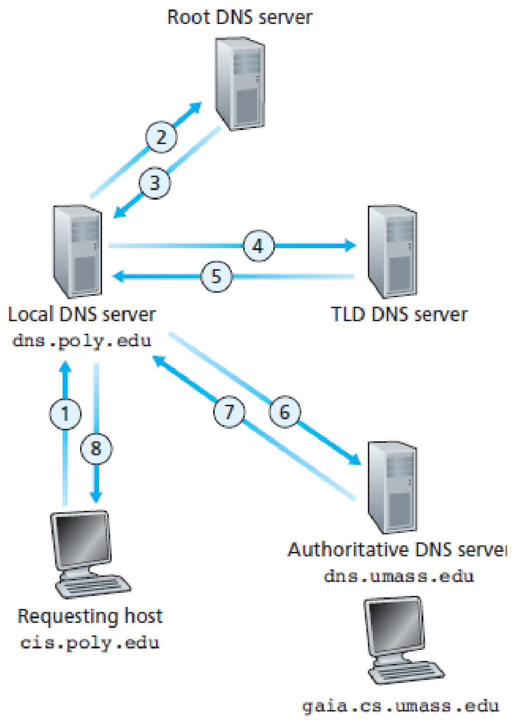


Figure 2.21 ♦ Interaction of the various DNS servers

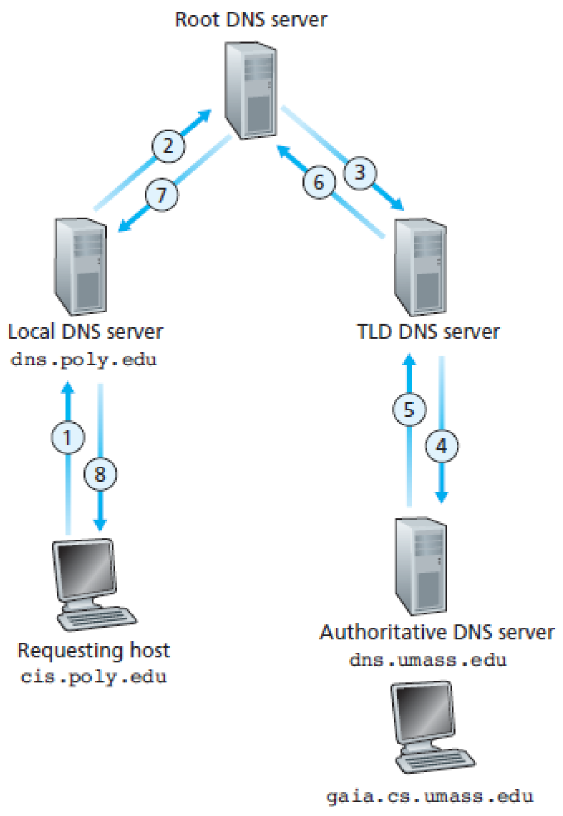


Figure 2.22 ♦ Recursive queries in DNS

DNS Records and Messages

The DNS servers that together implement the DNS distributed database store **resource records (RRs)**, including RRs that provide hostname-to-IP address mappings. Each DNS reply message carries one or more resource records. Resource record is a four-tuple that contains the following fields:

(Name, Value, Type, TTL)

TTL is the time to live of the resource record; it determines when a resource should be removed from a cache. In the example records given below, we ignore the TTL field. The meaning of Name and Value depend on Type:

• If Type=A, then Name is a hostname and Value is the IP address for the hostname. Thus, a Type A record provides the standard hostname-to-IP address mapping. As an example, (relay1.bar.foo.com, 145.37.93.126, A) is a Type A record.

• If Type=NS, then Name is a domain (such as foo.com) and Value is the hostname of an authoritative DNS server that knows how to obtain the IP addresses for hosts in the domain. This record is used to route DNS queries further along in the query chain. As an example, (foo.com, dns.foo.com, NS) is a Type NS record.

• If Type=CNAME, then Value is a canonical hostname for the alias hostname Name. This record can provide querying hosts the canonical name for a hostname. As an example, (foo.com, relay1.bar.foo.com, CNAME) is a CNAME record.

• If Type=MX, then Value is the canonical name of a mail server that has an alias hostname Name. As an example, (foo.com, mail.bar.foo.com, MX) is an MX record.

DNS Messages

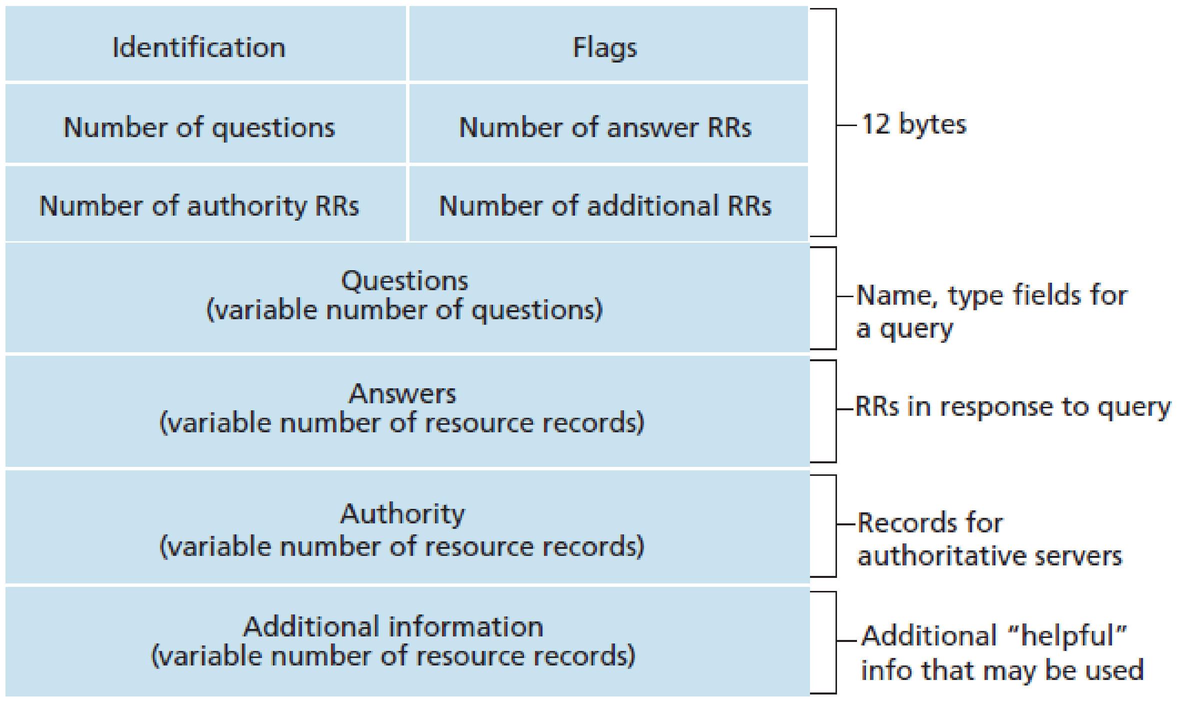

Identification
Flags
Number of questions
Number of answer RRs
Number of authority RRs
Number of additional RRs
12 bytes
Questions
(variable number of questions)
Name, type fields for a query
Answers
(variable number of resource records)
RRs in response to query
Authority
(variable number of resource records)
Records for authoritative servers
Additional information
(variable number of resource records)
Additional "helpful" info that may be used

Figure 2.23 ♦ DNS message format

The first 12 bytes is the *header section,* which has a number of fields. The first field is a 16-bit number that identifies the query. This identifier is copied into the reply message to a query, allowing the client to match received replies with sent queries.
The *question section* contains information about the query that is being made.
In a reply from a DNS server, the *answer section* contains the resource records for the name that was originally queried.
The *authority section* contains records of other authoritative servers.
The *additional section* contains other helpful records.

6 **Peer-to-Peer Applications**

6.1 **P2P File Distribution**

Distributing a large file from a single server to a large number of hosts (called peers).
Calculating the distribution time for the P2P architecture is somewhat more complicated than for the client-server architecture, since the distribution time depends on how each peer distributes portions of the file to the other peers.

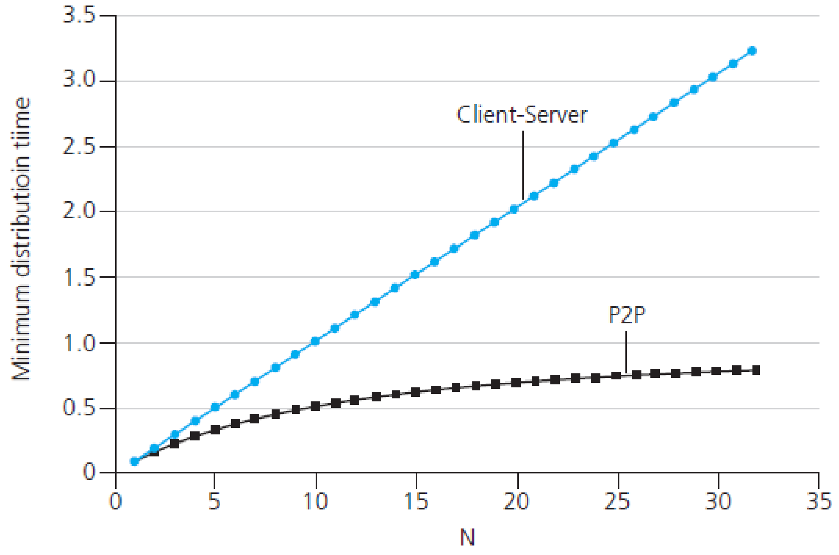


Figure 2.25 ♦ Distribution time for P2P and client-server architectures

N : Number of Peers

BitTorrent

BitTorrent is a popular P2P protocol for file distribution. In BitTorrent lingo, the collection of all peers participating in the distribution of a particular file is called a *torrent*. Peers in a torrent download equal-size *chunks* of the file from one another, with a typical chunk size of 256 KBytes. When a peer first joins a torrent, it has no chunks. Over time it accumulates more and more chunks. While it downloads chunks it also uploads chunks to other peers. Once a peer has acquired the entire file, it may (selfishly) leave the torrent, or (altruistically) remain in the torrent and continue to upload chunks to other peers. Also, any peer may leave the torrent at any time with only a subset of chunks, and later rejoin the torrent.

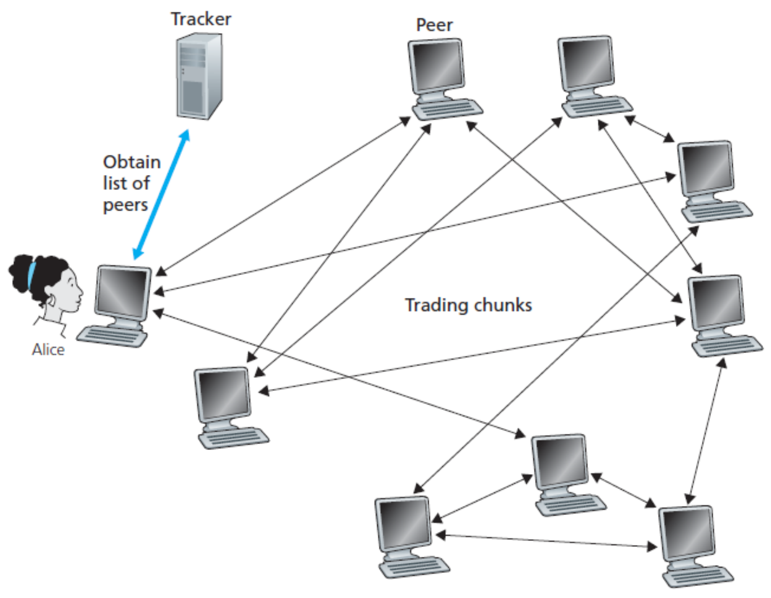


Figure 2.26 ♦ File distribution with BitTorrent

As shown in Figure 2.26, when a new peer, Alice, joins the torrent, the tracker randomly selects a subset of peers (for concreteness, say 50) from the set of participating peers, and sends the IP addresses of these 50 peers to Alice. Possessing this list of peers, Alice attempts to establish concurrent TCP connections with all the peers on this list. Let's call all the peers with which Alice succeeds in establishing a TCP connection "neighboring peers." (In Figure 2.26, Alice is shown to have only three neighboring peers. Normally, she would have many more.) As time evolves, some of these peers may leave and other peers (outside the initial 50) may attempt to establish TCP connections with Alice. So a peer's neighboring peers will fluctuate over time.

6.2 **Distributed Hash Tables (DHTs)**

Let's consider here how to implement a simple database in a P2P network.
A centralized version of this simple database, which will simply contain (key, value) pairs. An example key-value pair is (Led Zeppelin IV, 128.17.123.38). We query the database with a key. If there are one or more key-value pairs in the database that match the query key, the database returns the corresponding values.
Consider how to build a distributed, P2P version of this database that will store the (key, value) pairs over millions of peers. In the P2P system, each peer will only hold a small subset of the totality of the (key, value) pairs. We'll allow any peer to query the distributed database with a particular key. The distributed database will then locate the peers that have the corresponding (key, value) pairs and return the key-value pairs to the querying peer. Any peer will also be allowed to insert new key-value pairs into the database. Such a distributed database is referred to as a **distributed hash table (DHT)**.
In DHT randomly scatter the (key, value) pairs across all the peers and have each peer maintain a list of the IP addresses of all participating peers. In this design, the querying peer sends its query to all other peers, and the peers containing the (key, value) pairs that match the key can respond with their matching pairs. Such an approach is completely unscalable, of course, as it would require each peer to not only know about all other peers (possibly millions of such peers!) but even worse, have each query sent to *all* peers.

Circular DHT

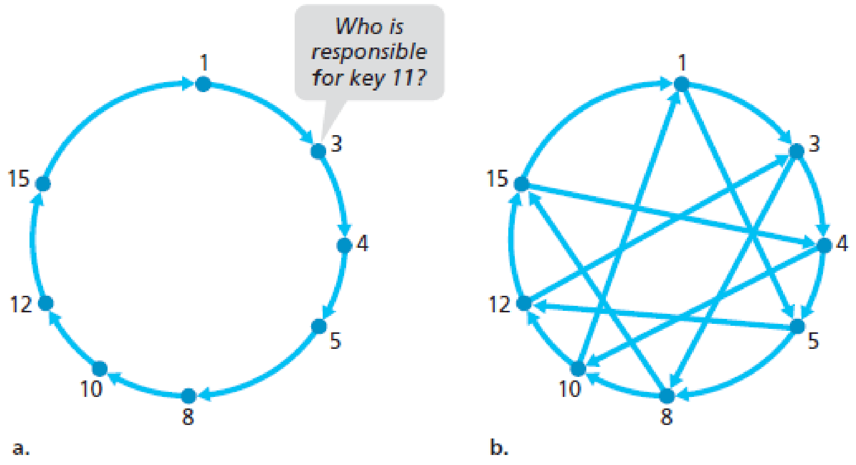


Figure 2.27 ♦ (a) A circular DHT. Peer 3 wants to determine who is responsible for key 11. (b) A circular DHT with shortcuts

Each peer is only aware of its immediate successor and predecessor; for example, peer 5 knows the IP address and identifier for peers 8 and 4 but does not necessarily know anything about any other peers that may be in the DHT. This circular arrangement of the peers is a special case of an **overlay network**. In an overlay network, the peers form an abstract logical network which resides above the "underlay" computer network consisting of physical links, routers, and hosts. The links in an overlay network are not physical links, but are simply virtual liaisons between pairs of peers. In the overlay in Figure 2.27(a), there are eight peers and eight overlay links; in the overlay in Figure 2.27(b) there are eight peers and 16 overlay links. A single overlay link typically uses many physical links and physical routers in the underlay network.

7 Socket Programming: Creating Network Applications

7.1 Socket Programming with UDP

Demonstration of socket programming for both UDP and TCP:
1. The client reads a line of characters (data) from its keyboard and sends the data to the server.
2. The server receives the data and converts the characters to uppercase.
3. The server sends the modified data to the client.
4. The client receives the modified data and displays the line on its screen.

We'll begin with the UDP client, which will send a simple application-level message to the server. In order for the server to be able to receive and reply to the client's message, it must be ready and running—that is, it must be running as a process before the client sends its message.

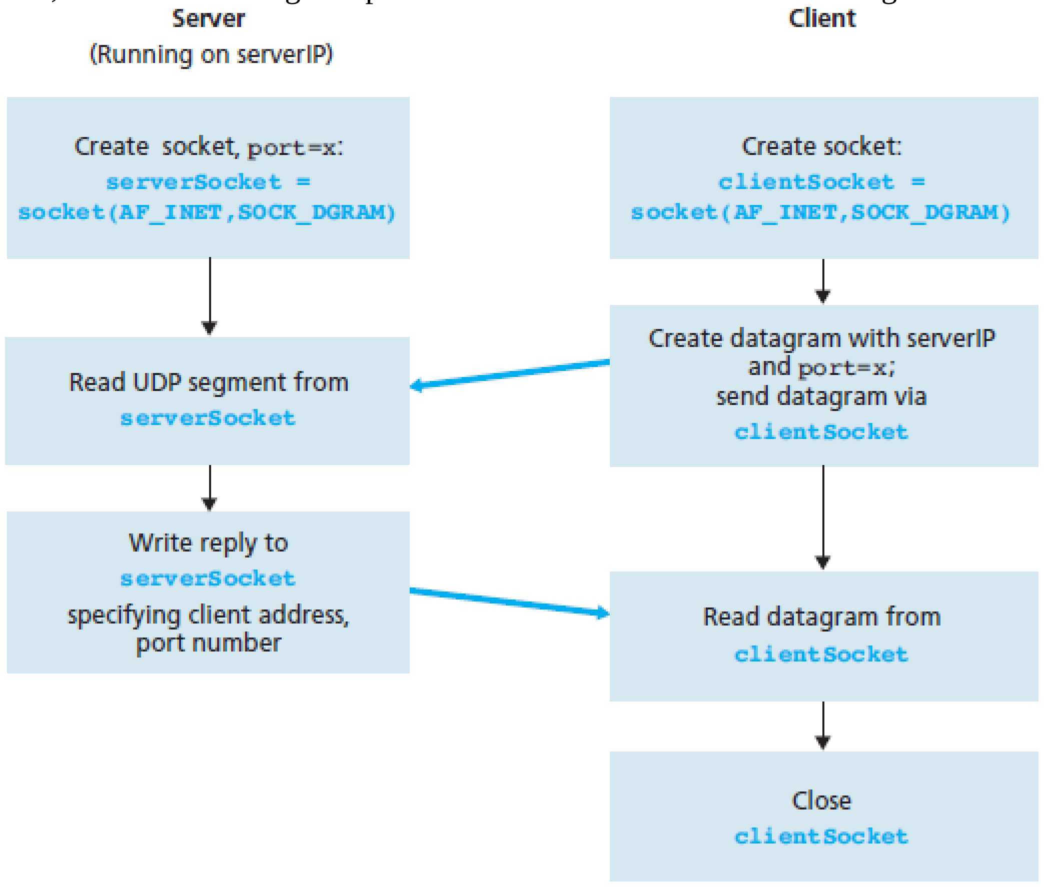

Server
(Running on serverIP)
Client
Create socket, port=x:
serverSocket = socket(AF_INET,SOCK_DGRAM)
Create socket:
clientSocket = socket(AF_INET,SOCK_DGRAM)
Read UDP segment from serverSocket
Create datagram with serverIP and port=x; send datagram via clientSocket
Write reply to serverSocket specifying client address, port number
Read datagram from clientSocket
Close clientSocket

Figure 2.28 ♦ The client-server application using UDP

AF_INET indicates that the underlying network is using IPv4.
The second parameter indicates that the socket is of type SOCK_DGRAM, which means it is a UDP socket.

7.2 Socket Programming with TCP

With the server process running, the client process can initiate a TCP connection to the server. This is done in the client program by creating a TCP socket. When the client creates its TCP socket, it specifies the

address of the welcoming socket in the server, namely, the IP address of the server host and the port number of the socket. After creating its socket, the client initiates a three-way handshake and establishes a TCP connection with the server. The three-way handshake, which takes place within the transport layer, is completely invisible to the client and server programs.
A special socket—that welcomes some initial contact from a client process running on an arbitrary host: TCP socket object that we call serverSocket; the newly created socket dedicated to the client making the connection is called connectionSocket.
From the application's perspective, the client's socket and the server's connection socket are directly connected by a pipe. As shown in Figure 2.29, the client process can send arbitrary bytes into its socket, and TCP guarantees that the server process will receive (through the connection socket) each byte in the order sent.

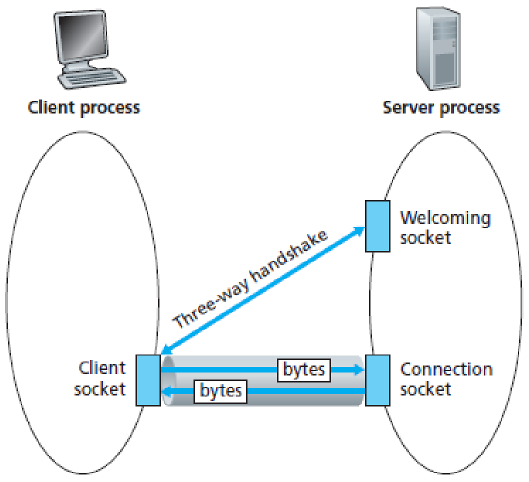


Figure 2.29 ♦ The `TCPServer` process has two sockets

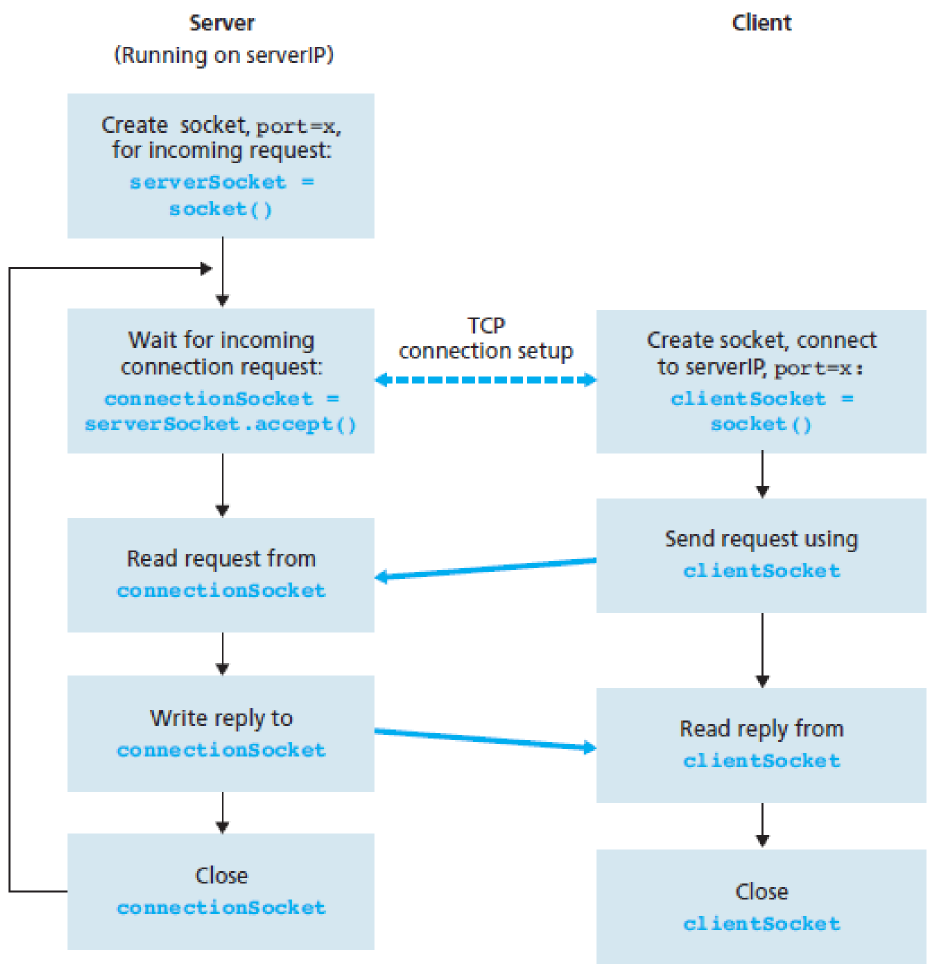

Server
(Running on serverIP)
Create socket, port=x, for incoming request:
serverSocket = socket()
Wait for incoming connection request:
connectionSocket = serverSocket.accept()
Read request from connectionSocket
Write reply to connectionSocket
Close connectionSocket
Client
TCP connection setup
Create socket, connect to serverIP, port=x:
clientSocket = socket()
Send request using clientSocket
Read reply from clientSocket
Close clientSocket

Figure 2.30 ♦ The client-server application using TCP

First line is the creation of the client socket.
clientSocket = socket(AF_INET, SOCK_STREAM)
The first parameter again indicates that the underlying network is using IPv4. The second parameter indicates that the socket is of type SOCK_STREAM, which means it is a TCP socket.

clientSocket.connect((serverName,serverPort))

Recall that before the client can send data to the server (or vice versa) using a TCP socket, a TCP connection must first be established between the client and server.
The above line initiates the TCP connection between the client and server. The parameter of the connect() method is the address of the server side of the connection.
After this line of code is executed, the three-way handshake is performed and a TCP connection is established between the client and server.

Module – 2

Transport Layer

Introduction and Transport-Layer Services

A transport-layer protocol provides for **logical communication** between application processes running on different hosts. By *logical communication*, we mean that from an application's perspective, it is as if the hosts running the processes were directly connected; in reality, the hosts may be on opposite sides of the planet, connected via numerous routers and a wide range of link types. Application processes use the logical communication provided by the transport layer to send messages to each other, free from the worry of the details of the physical infrastructure used to carry these messages.

Relationship between Transport and Network Layers

Recall that the transport layer lies just above the network layer in the protocol stack. Whereas a transport-layer protocol provides logical communication between *processes* running on different hosts, a network-layer protocol provides logical communication between *hosts*. This distinction is subtle but important. Let's examine this distinction with the aid of a household analogy. Consider two houses, one on the East Coast and the other on the West Coast, with each house being home to a dozen kids. The kids in the East Coast household are cousins of the kids in the West Coast household. The kids in the two households love to write to each other—each kid writes each cousin every week, with each letter delivered by the traditional postal service in a separate envelope. Thus, each household sends 144 letters to the other household every week. (These kids would save a lot of money if they had e-mail!) In each of the households there is one kid—Ann in the West Coast house and Bill in the East Coast house—responsible for mail collection and mail distribution. Each week Ann visits all her brothers and sisters, collects the mail, and gives the mail to a postal-service mail carrier, who makes daily visits to the house. When letters arrive at the West Coast house, Ann also has the job of distributing the mail to her brothers and sisters. Bill has a similar job on the East Coast.

In this example, the postal service provides logical communication between the two houses—the postal service moves mail from house to house, not from person to person. On the other hand, Ann and Bill provide logical communication among the cousins—Ann and Bill pick up mail from, and deliver mail to, their brothers and sisters. Note that from the cousins' perspective, Ann and Bill *are* the mail service, even though

Ann and Bill are only a part (the end-system part) of the end-to-end delivery process. This household example serves as a nice analogy for explaining how the
transport layer relates to the network layer:
application messages = letters in envelopes processes = cousins
hosts (also called end systems) = houses
transport-layer protocol = Ann and Bill
network-layer protocol = postal service (including mail carriers)

Overview of the Transport Layer in the Internet

Recall that the Internet, and more generally a TCP/IP network, makes two distinct transport-layer protocols available to the application layer. One of these protocols is **UDP** (User Datagram Protocol), which provides an unreliable, connectionless service to the invoking application. The second of these protocols is **TCP** (Transmission Control Protocol), which provides a reliable, connection-oriented service to the invoking application. When designing a network application, the application developer must specify one of these two transport protocols. As we saw in Section 2.7, the application developer selects between UDP and TCP when creating sockets.

To simplify terminology, when in an Internet context, we refer to the transport layer packet as a *segment*. We mention, however, that the Internet literature (for example, the RFCs) also refers to the transport-layer packet for TCP as a segment but often refers to the packet for UDP as a datagram. But this same Internet literature also uses the term *datagram* for the network-layer packet! For an introductory book on computer networking such as this, we believe that it is less confusing to refer to both TCP and UDP packets as segments, and reserve the term *datagram* for the network-layer packet. Before proceeding with our brief introduction of UDP and TCP, it will be useful to say a few words about the Internet's network layer.

The IP service model is a **best-effort delivery service**. This means that IP makes its "best effort" to deliver segments between communicating hosts, *but it makes noguarantees.* In particular, it does not guarantee segment delivery, it does not guarantee orderly delivery of segments, and it does not guarantee the integrity of the datain the segments. For these reasons, IP is said to be an **unreliable service**.

Extending host-to-host delivery to process-to-process delivery is called **transport-layer multiplexing** and **demultiplexing**.

.

Multiplexing and Demultiplexing

In this section, we discuss transport-layer multiplexing and demultiplexing, that is, extending the host-to-host delivery service provided by the network layer to a process-to-process delivery service for applications running on the hosts. In order to keep the discussion concrete, we'll discuss this basic transport-layer service in the context of the Internet. We emphasize, however, that a multiplexing/demultiplexing service is needed for all computer networks. At the destination host, the transport layer receives segments from the network layer just below. The transport layer has the responsibility of delivering the data in these segments to the appropriate application process running in the host. Let's take a look at an example. Suppose you are sitting in front of your computer, and you are downloading Web pages while running one FTP session and two Telnet sessions.

Now let's consider how a receiving host directs an incoming transport-layer segment to the appropriate socket. Each transport-layer segment has a set of fields in the segment for this purpose. At the receiving end, the transport layer examines these fields to identify the receiving socket and then directs the segment to that socket. This job of delivering the data in a transport-layer segment to the correct socket is called **demultiplexing**. The job of gathering data chunks at the source host from different sockets, encapsulating each data chunk with header information (that will later be
used in demultiplexing) to create segments, and passing the segments to the network layer is called **multiplexing**. Note that the transport layer in the middle host in Figure 2.2 must demultiplex segments arriving from the network layer below to either process P1 or P2 above; this is done by directing the arriving segment's data to the corresponding process's socket. The transport layer in the middle host must also gather outgoing data from these sockets, form transport-layer segments, and pass these segments down to the network layer.

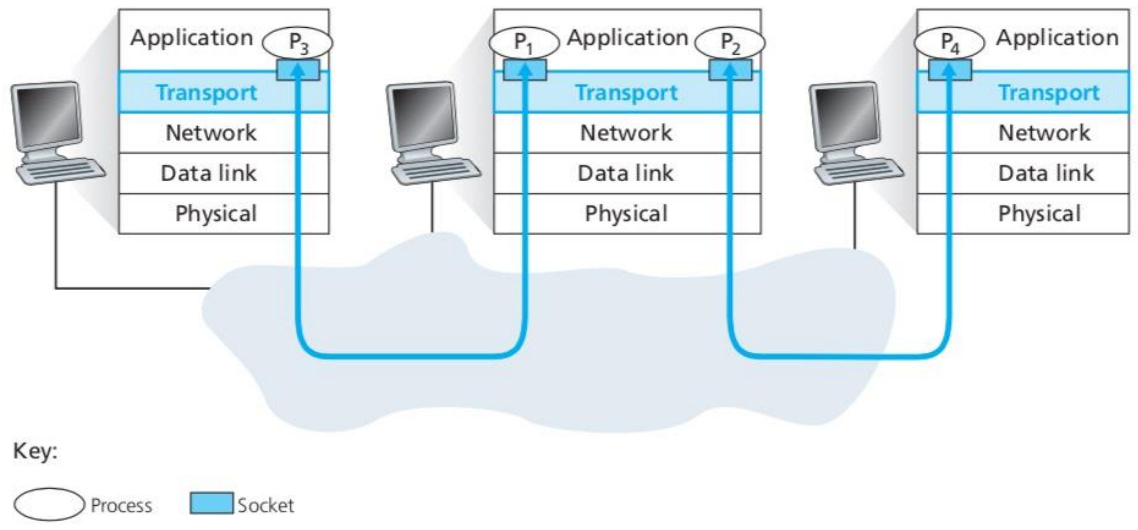


Figure 2.1 Transport-layer multiplexing and demultiplexing

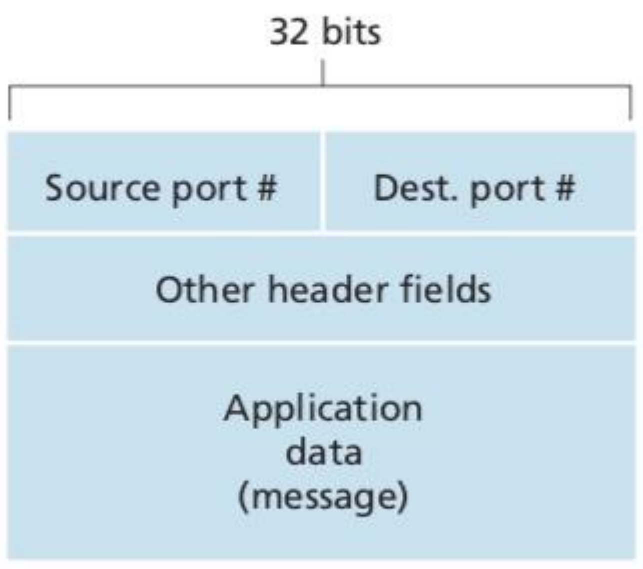


Figure 2.3 Source and destination port-number fields in a transport-layer
Segment

Connectionless Multiplexing and Demultiplexing

Recall from Section that the Python program running in a host can create a UDP socket with the line

```
clientSocket = socket(socket.AF_INET, socket.SOCK_DGRAM)
```

When a UDP socket is created in this manner, the transport layer automatically assigns a port number to the socket. In particular, the transport layer assigns a port number in the range 1024 to 65535 that is currently not being used by any other UDP port in the host. Alternatively, we can add a line into our Python program after we create the socket to associate a specific port number (say, 19157) to this UDP socket via the socket bind() method:

```
clientSocket.bind((‘’, 19157))
```

If the application developer writing the code were implementing the server side of a “well-known protocol,” then the developer would have to assign the corresponding well-known port number. Typically, the client side of the application lets the transport layer automatically (and transparently) assign the port number, whereas the server side of the application assigns a specific port number.

It is important to note that a UDP socket is fully identified by a two-tuple consisting of a destination IP address and a destination port number. As a consequence, if two UDP segments have different source IP addresses and/or source port numbers, but have the same *destination* IP address and *destination* port number, then the two segments will be directed to the same destination process via the same destination socket.

Connection-Oriented Multiplexing and Demultiplexing

In order to understand TCP demultiplexing, we have to take a close look at TCP sockets and TCP connection establishment. One subtle difference between a TCP socket and a UDP socket is that a TCP socket is identified by a four-tuple: (source IP address, source port number, destination IP address, destination port number). Thus, when a TCP segment arrives from the network to a host, the host uses all four values to direct (demultiplex) the segment to the appropriate socket. In particular, and in contrast with UDP, two arriving TCP segments with different source IP addresses or source port numbers will (with the exception of a TCP segment carrying the original connection-establishment request) be directed to two different sockets.

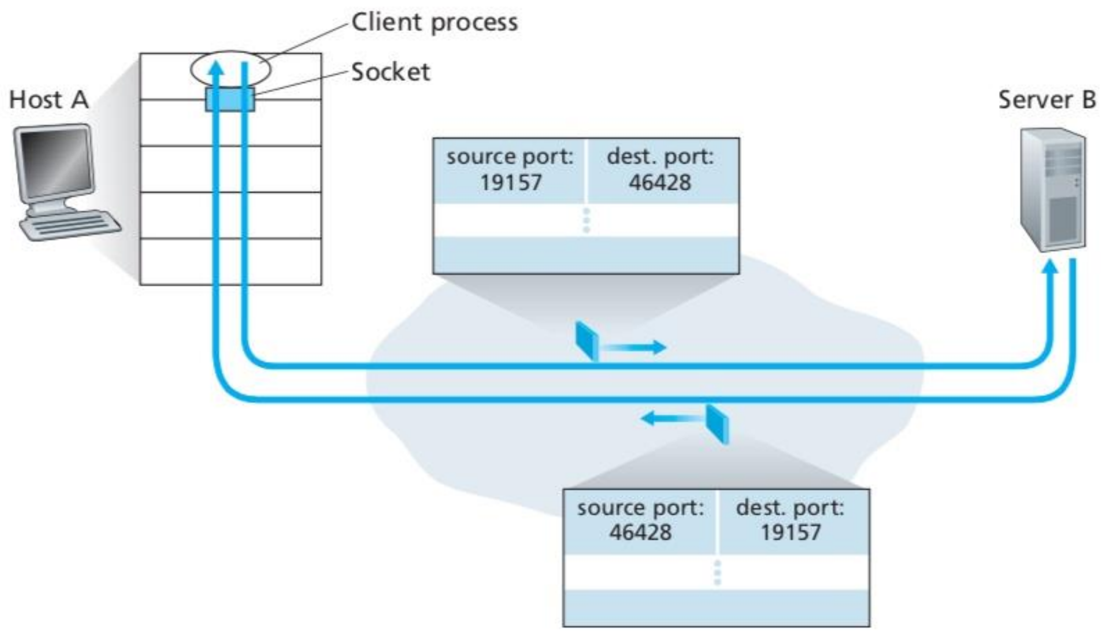


Figure 2.4 The inversion of source and destination port numbers

• The TCP server application has a "welcoming socket," that waits for connectionestablishment requests from TCP clients on port number 12000.

• The TCP client creates a socket and sends a connection establishment request segment with the lines:

clientSocket = socket(AF_INET, SOCK_STREAM)

clientSocket.connect((serverName,12000))

• Aconnection-establishment request is nothing more than a TCP segment with destination port number 12000 and a special connection-establishment bit set in the TCP header (discussed in Section 3.5). The segment also includes a source port number that was chosen by the client.

• When the host operating system of the computer running the server process receives the incoming connection-request segment with destination port 12000, it locates the server process that is waiting to accept a connection on port number 12000. The server process then creates a new socket:

connectionSocket, addr = serverSocket.accept()

• Also, the transport layer at the server notes the following four values in the connection- request segment: (1) the source port number in the segment, (2) the IP address of the source host, (3) the destination port number in the segment, and (4) its own IP address. The newly created connection socket is identified by these four values; all subsequently arriving segments whose source port, source IP address, destination port, and destination IP address match these four values will be demultiplexed to this socket. With the TCP connection now in place, the client and server can now send data to each other.

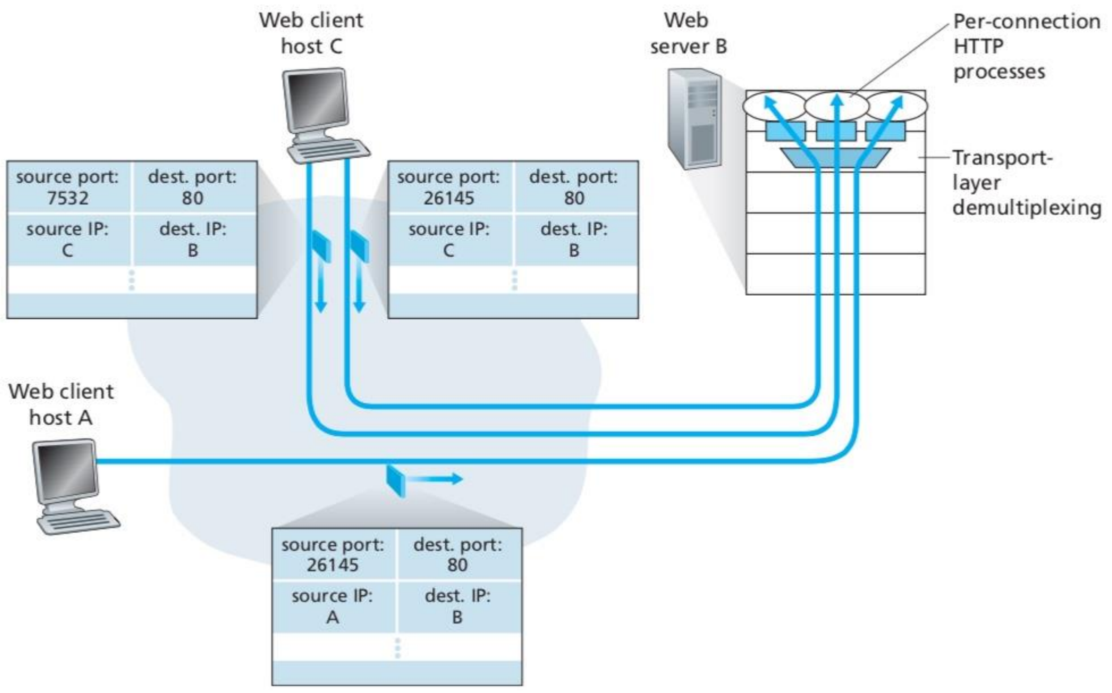


Figure 2.5 Two clients, using the same destination port number (80) to communicate with the same Web server application

Web Servers and TCP

Figure 3.5 shows a Web server that spawns a new process for each connection. As shown in Figure 3.5, each of these processes has its own connection socket through which HTTP requests arrive and HTTP responses are sent. We mention, however, that there is not always a one-to-one correspondence between connection sockets and processes. In fact, today's high-performing Web servers often use only one process, and create a new thread with a new connection socket for each new client connection. (A thread can be viewed as a lightweight subprocess.) If you did the first programming assignment in Chapter 2, you built a Web server that does just this. For such a server, at any given time there may be many connection sockets (with different identifiers) attached to the same process.

If the client and server are using persistent HTTP, then throughout the duration of the persistent connection the client and server exchange HTTP messages via the same server socket. However, if the client and server use non-persistent HTTP, then a new TCP connection is created and closed for every request/response, and hence a new socket is created and later closed for every request/response. This frequent creating and closing of sockets can severely impact the performance of a busy Web server

Connectionless Transport: UDP

UDP, defined in [RFC 768], does just about as little as a transport protocol can do. Aside from the multiplexing/demultiplexing function and some light error checking, it adds nothing to IP. In fact, if the application developer chooses UDP instead of TCP, then the application is almost directly talking with IP. UDP takes messages from the application process, attaches source and destination port number fields for the multiplexing/demultiplexing service, adds two other small fields, and passes the resulting segment to the network layer. The network layer encapsulates the transport-layer segment into an IP datagram and then makes a best-effort attempt to deliver the segment to the receiving host. If the segment arrives at the receiving host, UDP uses the destination port number to deliver the segment's data to the correct application process. Note that with UDP there is no handshaking between sending and receiving transport-layer entities before sending a segment. For this reason, UDP is said to be *connectionless.*

DNS is an example of an application-layer protocol that typically uses UDP. When the DNS application in a host wants to make a query, it constructs a DNS query message and passes the message to UDP. Without performing any handshaking with the UDP entity running on the destination end system, the host-side UDP adds header fields to the message and passes the resulting segment to the network layer. The network layer encapsulates the UDP segment into a datagram and sends the datagram to a name server. The DNS application at the querying host then waits for a reply to its query. If it doesn't receive a reply (possibly because the underlying network lost the query or the reply), either it tries sending the query to another name server, or it informs the invoking application that it can't get a reply.

Many applications are better suited for UDP for the following reasons:

• *Finer application-level control over what data is sent, and when.* Under UDP, as soon as an application process passes data to UDP, UDP will package the data inside a UDP segment and immediately pass the segment to the network layer. TCP, on the other hand, has a congestion-control mechanism that throttles the transport-layer TCP sender when one or more links between the source and destination hosts become excessively congested. TCP will also continue to resend a segment until the receipt of the segment has been acknowledged by the destination, regardless of how long reliable delivery takes. Since real-time applications often require a minimum sending rate, do not want to overly delay segment transmission, and can tolerate some data loss, TCP's service model is not particularly well matched to these applications' needs.

• *No connection establishment.* As we'll discuss later, TCP uses a three-way handshake before it starts to transfer data. UDP just blasts away without any formal preliminaries. Thus UDP does not introduce any delay to establish a connection. This is probably the principal reason why DNS runs over UDP rather than TCP—DNS would be much slower if it ran over TCP. HTTP uses TCP rather than UDP, since reliability is critical for Web pages with text. But, as we briefly discussed in Section 2.2, the TCP connection-establishment delay in HTTP is an important contributor
to the delays associated with downloading Web documents.

• *No connection state.* TCP maintains connection state in the end systems. This connection state includes receive and send buffers, congestion-control parameters, and sequence and acknowledgment number parameters. We will see in Section 3.5 that this state information is needed to implement TCP’s reliable data transfer service and to provide congestion control. UDP, on the other hand, does not maintain connection state and does not track any of these parameters. For this reason, a server devoted to a particular application can typically support many more active clients when the application runs over UDP rather than TCP.

• *Small packet header overhead.* The TCP segment has 20 bytes of header overhead in every segment, whereas UDP has only 8 bytes of overhead.

UDP Segment Structure

The UDP segment structure, shown in Figure 3.7, is defined in RFC 768. The application data occupies the data field of the UDP segment. For example, for DNS, the data field contains either a query message or a response message. For a streaming audio application, audio samples fill the data field. The UDP header has only four fields, each consisting of two bytes. As discussed in the previous section, the port numbers allow the destination host to pass the application data to the correct process running on the destination end system (that is, to perform the demultiplexing function). The length field specifies the number of bytes in the UDP segment (header plus data). An explicit length value is needed since the size of the data field may differ from one UDP segment to the next. The checksum is used by the receiving host to check whether errors have been introduced into the segment. In truth, the checksum is also calculated over a few of the fields in the IP header in addition to the UDP segment. The length field specifies the length of the UDP segment, including the header, in bytes.

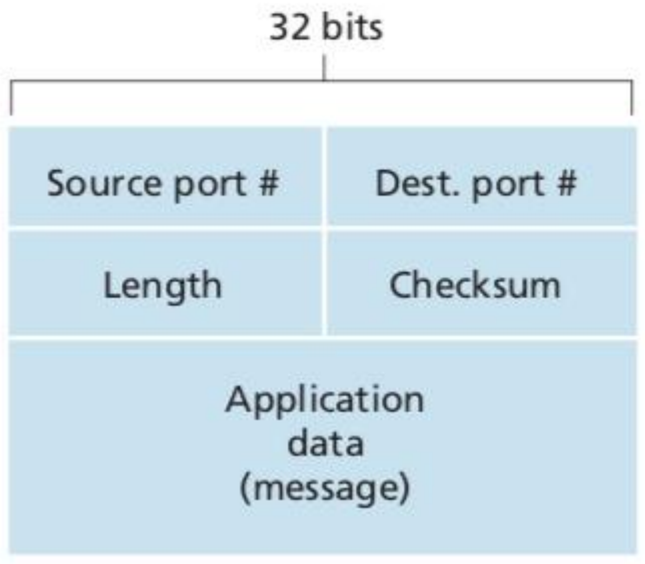


Figure 2.7 _ UDP segment structure

UDP Checksum

The UDP checksum provides for error detection. That is, the checksum is used to determine whether bits within the UDP segment have been altered (for example, by noise in the links or while stored in a router) as it moved from source to destination. UDP at the sender side performs the 1s complement of the sum of all the 16-bit words in the segment, with any overflow encountered during the sum being wrapped around. This result is put in the checksum field of the UDP segment.

Here we give a simple example of the checksum calculation. You can find details about efficient implementation of the calculation in RFC 1071 and performance over real data in [Stone 1998; Stone 2000]. As an example, suppose that we have the following three 16-bit words:

0110011001100000

0101010101010101

1000111100001100

The sum of first two of these 16-bit words is

0110011001100000

0101010101010101

1011101110110101

Adding the third word to the above sum gives

1011101110110101

1000111100001100

0100101011000010

Note that this last addition had overflow, which was wrapped around. The 1s complement is obtained by converting all the 0s to 1s and converting all the 1s to 0s. Thus the 1s complement of the sum 0100101011000010 is 1011010100111101, which becomes the checksum. At the receiver, all four 16-bit words are added, including the checksum. If no errors are introduced into the packet, then clearly the sum at the receiver will be 1111111111111111. If one of the bits is a 0, then we know that errors have been introduced into the packet.

Principles of Reliable Data Transfer

It is the responsibility of a **reliable data transfer protocol** to implement this service abstraction. This task is made difficult by the fact that the layer *below* the reliable data transfer protocol may be unreliable. For example, TCP is a reliable data transfer protocol that is implemented on top of an unreliable (IP) end-to-end network layer. More generally, the layer beneath the two reliably communicating end points might consist of a single physical link (as in the case of a link-level data transfer protocol) or a global internetwork (as in the case of a transport-level protocol). For our purposes, however, we can view this lower layer simply as an unreliable point-to-point channel.

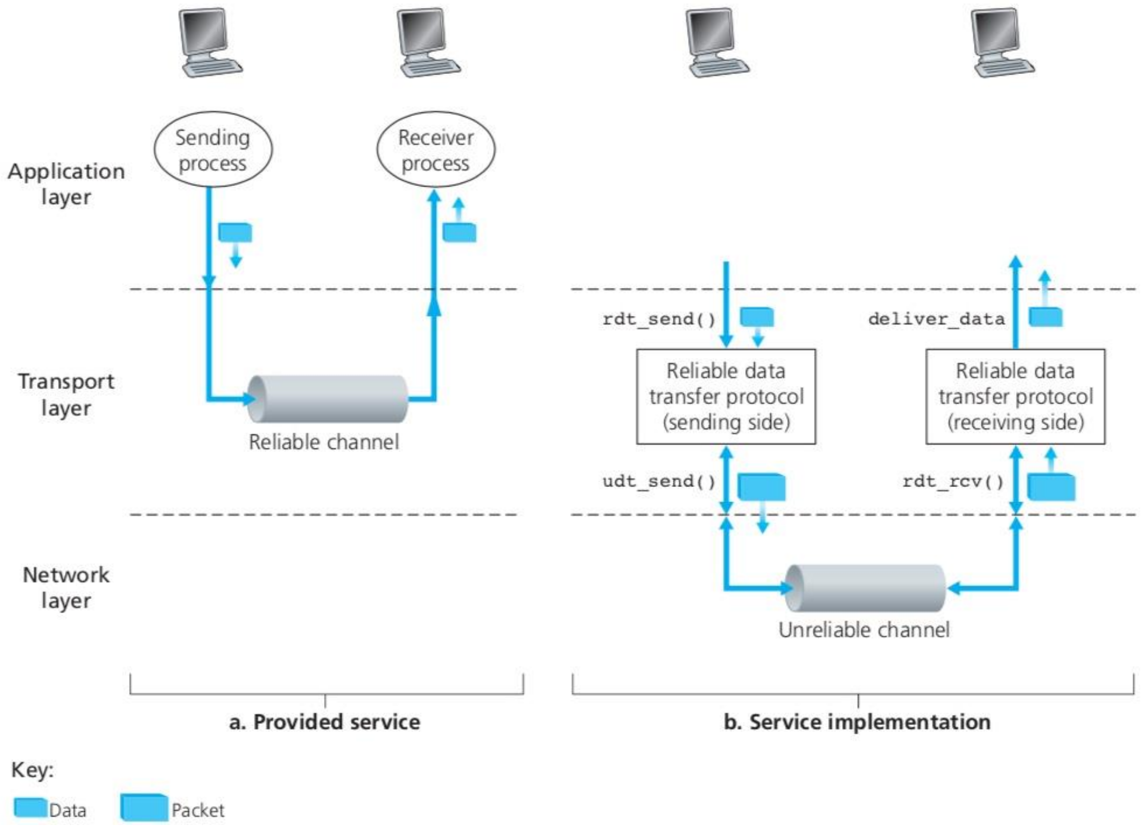


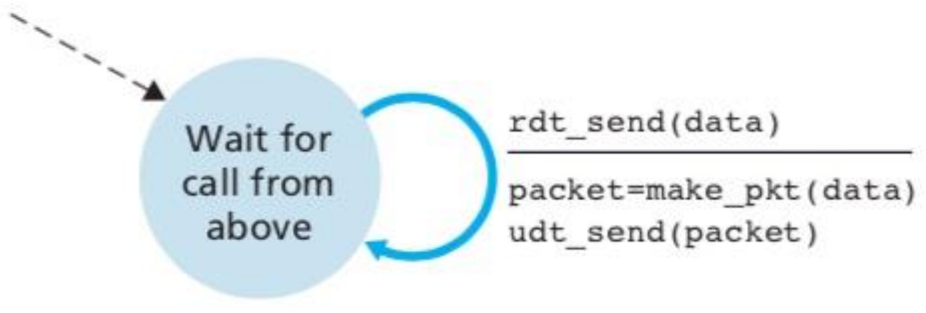
Figure 2.8 _ Reliable data transfer: Service model and service Implementation

Building a Reliable Data Transfer Protocol

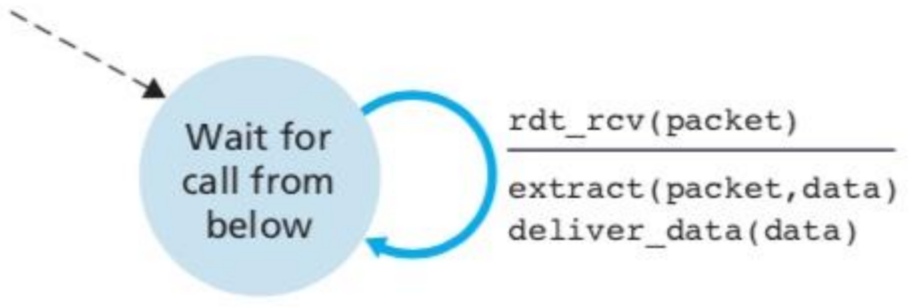
Reliable Data Transfer over a Perfectly Reliable Channel: rdt1.0

The **finite-state machine (FSM)** definitions for the rdt1.0 sender and receiver are shown in Figure 2.9. The FSM in Figure 2.9(a) defines the operation of the sender, while the FSM in Figure 2.9(b) defines the operation of the receiver. It is important to note that there are *separate* FSMs for the sender and for the receiver. The sender and receiver FSMs in Figure 2.9 each have just one state. The arrows in the

FSM description indicate the transition of the protocol from one state to another. (Since each FSM in Figure 2.9 has just one state, a transition is necessarily from the one
state back to itself; we'll see more complicated state diagrams shortly.) The event causing the transition is shown above the horizontal line labeling the transition, and the actions taken when the event occurs are shown below the horizontal line.



a. rdt1.0: sending side



b. rdt1.0: receiving side

Figure 2.9 _ rdt1.0 – A protocol for a completely reliable channel

When no action is taken on an event, or no event occurs and an action is taken, we'll use the symbol _ below or above the horizontal, respectively, to explicitly denote the lack of an action or event. The initial state of the FSM is indicated by the dashed arrow. Although the FSMs in Figure 3.9 have but one state, the FSMs we will see shortly have multiple states, so it will be important to identify the initial state of each FSM.

The sending side of rdt simply accepts data from the upper layer via the rdt_send(data) event, creates a packet containing the data (via the action make_pkt(data)) and sends the packet into the channel. In practice, the rdt_send(data) event would result from a procedure call (for example, to
rdt_send()) by the upper-layer application. On the receiving side, rdt receives a packet from the underlying channel via the rdt_rcv(packet) event, removes the data from the packet (via the action
extract (packet, data)) and passes the data up to the upper layer (via the action deliver_data(data)). In practice, the rdt_rcv(packet) event would result from a procedure call (for example, to rdt_rcv()) from the lowerlayer protocol.

Reliable Data Transfer over a Channel with Bit Errors: rdt2.0

Fundamentally, three additional protocol capabilities are required in ARQ protocols to handle the presence of bit errors:

• *Error detection.* First, a mechanism is needed to allow the receiver to detect when bit errors have occurred. Recall from the previous section that UDP uses the Internet checksum field for exactly this purpose.
• *Receiver feedback.* Since the sender and receiver are typically executing on different end systems, possibly separated by thousands of miles, the only way for the sender to learn of the receiver's view of the world
The positive (ACK) and negative (NAK) acknowledgment replies in the message-dictation scenario are examples of such feedback. Our rdt2.0 protocol will similarly send ACK and NAK packets back from the

receiver to the sender. In principle, these packets need only be one bit long; for example, a 0 value could indicate a NAK and a value of 1 could indicate an ACK.

• *Retransmission.* A packet that is received in error at the receiver will be retransmitted by the sender.

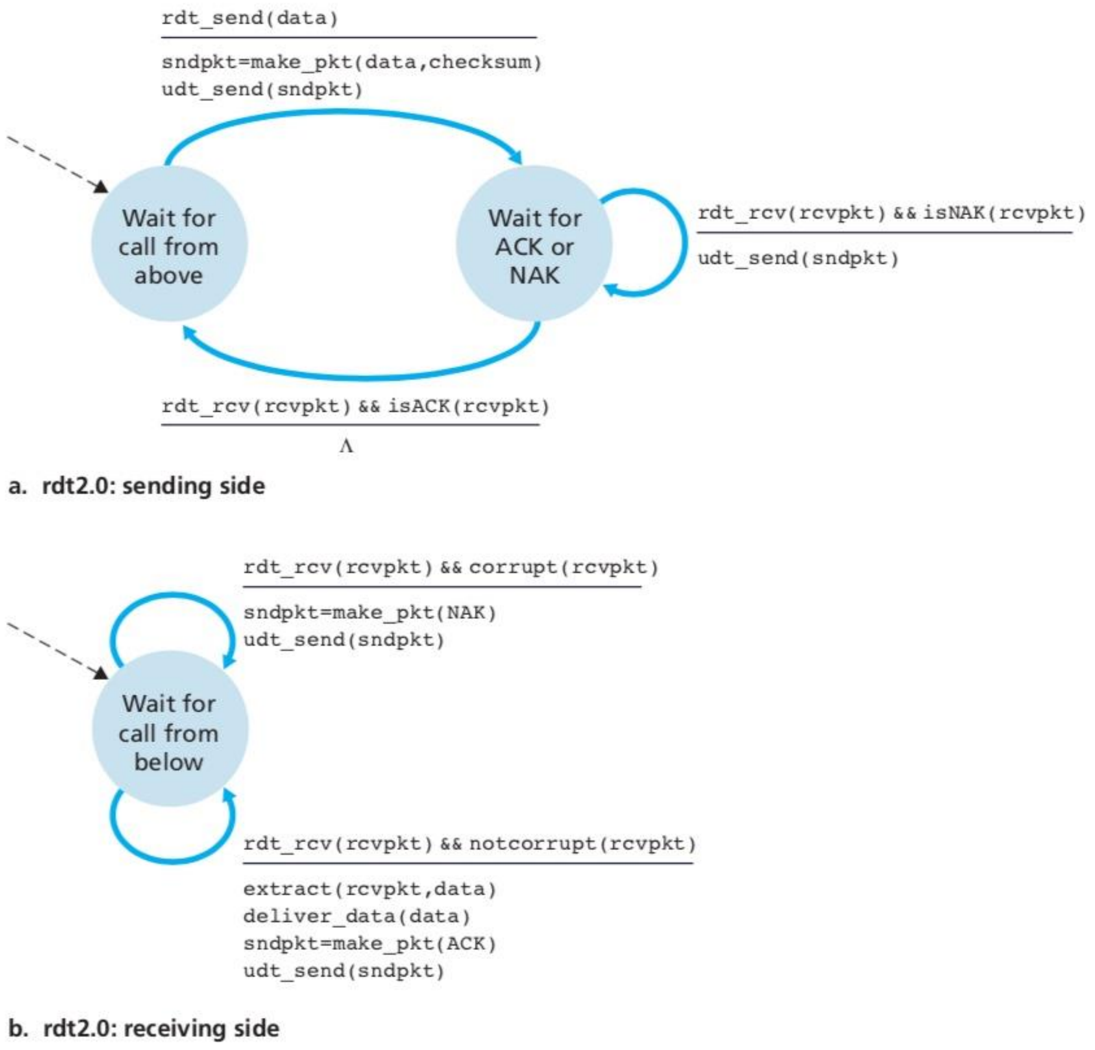


Figure 2.10 _ rdt2.0–A protocol for a channel with bit errors

Figure 2.10 shows the FSM representation of rdt2.0, a data transfer protocol employing error detection, positive acknowledgments, and negative acknowledgments. The send side of rdt2.0 has two states. In the leftmost state, the send-side protocol is waiting for data to be passed down from the upper layer. When the rdt_send(data) event occurs, the sender will create a packet (sndpkt) containing the data to be sent, along with a packet checksum and then send the packet via the udt_send(sndpkt) operation.

In the rightmost state, the sender protocol is waiting for an ACK or a NAK packet from the receiver. If an ACK packet is received (the notation rdt_rcv(rcvpkt) && isACK (rcvpkt) in Figure 2.10 corresponds to this event), the sender knows that the most recently transmitted packet has been received correctly and thus the protocol returns to the state of waiting for data from the Upper layer.

If a NAK is received, the protocol retransmits the last packet and waits for an ACK or NAK to be returned by the receiver in response to the retransmitted data packet. It is important to note that when the sender is in the wait-for-ACK-or-NAK state, it *cannot* get more data from the upper layer; that is, the rdt_send() event can not occur; that will happen only after the sender receives an ACK and leaves this state.
Thus, the sender will not send a new piece of data until it is sure that the receiver has correctly received the current packet. Because of this behavior, protocols such as rdt2.0 are known as **stop-and-wait** protocols.
The receiver-side FSM for rdt2.0 still has a single state. On packet arrival, the receiver replies with either an ACK or a NAK, depending on whether or not the received packet is corrupted. In Figure 3.10, the notation rdt_rcv(rcvpkt) && corrupt(rcvpkt) corresponds to the event in which a packet is received and is found to be in error.

Consider three possibilities for handling corrupted ACKs or NAKs:
• For the first possibility, consider what a human might do in the message dictation scenario. If the speaker didn't understand the "OK" or "Please repeat that" reply from the receiver, the speaker would probably ask, "What did you say?" The receiver would then repeat the reply. But what if the speaker's "What did you say?" is corrupted? The receiver, having no idea whether the garbled sentence was part of the dictation or a request to repeat the last reply, would probably then respond with "What did *you* say?" And then, of course, that response might be garbled.

Clearly, we're heading down a difficult path.
• A second alternative is to add enough checksum bits to allow the sender not only to detect, but also to recover from, bit errors. This solves the immediate problem for a channel that can corrupt packets but not lose them.

• A third approach is for the sender simply to resend the current data packet when it receives a garbled ACK or NAK packet. This approach, however, introduces **duplicate packets** into the sender-to-receiver channel.

The fundamental difficulty with duplicate packets is that the receiver doesn't know whether the ACK or NAK it last sent was received correctly at the sender. Thus, it cannot know *a priori* whether an arriving packet contains new data or is a retransmission!

Figures 2.11 and 2.12 show the FSM description for rdt2.1, our fixed version of rdt2.0. The rdt2.1 sender and receiver FSMs each now have twice as many states as before. This is because the protocol state must now reflect whether the packet currently being sent (by the sender) or expected (at the receiver) should have a sequence number of 0 or 1. Note that the actions in those states where a 0-numbered packet is being sent or expected are mirror images of those where a 1-numbered packet is being sent or expected; the only differences have to do with the handling of
the sequence number.

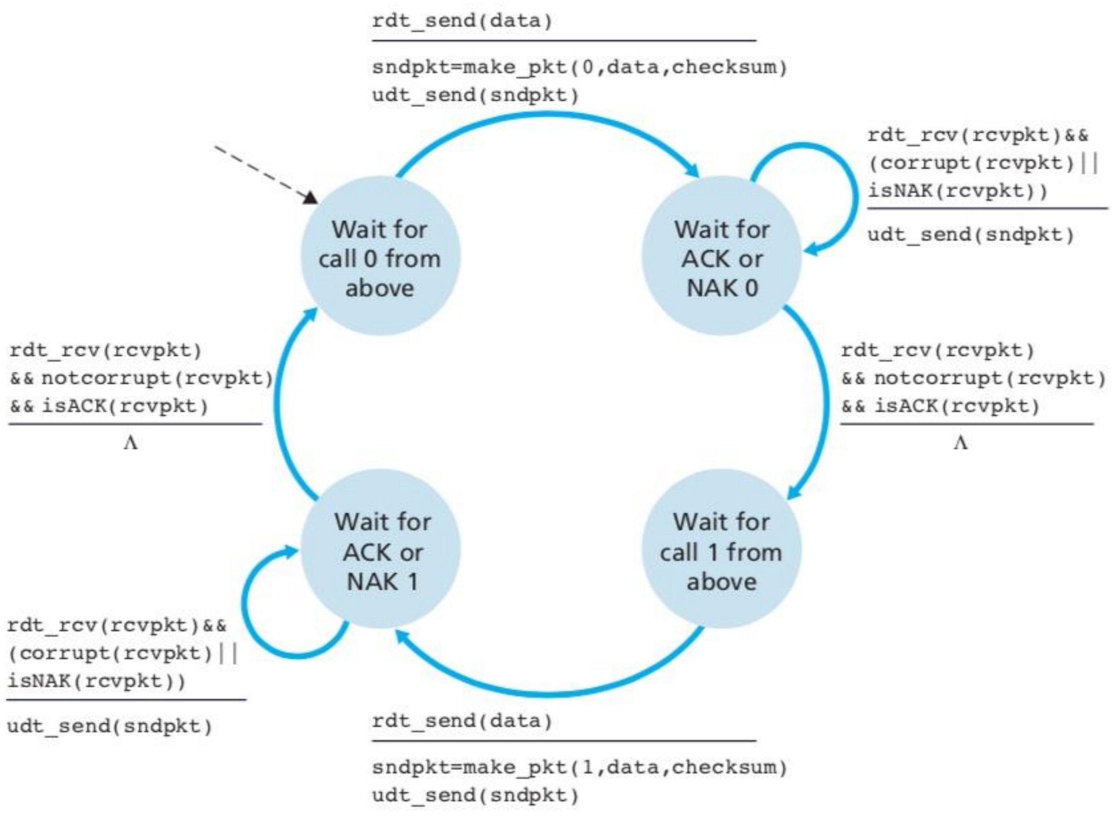

rdt_send(data)
sndpkt=make_pkt(0,data,checksum)
udt_send(sndpkt)
Wait for call 0 from above
Wait for ACK or NAK 0
rdt_rcv(rcvpkt)&&(corrupt(rcvpkt)||isNAK(rcvpkt))
udt_send(sndpkt)
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)
Λ
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)
Λ
Wait for ACK or NAK 1
Wait for call 1 from above
rdt_rcv(rcvpkt)&&(corrupt(rcvpkt)||isNAK(rcvpkt))
udt_send(sndpkt)
rdt_send(data)
sndpkt=make_pkt(1,data,checksum)
udt_send(sndpkt)

Figure 2.11 _ rdt2.1 sender

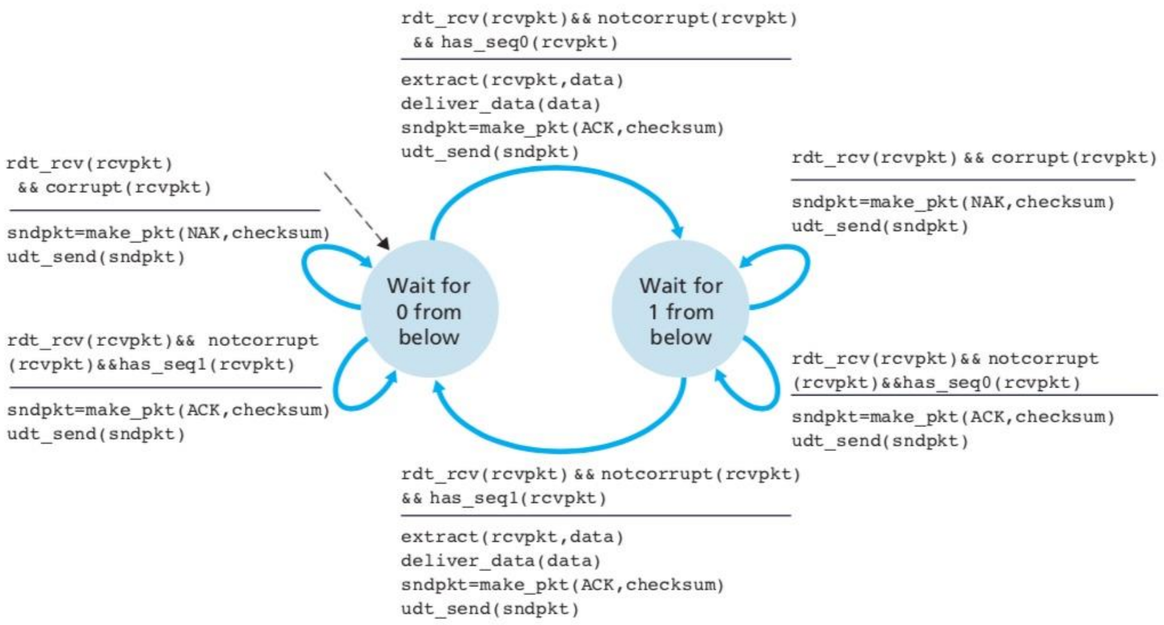

rdt_rcv(rcvpkt)&& notcorrupt(rcvpkt) && has_seq0(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pkt(ACK,checksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && corrupt(rcvpkt)
sndpkt=make_pkt(NAK,checksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && corrupt(rcvpkt)
sndpkt=make_pkt(NAK,checksum)
udt_send(sndpkt)
Wait for 0 from below
Wait for 1 from below
rdt_rcv(rcvpkt)&& notcorrupt (rcvpkt)&&has_seq1(rcvpkt)
sndpkt=make_pkt(ACK,checksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt)&& notcorrupt (rcvpkt)&&has_seq0(rcvpkt)
sndpkt=make_pkt(ACK,checksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq1(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pkt(ACK,checksum)
udt_send(sndpkt)

Figure 2.12 _ rdt2.1 receiver

Reliable Data Transfer over a Lossy Channel with Bit Errors: rdt3.0

Two additional concerns must now be addressed by the protocol: how to detect packet loss and what to do when packet loss occurs. The use of checksumming, sequence numbers, ACK packets, and retransmissions—the techniques already developed in rdt2.2—will allow us to answer the latter concern. Handling the first concern will require adding a new protocol mechanism. There are many possible approaches toward dealing with packet loss.

In all cases, the action is the same: retransmit. Implementing a time-based retransmission mechanism requires a **countdown timer** that can interrupt the sender after a given amount of time has expired. The sender will thus need to be able to (1) start the timer each time a packet (either a first-time packet or a retransmission) is sent, (2) respond to a timer interrupt (taking appropriate
actions), and (3) stop the timer.

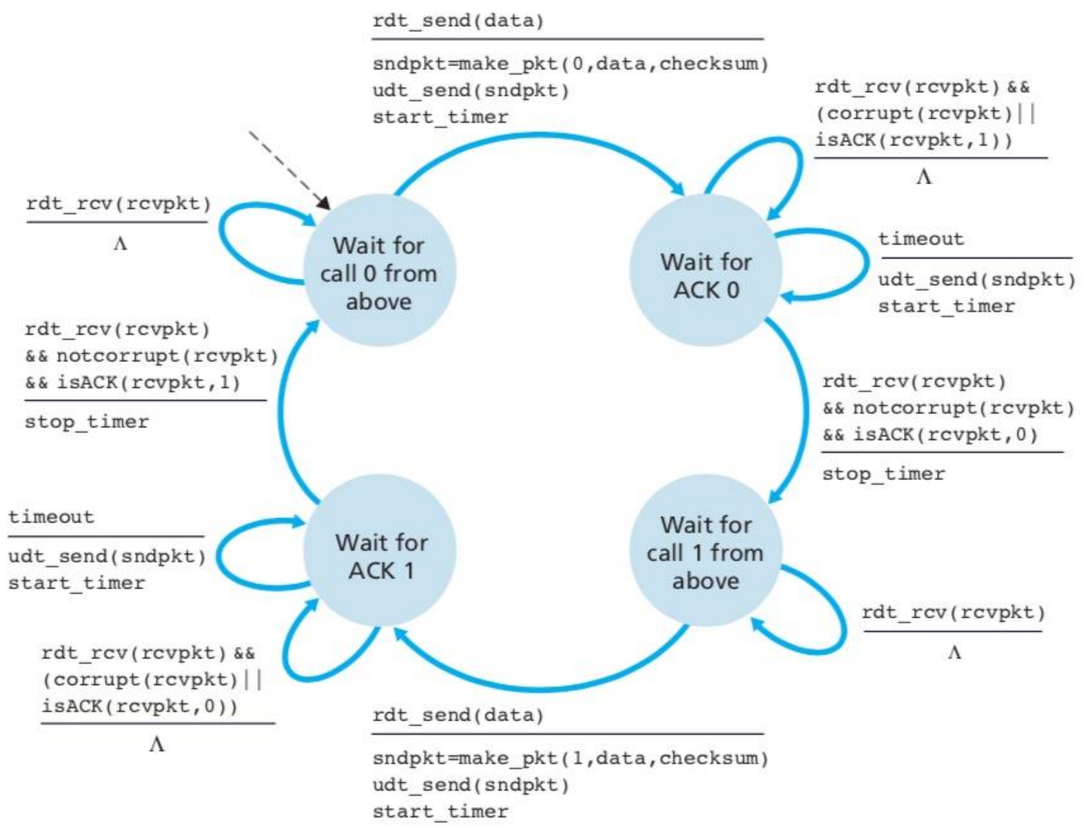


Figure 2.15 _ rdt3.0 sender

Figure 2.15 shows the sender FSM for rdt3.0, a protocol that reliably transfers data over a channel that can corrupt or lose packets; in the homework problems, you'll be asked to provide the receiver FSM for rdt3.0. Figure 2.16 shows how the protocol operates with no lost or delayed packets and how it handles lost data packets. In Figure 2.16, time moves forward from the top of the diagram toward the bottom of the diagram; note that a receive time for a packet is necessarily later than the send time for a packet as a result of transmission and propagation delays. In Figures 2.16(b)–(d),
the send-side brackets indicate the times at which a timer is set and later times out.

Several of the more subtle aspects of this protocol are explored in the exercises at the end of this chapter. Because packet sequence numbers alternate between 0 and 1, protocol rdt3.0 is sometimes known as the **alternating-bit protocol**.

Pipelined Reliable Data Transfer Protocols

Protocol rdt3.0 is a functionally correct protocol, but it is unlikely that anyone would be happy with its performance, particularly in today's high-speed networks. At the heart of rdt3.0's performance problem is the fact that it is a stop-and-wait protocol. To appreciate the performance impact of this stop-and-wait behavior, consider an idealized case of two hosts, one located on the West Coast of the United States and the other located on the East Coast, as shown in Figure 2.17. The speed-of-light round-trip propagation delay between these two end systems, RTT, is approximately 30 milliseconds. Suppose that they are connected by a channel with a transmission

rate, *R*, of 1 Gbps (109 bits per second). With a packet size, *L*, of 1,000 bytes (8,000 bits) per packet, including both header fields and data, the time needed to actually transmit the packet into the 1 Gbps link is *dtrans*

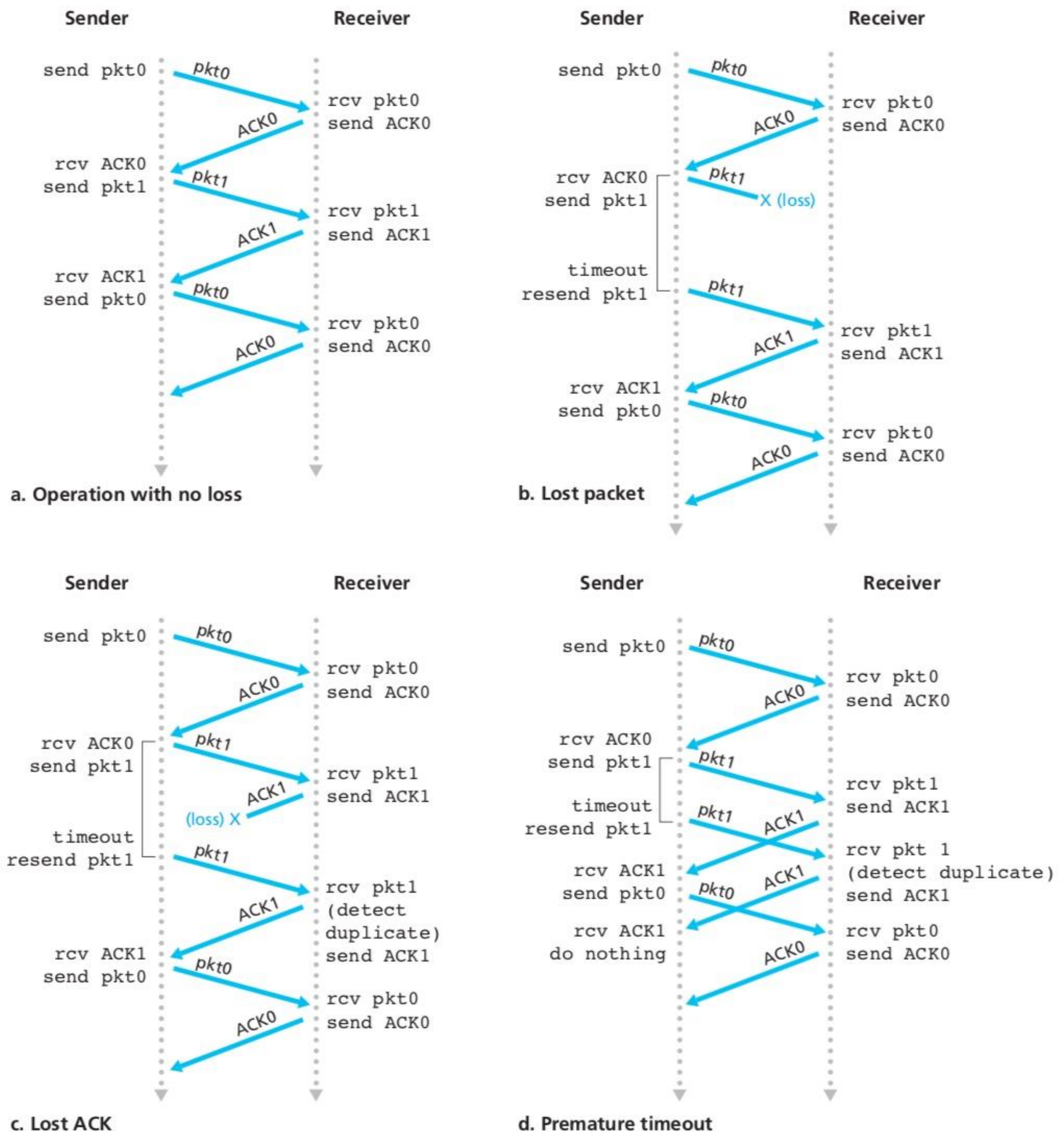


Figure 2.16 _ Operation of rdt3.0, the alternating-bit protocol

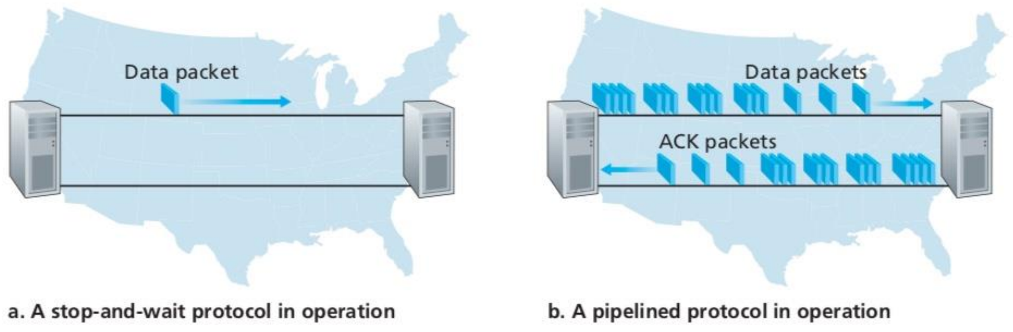

Data packet
a. A stop-and-wait protocol in operation
Data packets
ACK packets
b. A pipelined protocol in operation

Figure 2.17 _ Stop-and-wait versus pipelined protocol

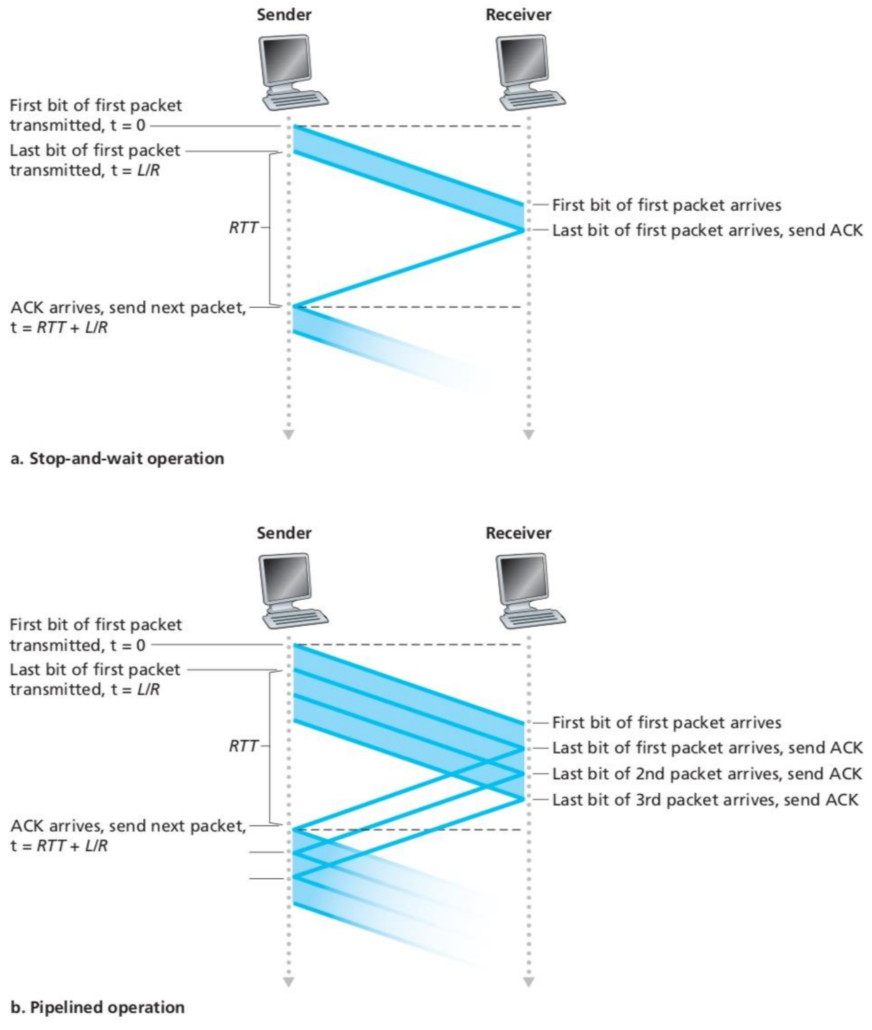

Sender
Receiver
First bit of first packet transmitted, t = 0
Last bit of first packet transmitted, t = L/R
RTT
First bit of first packet arrives
Last bit of first packet arrives, send ACK
ACK arrives, send next packet, t = RTT + L/R
a. Stop-and-wait operation
Sender
Receiver
First bit of first packet transmitted, t = 0
Last bit of first packet transmitted, t = L/R
RTT
First bit of first packet arrives
Last bit of first packet arrives, send ACK
Last bit of 2nd packet arrives, send ACK
Last bit of 3rd packet arrives, send ACK
ACK arrives, send next packet, t = RTT + L/R
b. Pipelined operation

Figure 2.18 _ Stop-and-wait and pipelined sending

Figure 2.18(a) shows that with our stop-and-wait protocol, if the sender begins sending the packet at $t = 0$, then at $t = L/R = 8$ microseconds, the last bit enters the channel at the sender side. The packet then makes its 15-msec cross-country journey, with the last bit of the packet emerging at the receiver at $t = \text{RTT}/2 + L/R = 15.008$ msec.

The solution to this particular performance problem is simple: Rather than operate in a stop-and-wait manner, the sender is allowed to send multiple packets without waiting for acknowledgments, as illustrated in Figure 3.17(b). Figure 3.18(b) shows that if the sender is allowed to transmit three packets before having to wait for acknowledgments, the utilization of the sender is essentially tripled. Since the many in-transit sender-to-receiver packets can be visualized as filling a pipeline, this technique is known as **pipelining**.

Pipelining has the following consequences for reliable data transfer protocols:

• The range of sequence numbers must be increased, since each in-transit packet (not counting retransmissions) must have a unique sequence number and there may be multiple, in-transit, unacknowledged packets.

• The sender and receiver sides of the protocols may have to buffer more than one packet. Minimally, the sender will have to buffer packets that have been transmitted but not yet acknowledged. Buffering of correctly received packets may also be needed at the receiver, as discussed below.

• The range of sequence numbers needed and the buffering requirements will depend on the manner in which a data transfer protocol responds to lost, corrupted, and overly delayed packets. Two basic approaches toward pipelined error recovery can be identified: **Go-Back-N** and **selective repeat**.

Go-Back-N (GBN)

In a **Go-Back-N (GBN) protocol**, the sender is allowed to transmit multiple packets without waiting for an acknowledgment, but is constrained to have no more than some maximum allowable number, *N*, of unacknowledged packets in the pipeline.

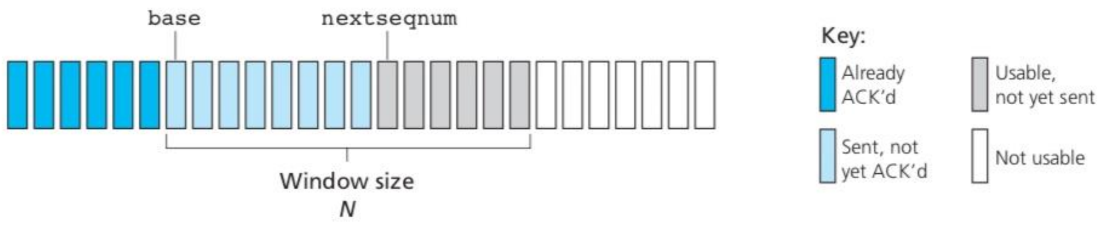
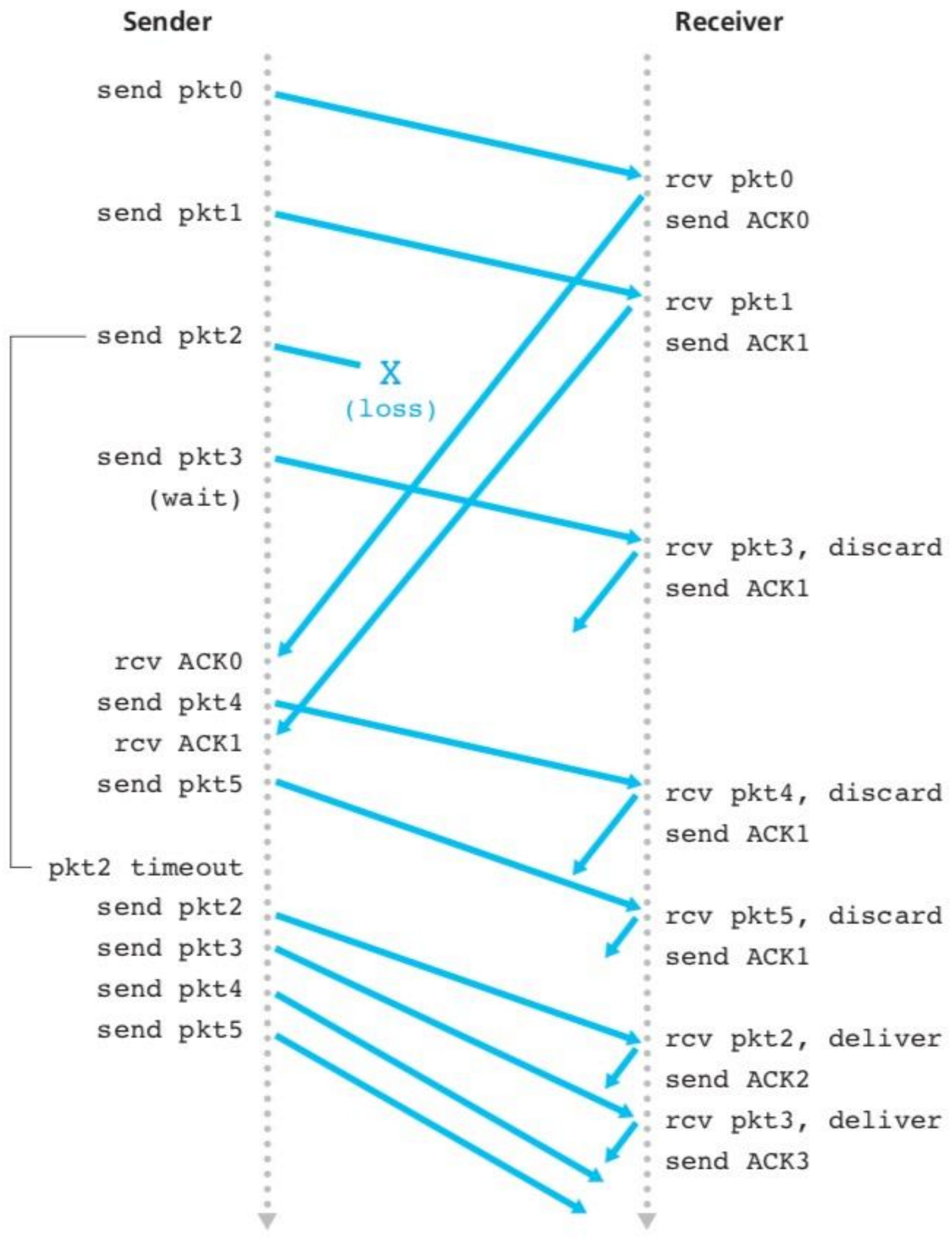


Figure 2.19 _ Sender's view of sequence numbers in Go-Back-N

Figure 2.19 shows the sender's view of the range of sequence numbers in a GBN protocol. If we define base to be the sequence number of the oldest unacknowledged packet and nextseqnum to be the smallest unused sequence number then four intervals in the range of sequence numbers can be identified. Sequence numbers in the interval [0,base-1] correspond to packets that have already been transmitted and acknowledged. The interval [base,nextseqnum-1] corresponds to packets that have been sent but not yet acknowledged. Sequence numbers in the interval [nextseqnum,base+N-1] can be used for packets that can be sent immediately, should data arrive from the upper layer.

Finally, sequence numbers greater than or equal to base+N cannot be used until an unacknowledged packet currently in the pipeline (specifically, the packet with sequence number base) has been acknowledged. As suggested by Figure 2.19, the range of permissible sequence numbers for transmitted but not yet acknowledged packets can be viewed as a window of size *N*
over the range of sequence numbers. As the protocol operates, this window slides forward over the sequence number space. For this reason, *N* is often referred to as the **window size** and the GBN protocol itself as a **sliding-window protocol**.


Sender
Receiver
send pkt0
rcv pkt0
send ACK0
send pkt1
rcv pkt1
send ACK1
send pkt2
X
(loss)
send pkt3
(wait)
rcv pkt3, discard
send ACK1
rcv ACK0
send pkt4
rcv ACK1
send pkt5
rcv pkt4, discard
send ACK1
pkt2 timeout
rcv pkt5, discard
send ACK1
send pkt2
send pkt3
send pkt4
send pkt5
rcv pkt2, deliver
send ACK2
rcv pkt3, deliver
send ACK3

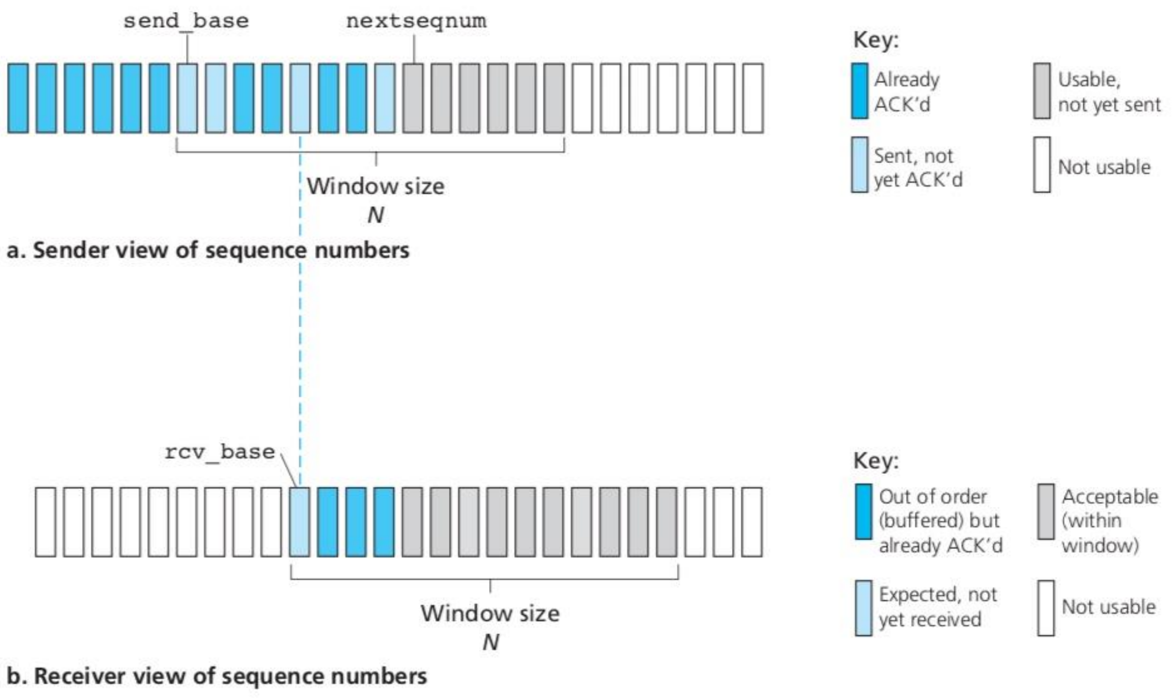

send_base
nextseqnum
Window size
N
a. Sender view of sequence numbers
Key:
Already ACK'd
Usable, not yet sent
Sent, not yet ACK'd
Not usable
rcv_base
Window size
N
b. Receiver view of sequence numbers
Key:
Out of order (buffered) but already ACK'd
Acceptable (within window)
Expected, not yet received
Not usable

Figure 2.20 _ Extended FSM description of GBN sender

The GBN sender must respond to three types of events:

• *Invocation from above.* When rdt_send() is called from above, the sender first checks to see if the window is full, that is, whether there are *N* outstanding, unacknowledged packets. If the window is not full, a packet is created and sent, and variables are appropriately updated. If the window is full, the sender simply returns the data back to the upper layer, an implicit indication that the window is full. The upper layer would presumably then have to try again later. In a real
implementation, the sender would more likely have either buffered (but not immediately sent) this data, or would have a synchronization mechanism (for example, a semaphore or a flag) that would allow the upper layer to call rdt_send() only when the window is not full.

• *Receipt of an ACK.* In our GBN protocol, an acknowledgment for a packet with sequence number *n* will be taken to be a **cumulative acknowledgment**, indicating that all packets with a sequence number up to and including *n* have been correctly received at the receiver. We'll come back to this issue shortly when we examine the receiver side of GBN.

• *A timeout event.* The protocol's name, "Go-Back-N," is derived from the sender's behavior in the presence of lost or overly delayed packets. As in the stop-and-wait protocol, a timer will again be used to recover from lost data or acknowledgment packets. If a timeout occurs, the sender resends *all* packets that have been previously sent but that have not yet been acknowledged. Our sender in Figure 2.20 uses only a single timer, which can be thought of as a timer for the oldest transmitted but not yet acknowledged packet. If an ACK is received but there are still additional
transmitted but not yet acknowledged packets, the timer is restarted. If there are no outstanding, unacknowledged packets, the timer is stopped.

Selective Repeat (SR)

A single packet error can thus cause GBN to retransmit a large number of packets, many unnecessarily. As the probability of channel errors increases, the pipeline can become filled withthese unnecessary retransmissions. Imagine, in our message-dictation scenario, that if every time a word was garbled, the surrounding 1,000 words (for example, a window size of 1,000 words) had to be repeated. The dictation would be slowed by all of the reiterated words. As the name suggests, selective-repeat protocols avoid unnecessary retransmissions by having the sender retransmit only those packets that it suspects were received in error (that is, were lost or corrupted) at the receiver. This individual, as needed, retransmission will require that the receiver *individually* acknowledge correctly received packets. A window size of *N* will again be used to limit the numberof outstanding, unacknowledged packets in the pipeline. However, unlike GBN, the
sender will have already received ACKs for some of the packets in the window.

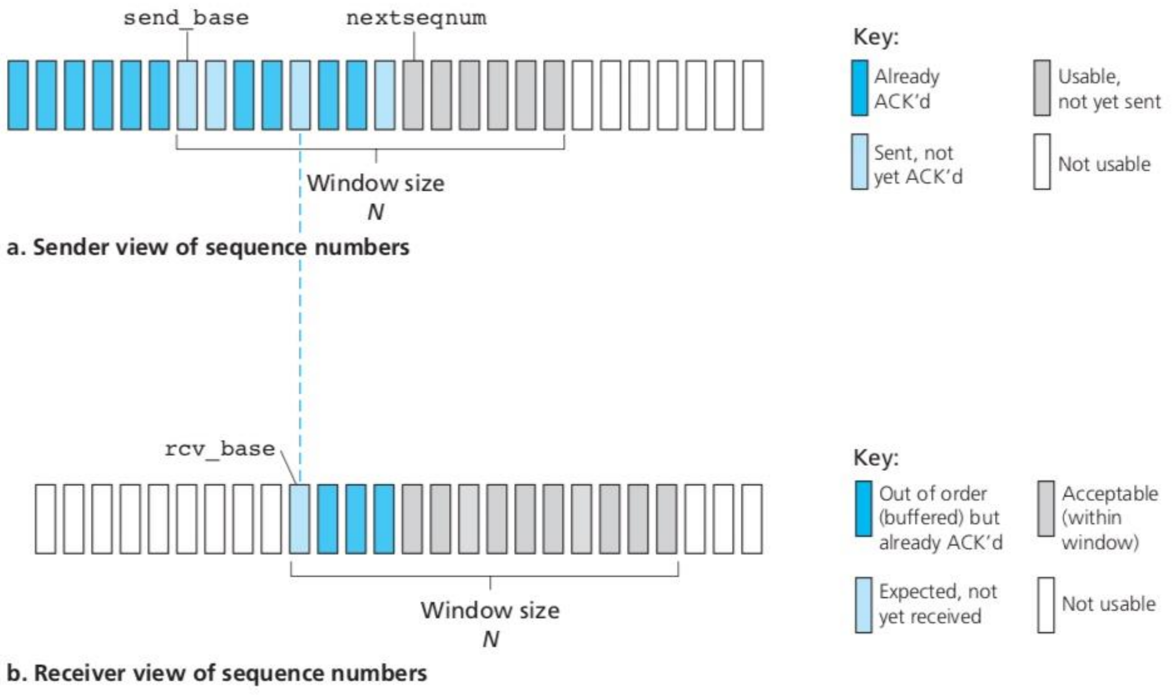

send_base
nextseqnum
Window size
N
a. Sender view of sequence numbers
Key:
Already ACK'd
Usable, not yet sent
Sent, not yet ACK'd
Not usable
rcv_base
Window size
N
b. Receiver view of sequence numbers
Key:
Out of order (buffered) but already ACK'd
Acceptable (within window)
Expected, not yet received
Not usable

Figure 2.23 _ Selective-repeat (SR) sender and receiver views of sequence-number space

Figure 3.23 shows the SR sender's view of the sequence number space. Figure 2.24 details the various actions taken by the SR sender. The SR receiver will acknowledge a correctly received packet whether or not it is in order. Out-of-order packets are buffered until any missing packets (that is, packets with lower sequence numbers) are received, at which point a batch of packets can be delivered in order to the upper layer. Figure 2.25 itemizes the various actions taken by the SR receiver. Figure 2.26 shows an example of SR operation in the presence of lost packets. Note that in Figure 2.26, the receiver initially buffers packets 3, 4, and 5, and delivers them together with packet 2 to the upper layer when packet 2 is finally received.

It is important to note that in Step 2 in Figure 2.25, the receiver reacknowledges (rather than ignores) already received packets with certain sequence numbers *below* the current window base. You should convince yourself that this reacknowledgment is indeed needed. Given the sender and receiver sequence number spaces in Figure 2.23, for example, if there is no ACK for packet send_base propagating from the receiver to the sender, the sender will eventually retransmit packet send_base, even though it is clear (to us, not the sender!) that the receiver has already received

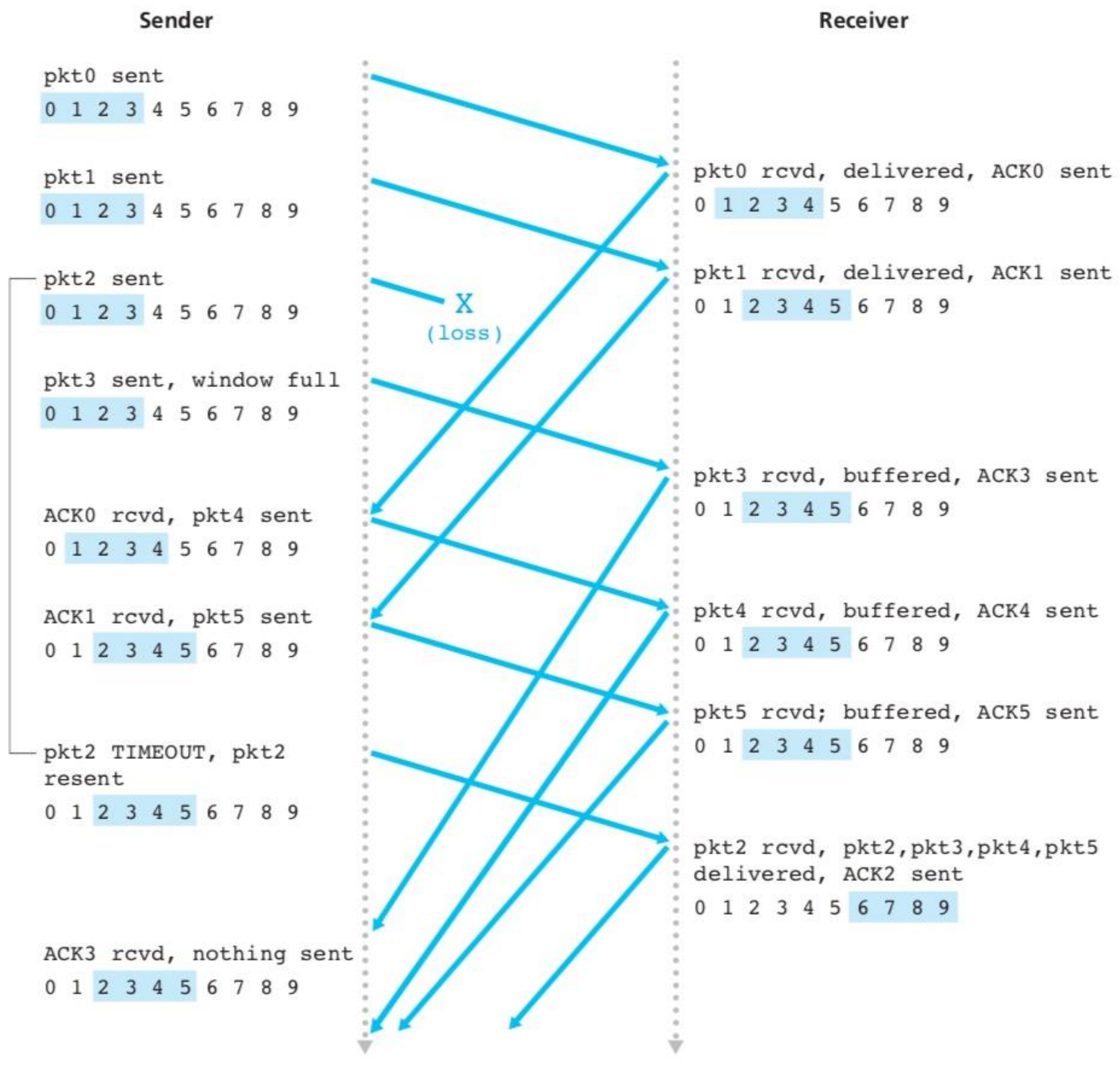


Figure 2.22 _ Go-Back-N in operation

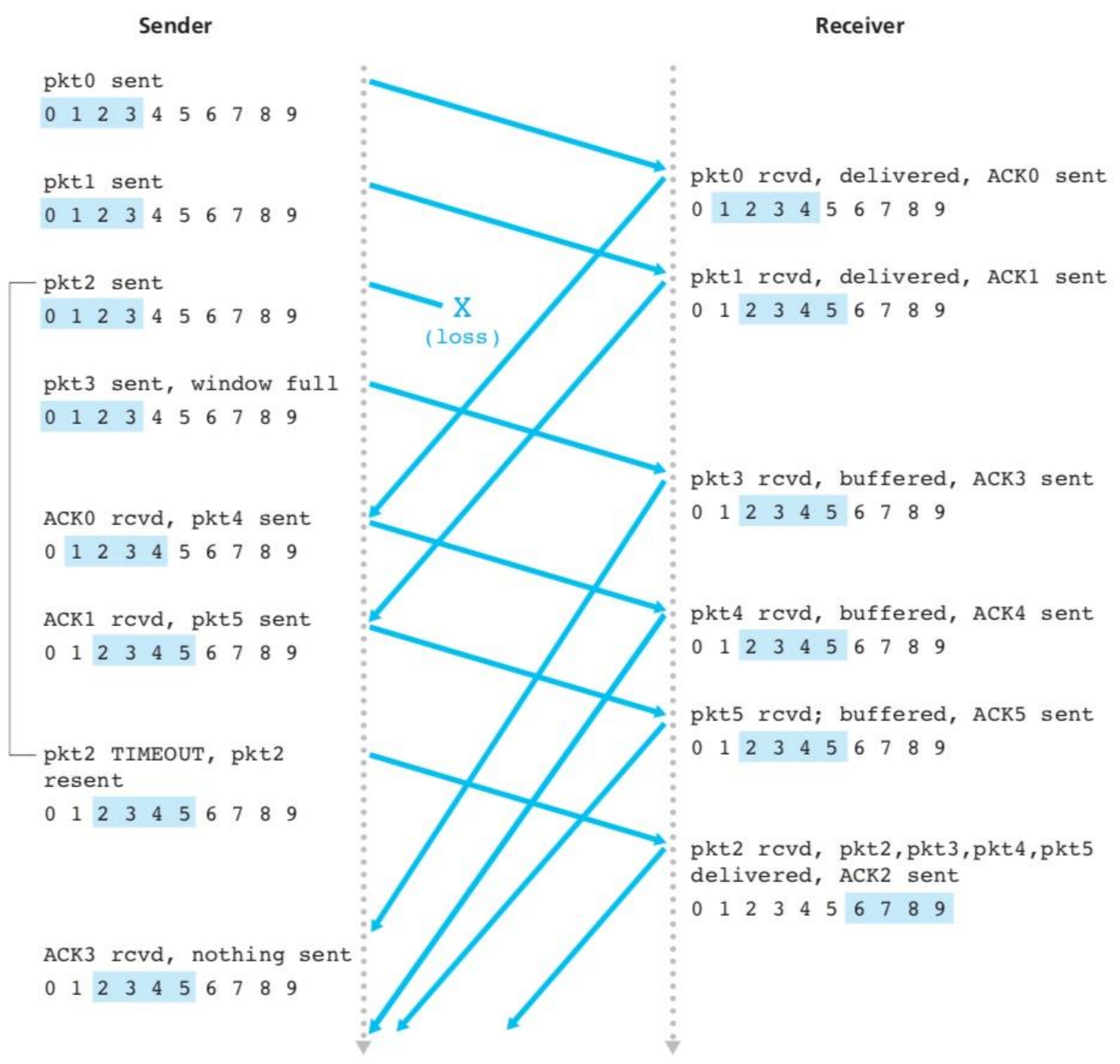


Figure 2.23 _ Selective-repeat (SR) sender and receiver views of sequence-number space

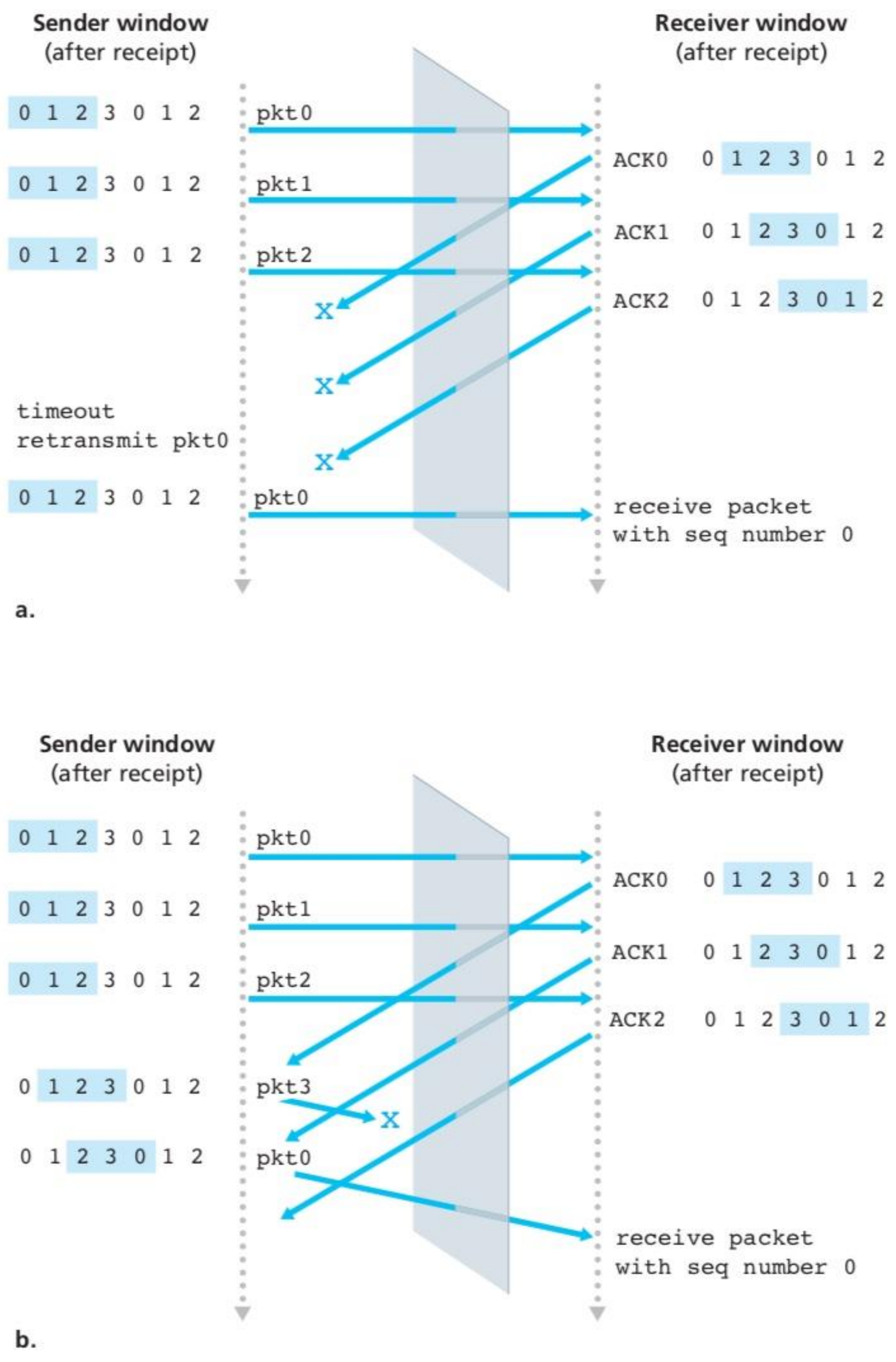

Sender window (after receipt)
Receiver window (after receipt)
0 1 2 3 0 1 2
pkt0
pkt1
pkt2
ACK0
ACK1
ACK2
timeout
retransmit pkt0
receive packet with seq number 0
a.
pkt3
b.

Figure 2.14 _ rdt2.2 receiver

Connection-Oriented Transport: TCP

The TCP Connection

TCP is said to be **connection-oriented** because before one application process can begin to send data to another, the two processes must first "handshake" with each other that is, they must send some preliminary segments to each other to establish the parameters of the ensuing data transfer. As part of TCP connection establishment, both sides of the connection will initialize many TCP state variables associated with the TCP connection.

The TCP "connection" is not an end-to-end TDM or FDM circuit as in a circuit switched network. Nor is it a virtual circuit (see Chapter 1), as the connection state resides entirely in the two end systems. Because the TCP protocol runs only in the end systems and not in the intermediate network elements (routers and link-layer switches), the intermediate network elements do not maintain TCP connection state. A TCP connection provides a **full-duplex service**: If there is a TCP connection between Process A on one host and Process B on another host, then applicationlayer data can flow from Process A to Process B at the same time as application layer data flows from Process B to Process A. A TCP connection is also always **point-to-point**, that is, between a single sender and a single receiver. So-called "multicasting" the transfer of data from one sender to many receivers in a single send operation is not possible with TCP.
Let's now take a look at how a TCP connection is established. Suppose a process running in one host wants to initiate a connection with another process in another host. Recall that the process that is initiating the connection is called the *client process*, while the other process is called the *server process*. The client application process first informs the client transport layer that it wants to establish a connection to a process in the server. Recall from earlier, a Python client program does this by issuing the command

```
clientSocket.connect((serverName,serverPort))
```

where serverName is the name of the server and serverPort identifies the process on the server. TCP in the client then proceeds to establish a TCP connection with TCP in the server. At the end of this section we discuss in some detail the connection establishment procedure. For now it suffices to know that the client first sends a special TCP segment; the server responds with a second special TCP segment; and finally the client responds again with a third special segment. The first two segments carry no payload, that is, no application-layer data; the third of these segments may carry a payload. Because three segments are sent between the two hosts, this connection establishment procedure is often referred to as a **three-way handshake**.

Once a TCP connection is established, the two application processes can send data to each other. Let’s consider the sending of data from the client process to the server process. The client process passes a stream of data through the socket (the door of the process), as described. Once the data passes through the door, the data is in the hands of TCP running in the client. As shown in Figure
2.28, TCP directs this data to the connection’s **send buffer**, which is one of the buffers that is set aside during the initial three-way handshake. From time to time, TCP will grab chunks of data from the send buffer and pass the data to the network layer. Interestingly, the TCP specification [RFC 793] is very laid back about specifying when TCP should actually send buffered data, stating that TCP should “send that data in segments at its own convenience.”

The maximum amount of data that can be grabbed and placed in a segment is limited by the **maximum segment size (MSS)**. The MSS is typically set by first determining the length of the largest link-layer frame that can be sent by the local sending host (the so-called **maximumtransmission unit**, **MTU**), and then setting the MSS to ensure that a TCP segment (when encapsulated in an IP datagram) plus the TCP/IP header length (typically 40 bytes) will fit into a single link-layer frame. Both Ethernet and PPP link-layer protocols have an MSS of 1,500 bytes. Approaches have also been proposed for discovering the path MTU—the largest link-layer frame that can be sent on all links from source to destination [RFC 1191]—and setting the MSS based on the path MTU value. Note that the MSS is the maximum amount of application-layer data in the segment, not the maximum size of the TCP segment including headers.

TCP Segment Structure

As with UDP, the header includes **source and destination port numbers**, which are used for multiplexing/demultiplexing data from/to upper-layer applications. Also, as with UDP, the header includes a **checksum field**. A TCP segment header also contains the following fields:

• The 32-bit **sequence number field** and the 32-bit **acknowledgment number field** are used by the TCP sender and receiver in implementing a reliable data transfer service, as discussed below.

• The 16-bit **receive window** field is used for flow control. We will see shortly that it is used to indicate the number of bytes that a receiver is willing to accept.

• The 4-bit **header length field** specifies the length of the TCP header in 32-bit words. The TCP header can be of variable length due to the TCP options field.

• The optional and variable-length **options field** is used when a sender and receiver negotiate the maximum segment size (MSS) or as a window scaling factor for use in high-speed networks. A time-stamping option is also defined.

• The **flag field** contains 6 bits. The **ACK bit** is used to indicate that the value carried in the acknowledgment field is valid; that is, the segment contains an acknowledgment for a segment that has been successfully received. The **RST**, **SYN**, and **FIN** bits are used for connection setup and teardown, as we will discuss at the end of this section. Setting the **PSH** bit indicates that the receiver should pass the data to the upper layer immediately. Finally, the **URG** bit is used to indicate that there is data in this segment that the sending-side upper-layer entity has marked as "urgent." The location of the last byte of this urgent data is indicated by the 16-bit **urgent data pointer field**. TCP must inform the receiving side upper-layer entity when urgent data exists and pass it a pointer to the end of the urgent data.

Round-Trip Time Estimation and Timeout

Estimating the Round-Trip Time

Let's begin our study of TCP timer management by considering how TCP estimates the round-trip time between sender and receiver. This is accomplished as follows. The sample RTT, denoted SampleRTT, for a segment is the amount of time between when the segment is sent (that is, passed to IP) and when an acknowledgment for the segment is received. Instead of measuring a SampleRTT for every transmitted segment, most TCP implementations take only one SampleRTT measurement at a time. That is, at any point in time, the SampleRTT is being estimated for only one of the transmitted but currently unacknowledged segments, leading to a new value of SampleRTT approximately once every RTT. Also, TCP never computes a SampleRTT for a segment that has been retransmitted; it only measures SampleRTT for segments that have been transmitted once. Obviously, the SampleRTT values will fluctuate from segment to segment due to congestion in the routers and to the varying load on the end systems. Because of this fluctuation, any given SampleRTT value may be atypical. In order to estimate a typical RTT, it is therefore natural to take some sort of average of the SampleRTT values. TCP maintains an average, called EstimatedRTT, of the SampleRTT values. Upon obtaining a new SampleRTT, TCP updates EstimatedRTT according to the following formula:

EstimatedRTT = (1 – _) • EstimatedRTT + _ • SampleRTT

The formula above is written in the form of a programming-language statement the new value of EstimatedRTT is a weighted combination of the previous value of EstimatedRTT and the new value for SampleRTT. The recommended value of _ is _ = 0.125 (that is, 1/8), in which case the formula above becomes:

EstimatedRTT = 0.875 • EstimatedRTT + 0.125 • SampleRTT

Note that EstimatedRTT is a weighted average of the SampleRTT values. more recent samples better reflect the current congestion in the network. In statistics, such an average is called an **exponential weighted moving average (EWMA)**. The word "exponential" appears in EWMA because the weight of a given SampleRTT decays exponentially fast as the updates proceed. In the homework problems you will be asked to derive the exponential term in EstimatedRTT. Figure 3.32 shows the SampleRTT values and EstimatedRTT for a value of = 1/8 for a TCP connection between gaia.cs.umass.edu (in Amherst, Massachusetts) to fantasia.eurecom.fr (in the south of France).
Clearly, the variations in the SampleRTT are smoothed out in the computation of the EstimatedRTT. In addition to having an estimate of the RTT, it is also valuable to have a measure of the variability of the RTT. [RFC 6298] defines the RTT variation, DevRTT, as an estimate of how much SampleRTT typically deviates from

EstimatedRTT: DevRTT = (1 – _) • DevRTT + _•| SampleRTT – EstimatedRTT |

Note that DevRTT is an EWMA of the difference between SampleRTT and EstimatedRTT. If the SampleRTT values have little fluctuation, then DevRTT will be small; on the other hand, if there is a lot of fluctuation, DevRTT will be large. The recommended value of β is 0.25.

Setting and Managing the Retransmission Timeout Interval

TimeoutInterval = EstimatedRTT + 4 • DevRTT

Reliable Data Transfer

TCP creates a **reliable data transfer service** on top of IP’s unreliable best effort service. TCP’s reliable data transfer service ensures that the data stream that a process reads out of its TCP receive buffer is uncorrupted, without gaps, without duplication, and in sequence; that is, the byte stream is exactly the same byte stream that was sent by the end system on the other side of the connection. How TCP provides a reliable data transfer involves many of the principles.

In our earlier development of reliable data transfer techniques, it was conceptually easiest to assume that an individual timer is associated with each transmitted but not yet acknowledged segment. While this is great in theory, timer management can require considerable overhead. Thus, the recommended TCP timer management procedures [RFC 6298] use only a *single* retransmission timer, even if there are multiple transmitted but not yet acknowledged segments. The TCP protocol described in this section follows this single-timer recommendation.

```
NextSeqNum=InitialSeqNumber
SendBase=InitialSeqNumber
loop (forever) {
    switch(event)
        event: data received from application above
            create TCP segment with sequence number NextSeqNum
            if (timer currently not running)
                start timer
            pass segment to IP
            NextSeqNum=NextSeqNum+length(data)
            break;
        event: timer timeout
            retransmit not-yet-acknowledged segment with
            smallest sequence number
            start timer
break;
event: ACK received, with ACK field value of y
if (y > SendBase) {
SendBase=y
if (there are currently any not-yet-acknowledged segments)
start timer
}
break;
} /* end of loop forever */
```

Figure 2.33 _ Simplified TCP sender

The second major event is the timeout. TCP responds to the timeout event by retransmitting the segment that caused the timeout. TCP then restarts the timer.

Flow Control

TCP provides a **flow-control service** to its applications to eliminate the possibility of the sender overflowing the receiver's buffer. Flow control is thus a speed-matching service matching the rate at which the sender is sending against the rate at which the receiving application is reading. As noted earlier, a TCP sender can also be throttled due to congestion within the IP network; this form of sender control is referred to as **congestion control**, a topic we will explore. Even though the actions taken by flow and congestion control are similar (the throttling of the sender), they are obviously taken for very different reasons. Unfortunately, many authors use the terms interchangeably, and the savvy reader would be wise to distinguish between them. Let's now discuss how TCP provides its flow-control service. In order to see the forest for the trees, we suppose throughout this section that the TCP implementation is such that the TCP receiver discards out-of-order segments.

TCP provides flow control by having the *sender* maintain a variable called the **receive window**. Informally, the receive window is used to give the sender an idea of how much free buffer space is available at the receiver. Because TCP is full-duplex, the sender at each side of the connection maintains a distinct receive window. Let's investigate the receive window in the context of a file transfer. Suppose that Host Ais sending a large file to Host B over a TCP connection. Host B allocates a receive buffer to this connection; denote its size by RcvBuffer. From time to time, the application process in Host B reads from the buffer. Define the following variables:

• LastByteRead: the number of the last byte in the data stream read from the buffer by the application process in B.
• LastByteRcvd: the number of the last byte in the data stream that has arrived from the network and has been placed in the receive buffer at B.

Because TCP is not permitted to overflow the allocated buffer, we must have

LastByteRcvd – LastByteRead _ RcvBuffer

The receive window, denoted rwnd is set to the amount of spare room in the buffer:

rwnd = RcvBuffer – [LastByteRcvd – LastByteRead]

Because the spare room changes with time, rwnd is dynamic. The variable rwnd is illustrated in Figure 3.38.

TCP Connection Management

The TCP in the client then proceeds to establish a TCP connection with the TCP in the server in the following manner:

• *Step 1*. The client-side TCP first sends a special TCP segment to the server-side TCP. This special segment contains no application-layer data. But one of the flag bits in the segment's header, the SYN bit, is set to 1. For this reason, this special segment is referred to as a SYN segment. In addition, the client randomly chooses an initial sequence number (client_isn) and puts
this number in the sequence number field of the initial TCP SYN segment. This segment is encapsulated within an IP datagram and sent to the server. There has been considerable interest in properly randomizing the choice of the client_isn in order to avoid certain security attacks.

• *Step 2.* Once the IP datagram containing the TCP SYN segment arrives at the server host (assuming it does arrive!), the server extracts the TCP SYN segment from the datagram, allocates the TCP buffers and variables to the connection, and sends a connection-granted segment to the client TCP. This connection-granted segment also contains no application layer data. However, it does contain three important pieces of information in the segment header. First, the SYN bit is set to 1. Second, the acknowledgment field of the TCP segment header is set to client_isn+1. Finally, the server chooses its own initial sequence number (server_isn) and puts this value in the sequence number field of the TCP segment header. This connection-granted segment is saying, in effect, "I received your SYN packet to start a connection with your initial sequence number, client_isn. I agree to establish this connection. My own initial sequence number is server_isn." The connectiongranted segment is referred to as a **SYNACK segment**.

• *Step 3.* Upon receiving the SYNACK segment, the client also allocates buffers and variables to the connection. The client host then sends the server yet another segment; this last segment acknowledges the server's connection-granted segment (the client does so by putting the value server_isn+1 in the acknowledgment field of the TCP segment header). The SYN bit is set to zero, since the connection is established. This third stage of the three-way handshake may carry
client-to-server data in the segment payload.

Once these three steps have been completed, the client and server hosts can send segments containing data to each other. In each of these future segments, the SYN bit will be set to zero. Note that in order to establish the connection, three packets are sent between the two hosts. For this reason, this connectionestablishment procedure is often referred to as a **three-way handshake**.

There are three possible outcomes:

• *The source host receives a TCP SYNACK segment from the target host*. Since this means that an application is running with TCP port 6789 on the target post, nmap returns "open."

• *The source host receives a TCP RST segment from the target host*. This means that the SYN segment reached the target host, but the target host is not running an application with TCP port 6789. But the attacker at least knows that the segments destined to the host at port 6789 are not blocked by any firewall on the path between source and target hosts.

• *The source receives nothing*. This likely means that the SYN segment was blocked by an intervening firewall and never reached the target host.

Principles of Congestion Control

The Causes and the Costs of Congestion

Scenario 1: Two Senders, a Router with Infinite Buffers

We begin by considering perhaps the simplest congestion scenario possible: Two hosts (A and B) each have a connection that shares a single hop between source and destination. Let's assume that the application in Host A is sending data into the connection (for example, passing data to the transport-level protocol via a socket) at an average rate of _in bytes/sec. These data are original in the sense that each unit of data is sent into the socket only once. The underlying transport-level protocol is a simple one.

Data is encapsulated and sent; no error recovery (for example, retransmission), flow control, or congestion control is performed. Ignoring the additional overhead due to adding transport- and lower-layer header information, the rate at which Host A offers traffic to the router in this first scenario is thus _in bytes/sec. Host B operates in a similar manner, and we assume for simplicity that it too is sending at a rate of _in bytes/sec. Packets from Hosts A and B pass through a router and over a shared outgoing link of capacity *R*. The router has buffers that allow it to store incoming packets when the packet-arrival rate exceeds the outgoing link's capacity.

Scenario 2: Two Senders and a Router with Finite Buffers

The performance realized under scenario 2 will now depend strongly on how retransmission is performed. First, consider the unrealistic case that Host A is able to somehow (magically!) determine whether or not a buffer is free in the router and thus sends a packet only when a buffer is free.

Scenario 3: Four Senders, Routers with Finite Buffers, and Multihop Paths

Let's consider the connection from Host A to Host C, passing through routers R1 and R2. The A–C connection shares router R1 with the D–B connection and shares router R2 with the B–D connection. For extremely small values of _in, buffer overflows are rare (as in congestion scenarios 1 and 2), and the throughput approximately equals the offered load. For slightly larger values of _in, the corresponding throughput is also larger, since more original data is being transmitted into thenetwork and delivered to the destination, and overflows are still rare. Thus, for small values of _in, an increase in _in results in an increase in _out.

Approaches to Congestion Control

At the broadest level, we can distinguish among congestion-control approaches by whether the network layer provides any explicit assistance to the transport layer for congestion-control purposes:

• *End-to-end congestion control.* In an end-to-end approach to congestion control, the network layer provides *no explicit support* to the transport layer for congestion control purposes. Even the presence of congestion in the network must be inferred by the end systems based only on observed network behavior (for example, packet loss and delay). TCP segment loss (as indicated by a timeout or a triple duplicate acknowledgment) is taken as an indication of network congestion and TCP decreases its window size accordingly. We will also see a more recent proposal for TCP congestion control that uses increasing round-trip delay values as indicators of increased network congestion.

• *Network-assisted congestion control.* With network-assisted congestion control, network-layer components (that is, routers) provide explicit feedback to the sender regarding the congestion state in the network. This feedback may be as simple as a single bit indicating congestion at a link. Recently proposed for TCP/IP networks is used in ATM available bit-rate (ABR) congestion control as well, as discussed below. More sophisticated network feedback is also possible. For example, one form of ATM ABR congestion control that we will study shortly allows a router to inform the sender explicitly of the transmission rate it (the router) can support on an outgoing link. The XCP protocol provides router-computed feedback to each source, carried in the packet header, regarding how that source should increase or decrease its transmission rate.

Network-Assisted Congestion-Control Example:
ATM ABR Congestion Control

Fundamentally ATM takes a virtual-circuit (VC) oriented approach toward packet switching. Recall from our discussion in Chapter 1, this means that each switch on the source-to-destination path will maintain

state about the source-to destination VC. This per-VC state allows a switch to track the behavior of individual senders (e.g., tracking their average transmission rate) and to take
source-specific congestion-control actions (such as explicitly signaling to the sender to reduce its rate when the switch becomes congested). This per-VC state at network switches makes ATM ideally suited to perform network-assisted congestion control.

ABR has been designed as an elastic data transfer service in a manner reminiscent of TCP. When the network is underloaded, ABR service should be able to take advantage of the spare available bandwidth; when the network is congested, ABR service should throttle its transmission rate to some predetermined minimum transmission rate.

In our discussion we adopt ATM terminology (for example, using the term *switch* rather than *router*, and the term *cell* rather than *packet*). With ATM ABR service, data cells are transmitted from a source to a destination through a series of intermediate switches. Interspersed with the data cells are **resource-management cells(RM cells)**; these RM cells can be used to convey congestion-related information among the hosts and switches. When an RM cell arrives at a destination, it will be turned around and sent back to the sender (possibly after the destination has
modified the contents of the RM cell). It is also possible for a switch to generate an RM cell itself and send this RM cell directly to a source. RM cells can thus be used to provide both direct network feedback and network feedback via the receiver.

ATM ABR congestion control is a rate-based approach. That is, the sender explicitly computes a maximum rate at which it can send and regulates itself accordingly. ABR provides three mechanisms for signaling congestion-related information from the switches to the receiver:

• *EFCI bit.* Each *data cell* contains an **explicit forward congestion indication (EFCI) bit**. A congested network switch can set the EFCI bit in a data cell to 1 to signal congestion to the destination host. The destination must check the EFCI bit in all received data cells. When an RM cell arrives at the destination, if the most recently received data cell had the EFCI bit set to 1, then the destination sets the congestion indication bit (the CI bit) of the RM cell to 1 and sends the RM cell back to the sender. Using the EFCI in data cells and the CI bit in RM cells, a sender can thus be notified about congestion at a network switch.

• *CI and NI bits.* As noted above, sender-to-receiver RM cells are interspersed with data cells. The rate of RM cell interspersion is a tunable parameter, with the default value being one RM cell every 32 data cells. These RM cells have a **congestion indication (CI) bit** and a **no increase (NI) bit** that can be set by a congested network switch. Specifically, a switch can set the NI bit in a passing RM cell to 1 under mild congestion and can set the CI bit to 1 under severe congestion conditions. When a destination host receives an RM cell, it will send the RM cell back to the sender with its CI and NI bits intact (except that CI may be set to 1 by the destination as a result of the EFCI mechanism described above).

• *ER setting.* Each RM cell also contains a 2-byte **explicit rate (ER) field**. A congested switch may lower the value contained in the ER field in a passing RM cell. In this manner, the ER field will be set to the minimum supportable rate of all switches on the source-to-destination path.

TCP Congestion Control

TCP answers these questions using the following guiding principles:

• *A lost segment implies congestion, and hence, the TCP sender's rate should be decreased when a segment is lost.* Recall from our discussion in 4, that a timeout event or the receipt of four acknowledgments for a given segment (one original ACK and then three duplicate ACKs) is interpreted as an implicit "loss event" indication of the segment following the quadruply ACKed
segment, triggering a retransmission of the lost segment. From a congestion control standpoint, the question is how the TCP sender should decrease its congestion window size, and hence its sending rate, in response to this inferred loss event.

• *An acknowledged segment indicates that the network is delivering the sender's segments to the receiver, and hence, the sender's rate can be increased when an ACK arrives for a previously unacknowledged segment.* The arrival of acknowledgments is taken as an implicit indication that all is well—segments are being successfully delivered from sender to receiver, and the network is thus not congested. The congestion window size can thus be increased.

• *Bandwidth probing*. Given ACKs indicating a congestion-free source-to-destination path and loss events indicating a congested path, TCP's strategy for adjusting its transmission rate is to increase its rate in response to arriving ACKs until a loss event occurs, at which point, the transmission rate is decreased. The TCP sender thus increases its transmission rate to probe for the rate that at which congestion onset begins, backs off from that rate, and then to begins probing again to see if the congestion onset rate has changed. The TCP sender's behavior is perhaps analogous to the child who requests (and gets) more and more goodies until finally he/she is finally told "No!", backs off a bit, but then begins making requestsagain shortly afterwards. Note that there is no explicit signaling of congestion state by the network—ACKs and loss events serve as implicit signals—and that each TCP sender acts on local information asynchronously from other TCP senders.

Given this overview of TCP congestion control, we're now in a position to consider the details of the celebrated **TCP congestion-control algorithm**

Slow Start

When a TCP connection begins, the value of cwnd is typically initialized to a small value of 1 MSS [RFC 3390], resulting in an initial sending rate of roughly MSS/RTT. For example, if MSS = 500 bytes and RTT = 200 msec, the resulting initial sending rate is only about 20 kbps. Since the available bandwidth to the TCP sender may be much larger than MSS/RTT, the TCP sender would like to find the amount of available bandwidth quickly. Thus, in the **slow-start** state, the
value of cwnd begins at 1 MSS and increases by 1 MSS every time a transmitted segment is first acknowledged.

In the example TCP sends the first segment into the network and waits for an acknowledgment. When this acknowledgment arrives, the TCP sender increases the congestion window by one MSS and sends out two maximum-sized segments. These segments are then acknowledged, with the sender increasing the congestion window by 1 MSS for each of the acknowledged segments, giving a congestion window of 4 MSS, and so on. This process results in a doubling of the sending rate every RTT. Thus, the TCP send rate starts slow but grows exponentially during the slow start phase.

Congestion Avoidance

On entry to the congestion-avoidance state, the value of cwnd is approximately half its value when congestion was last encountered—congestion could be just around the corner! Thus, rather than doubling the value of cwnd every RTT, TCP adopts a more conservative approach and increases the value of cwnd by just a single MSS every RTT [RFC 5681]. This can be accomplished in several ways. A common approach is for the TCP sender to increase cwnd by MSS bytes (MSS/cwnd) whenever a new acknowledgment arrives. For example, if MSS is 1,460 bytes and cwnd is 14,600 bytes, then 10 segments are being sent within an RTT. Each arriving ACK (assuming one ACK per segment) increases the congestion window size by 1/10 MSS, and thus, the value of the congestion window will have increased by one MSS after ACKs when all 10 segments have been received.

Fast Recovery

In fast recovery, the value of cwnd is increased by 1 MSS for every duplicate ACK received for the missing segment that caused TCP to enter the fast-recovery state. Eventually, when an ACK arrives for the missing segment, TCP enters the congestion-avoidance state after deflating cwnd. If a timeout event occurs, fast recovery transitions to the slow-start state after performing the same actions as in slow start and congestion avoidance: The value of cwnd is set to 1 MSS, and the value of ssthresh is set to half the value of cwnd when the loss event occurred.

Fast recovery is a recommended, but not required, component of TCP [RFC 5681]. It is interesting that an early version of TCP, known as **TCP Tahoe**, unconditionally cut its congestion window to 1 MSS and entered the slow-start phase after either a timeout-indicated or triple-duplicate-ACK-indicated loss event. The newer version of TCP, **TCP Reno**, incorporated fast recovery. The evolution of TCP's congestion window for both Reno and Tahoe. In this figure, the threshold is initially equal to 8 MSS. For the first eight transmission rounds, Tahoe and Reno take identical actions. The congestion window climbs exponentially fast during slow start and hits the threshold at the fourth round of transmission. The congestion window then climbs linearly until a triple duplicate- ACK event occurs, just after transmission round 8. Note that the congestion window is

12 • *MSS* when this loss event occurs. The value of ssthresh is then set to 0.5 • cwnd = 6 • MSS. Under TCP Reno, the congestion window is set to cwnd = 6 • MSS and then grows linearly. Under TCP Tahoe, the

congestion window is set to 1 MSS and grows exponentially until it reaches the value of ssthresh, at which point it grows linearly.

Fairness

Fairness and UDP

We have just seen how TCP congestion control regulates an application's transmission rate via the congestion window mechanism. Many multimedia applications, such as Internet phone and video conferencing, often do not run over TCP for this very reason—they do not want their transmission rate throttled, even if the network is very congested. Instead, these applications prefer to run over UDP, which does not have built-in congestion control. When running over UDP, applications can
pump their audio and video into the network at a constant rate and occasionally lose packets, rather than reduce their rates to "fair" levels at times of congestion and not lose any packets. From the perspective of TCP, the multimedia applications running over UDP are not being fair—they do not cooperate with the other connections nor adjust their transmission rates appropriately. Because TCP congestion control will decrease its transmission rate in the face of increasing congestion (loss), while UDP sources need not, it is possible for UDP sources to crowd out TCP traffic.

Fairness and Parallel TCP Connections

But even if we could force UDP traffic to behave fairly, the fairness problem would still not be completely solved. This is because there is nothing to stop a TCP-based application from using multiple parallel connections. For example, Web browsers often use multiple parallel TCP connections to transfer the multiple objects within a Web page. (The exact number of multiple connections is configurable in most browsers.) When an application uses multiple parallel connections, it gets a larger fraction of the bandwidth in a congested link. As an example, consider a link of rate *R* supporting nine ongoing client-server applications, with each of the applications
using one TCP connection. If a new application comes along and also uses one TCP connection, then each application gets approximately the same transmission rate of *R*/10. But if this new application instead uses 11 parallel TCP connections, then the new application gets an unfair allocation of more than *R*/2. Because Web traffic is so pervasive in the Internet, multiple parallel connections are not uncommon.

Module – 3

The Network layer

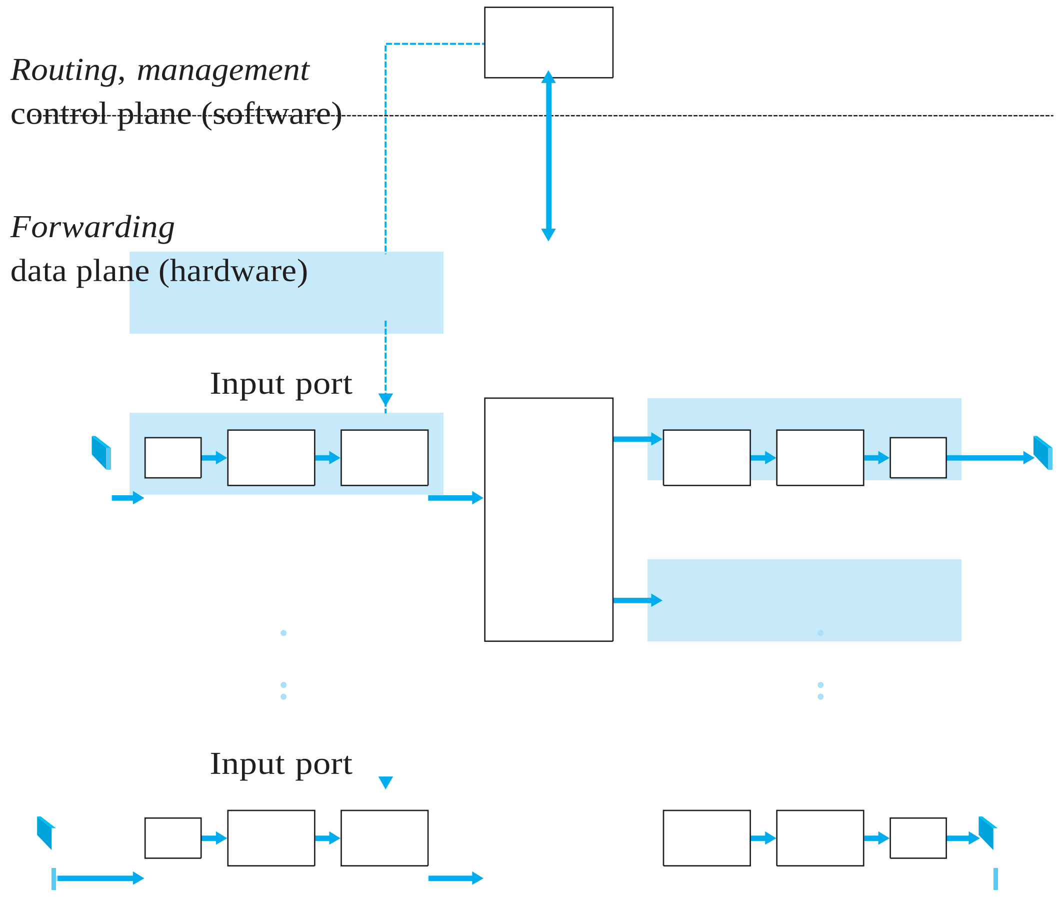
What's Inside a Router?

The actual transfer of packets from a router's incoming links to the appropriate outgoing links at that router. We already took a brief look at a few aspects of forwarding in Section 4.2, namely, addressing and longest prefix matching. We mention here in passing that the terms *forwarding* and *switching* are often used interchangeably by computer-networking researchers and practitioners; we'll use both terms interchangeably in this textbook as well. Four router components can be identified: *Input ports*. An input port performs several key functions. It performs the physical layer function of terminating an incoming physical link at a router; this is shown in the leftmost box of the input port and the rightmost box of the output port in. Per-haps most crucially, the lookup function is also performed at the input port; this will occur in the rightmost box of the input port. It is here that the for-warding table is consulted to determine the router output port to which an arriving packet will be forwarded via the switching fabric. Control packets (for example, packets carrying routing protocol information) are forwarded from an input port to the routing processor. Note that the term *port* here—referring to the physical input and output router interfaces.

- *Switching fabric*. The switching fabric connects the router's input ports to its output ports. This switching fabric is completely contained within the router—a network inside of a network router!

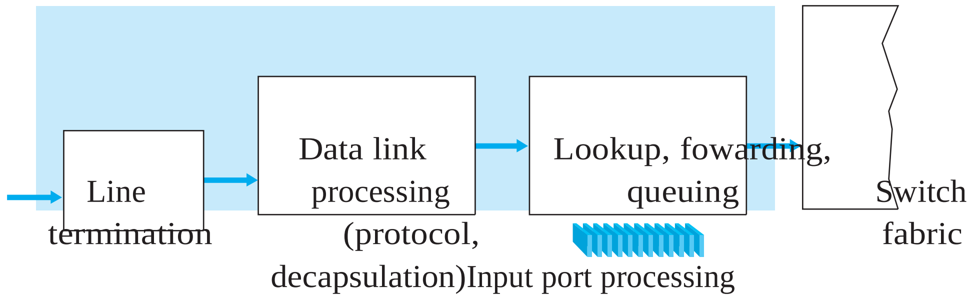
- *Output ports*. An output port stores packets received from the switching fabric and transmits these packets on the outgoing link by performing the necessary link-layer and physical-layer functions. When a link is bidirectional (that is carries traffic in both directions), an output port will typically be paired with the input port for that link on the same line card (a printed circuit board containing one or more input ports, which is connected to the switching fabric).

Routing processor. The routing processor executes the routing protocols (which we'll study in Section 4.6), maintains routing tables and attached link state infor-mation, and computes the forwarding table for the router.



Router architecture

Given the existence of a forwarding table, lookup is conceptually simple—we just search through the forwarding table looking for the longest prefix match, as described



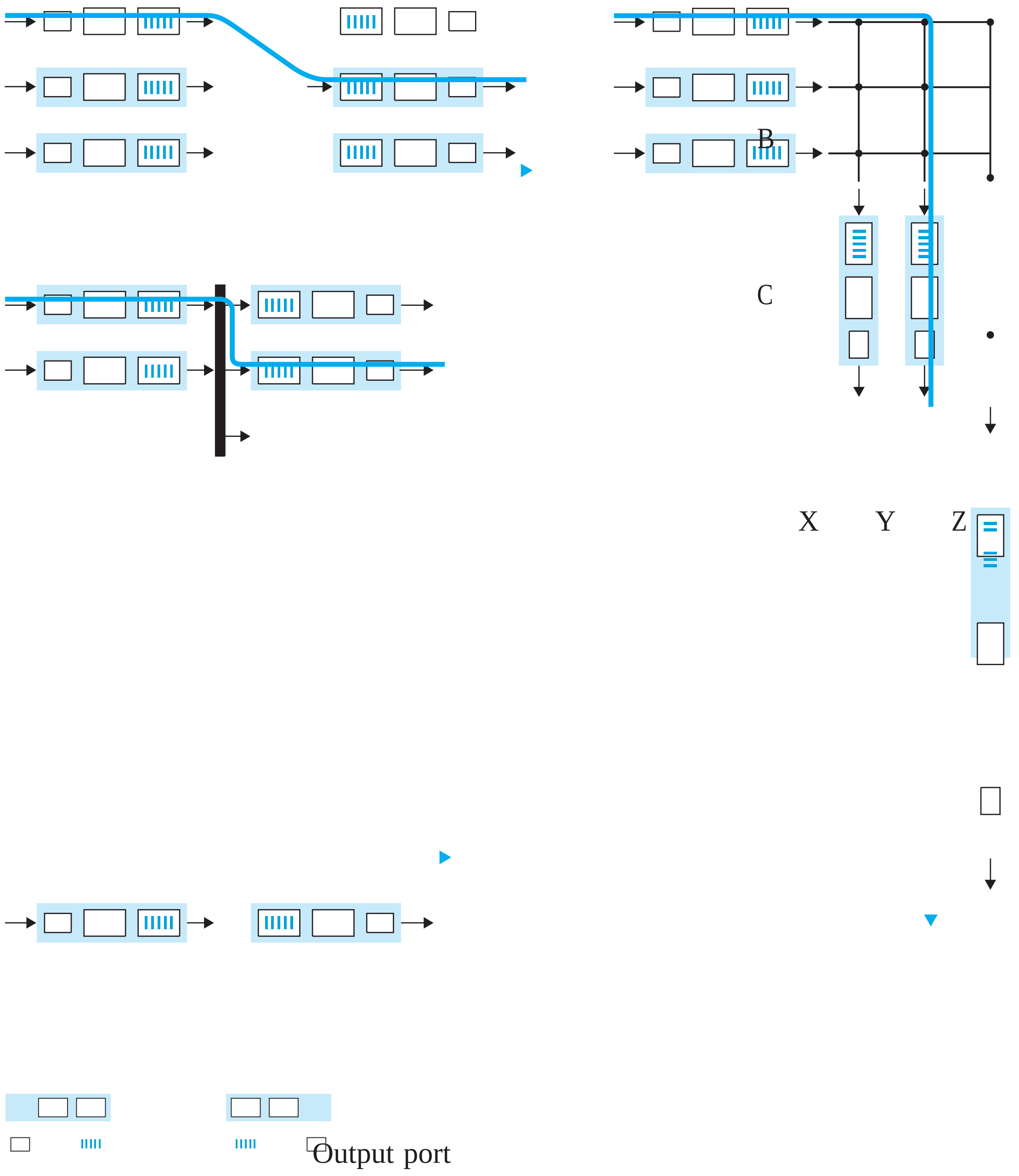
Input port processing

Switching

The switching fabric is at the very heart of a router, as it is through this fabric that the packets are actually switched (that is, forwarded) from an input port to an output port. Switching can be accomplished in a number of ways, as shown in Figure.

- *Switching via memory*. The simplest, earliest routers were traditional computers, with switching between input and output ports being done under direct control of the

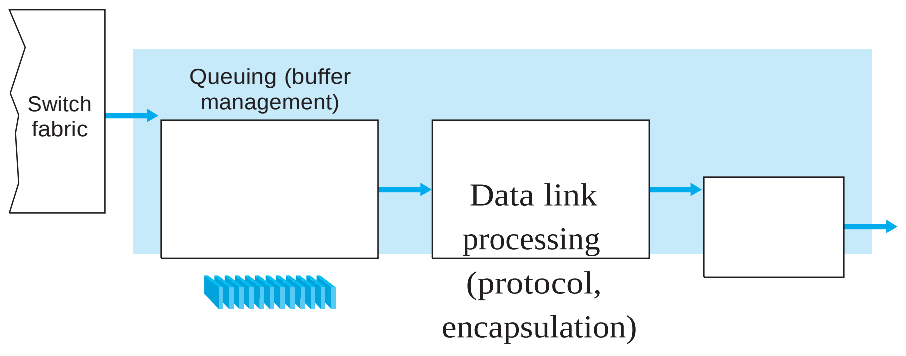
CPU (routing processor). Input and output ports functioned as traditional I/O devices in a traditional operating system. An input port with an arriving packet first signaled the routing processor via an interrupt. The packet was then copied from the input port into processor memory



Three switching techniques

Output Processing

Output port processing, shown in Figure 4.9, takes packets that have been stored in the output port's memory and transmits them over the output link. This includes selecting and de-queueing packets for transmission, and performing the needed link-layer and physical-layer transmission functions.


Switch fabric
Queuing (buffer management)
Data link processing (protocol, encapsulation)

Output port processing

IPv6 Datagram Format

The format of the IPv6 datagram is shown in Figure 4.24. The most important changes introduced in IPv6 are evident in the datagram format:

- *Expanded addressing capabilities*. IPv6 increases the size of the IP address from 32 to 128 bits. This ensures that the world won't run out of IP addresses. Now, every grain of sand on the planet can be IP-addressable. In addition to unicast and multicast addresses, IPv6 has introduced a new type of address, called an **any cast address**, which allows a datagram to be delivered to any one of a group of hosts. (This feature could be used, for example, to send an HTTP GET to the nearest of a number of mirror sites that contain a given document.)

As noted above, a comparison of Figure 4.24 with Figure 4.13 reveals the sim-
pler, more streamlined structure of the IPv6 datagram. The following fields are defined in IPv6:

- *Version*. This 4-bit field identifies the IP version number. Not surprisingly, IPv6 carries a value of 6 in this field. Note that putting a 4 in this field does not create a valid IPv4 datagram. (If it did, life would be a lot simpler—see the discussion below regarding the transition from IPv4 to IPv6.)

Traffic class. This 8-bit field is similar in spirit to the TOS field we saw in IPv4. • *Flow label*. As discussed above, this 20-bit field is used to identify a flow of datagrams.

- *Payload length*. This 16-bit value is treated as an unsigned integer giving the number of bytes in the IPv6 datagram following the fixed-length, 40-byte data-gram header.

- *Next header*. This field identifies the protocol to which the contents (data field) of this datagram will be delivered (for example, to TCP or UDP). The field uses the same values as the protocol field in the IPv4 header.
- *Hop limit*. The contents of this field are decremented by one by each router that forwards the datagram. If the hop limit count reaches zero, the datagram is discarded.
- *Source and destination addresses*. The various formats of the IPv6 128-bit address are described in RFC 4291.
- *Data*. This is the payload portion of the IPv6 datagram. When the datagram reaches its destination, the payload will be removed from the IP datagram and passed on to the protocol specified in the next header field.

The Link-State (LS) Routing Algorithm

- *D*(*v*): cost of the least-cost path from the source node to destination *v* as of this iteration of the algorithm.
- *p*(*v*): previous node (neighbor of *v*) along the current least-cost path from the source to *v*.
- *N* : subset of nodes; *v* is in *N* if the least-cost path from the source to *v* is definitively known.

The global routing algorithm consists of an initialization step followed by a loop. The number of times the loop is executed is equal to the number of nodes in the network. Upon termination, the algorithm will have calculated the shortest paths from the source node *u* to every other node in the network.

Link-State (LS) Algorithm for Source Node *u*

```
Initialization:
    N' = {u}
    for all nodes v
        if v is a neighbor of u
            then D(v) = c(u,v)
        else D(v) = ∞

Loop
    find w not in N' such that D(w) is a minimum
    add w to N'
    update D(v) for each neighbor v of w and not in N':
              D(v) = min( D(v), D(w) + c(w,v) )
    /* new cost to v is either old cost to v or known
      least path cost to w plus cost from w to v */
```

15 **until** N’= N

The Distance-Vector (DV) Routing Algorithm

Whereas the LS algorithm is an algorithm using global information, the **distance-vector** (**DV**) algorithm is iterative, asynchronous, and distributed. It is *distributed* in that each node receives some information from one or more of its *directly attached* neighbors, performs a calculation, and then distributes the results of its calculation back to its neighbors. It is *iterative* in that this process continues on until no more information is exchanged between neighbors. (Interestingly, the algorithm is also self-terminating—there is no signal that the computation should stop; it just stops.)

In the distributed, asynchronous algorithm, from time to time, each node sends a copy of its distance vector to each of its neighbors. When a node *x* receives a new distance vector from any of its neighbors *v*, it saves *v*’s distance vector, and then uses the Bellman-Ford equation to update its own distance vector as follows:

$$D(y) \quad \min\{c(x,v) + D(y)\} \quad \text{for each node } y \text{ in } N$$

At each node, *x*:

```
Initialization:
    for all destinations y in N:
        $D_x(y) = c(x,y)$    /* if y is not a neighbor then $c(x,y) = \infty$ */
    for each neighbor w
        $D_w(y)$ = ? for all destinations y in N
    for each neighbor w
        send distance vector $\mathbf{D}_x = [D_x(y)$: y in N] to w
loop
    wait (until I see a link cost change to some neighbor w or
          until I receive a distance vector from some neighbor w)

    for each y in N:
        $D_x(y) = \min_v\{c(x,v) + D_v(y)\}$

    if $D_x(y)$ changed for any destination
        send $\mathbf{D}_x$
```

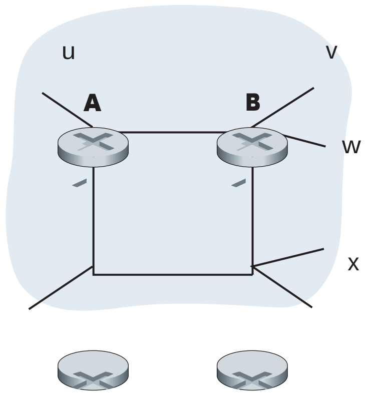
Routing in the Internet

Intra-AS Routing in the Internet: RIP

An intra-AS routing protocol is used to determine how routing is performed within an autonomous system (AS). Intra-AS routing protocols are also known as **interior**

gateway protocols. Historically, two routing protocols have been used extensively for routing within an autonomous system in the Internet: the **Routing Information Protocol (RIP)** and **Open Shortest Path First (OSPF)**. A routing protocol closely related to OSPF is the **IS-IS** protocol [RFC 1142, Perlman 1999]. We first discuss RIP and then consider OSPF.

The maximum cost of a path is limited to 15, thus limiting the use of RIP to autonomous systems that are fewer than 15 hops in diameter. Recall that in DV protocols, neighboring routers exchange distance vectors with each other. The distance vector for any one router is the current estimate of the shortest path distances from that router to the subnets in the AS. In RIP, routing updates are exchanged between neighbors approximately every 30 seconds using a **RIP response message**. The response message sent by a router or host contains a list of up to 25 destination subnets within the AS, as well as the sender's distance to each of those subnets. Response messages are also known as **RIP advertisements**.



Destination	Hops
u	1
v	2
w	2
x	3
y	3
z	2

Number of hops from source router A to various subnets

Intra-AS Routing in the Internet: OSPF

Like RIP, OSPF routing is widely used for intra-AS routing in the Internet. OSPF an its closely related cousin, IS-IS, are typically deployed in upper-tier ISPs whereas RIP is deployed in lower-tier ISPs and enterprise networks. The Open in OSPF indicates that the routing protocol specification is publicly available (for example, as opposed to Cisco's EIGRP protocol). The most recent version of OSPF, version 2, is defined in RFC 2328, a public document.

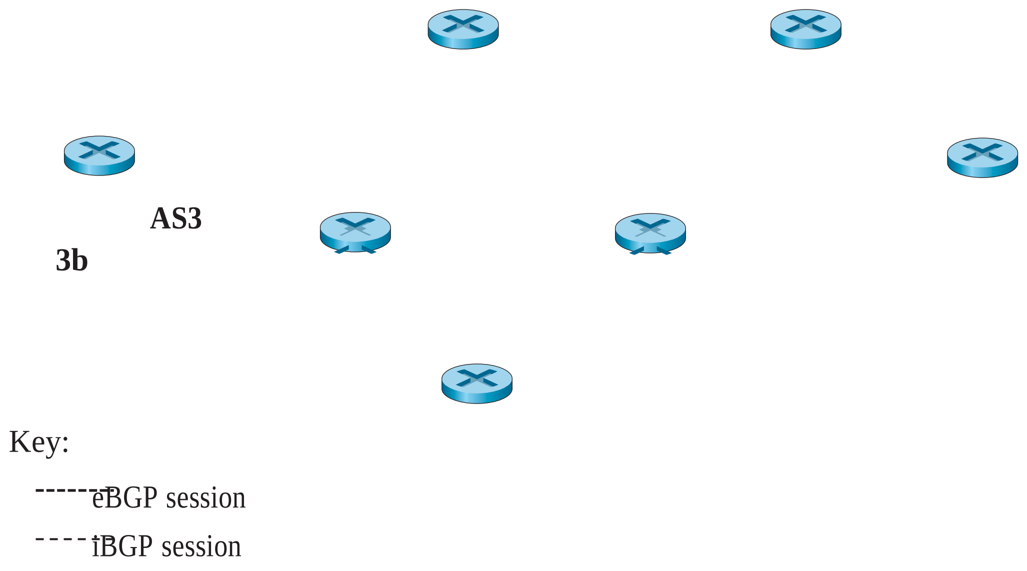
OSPF was conceived as the successor to RIP and as such has a number of advanced features. At its heart, however, OSPF is a link-state protocol that uses flooding of link-state information and a Dijkstra least-cost path algorithm. With OSPF, a router constructs a complete topological map (that is, a graph) of the entire autonomous system. The router then locally runs Dijkstra's shortest-path algorithm to determine a shortest-path tree to all *subnets*, with itself as the root node. Individ- ual link costs are configured by the network administrator (see Principles and Prac-tice: Setting OSPF Weights). The administrator might choose to set all link costs to

1, thus achieving minimum-hop routing, or might choose to set the link weights to be inversely proportional to link capacity in order to discourage traffic from using low-bandwidth links. OSPF does not mandate a policy for how link weights are set (that is the job of the network administrator), but instead provides the mechanisms (protocol) for determining least-cost path routing for the given set of link weights.

Inter-AS Routing: BGP

We just learned how ISPs use RIP and OSPF to determine optimal paths for source-destination pairs that are internal to the same AS. Let's now examine how paths are determined for source-destination pairs that span multiple ASs. The **Border Gate-way Protocol** version 4, specified in RFC 4271 (see also [RFC 4274), is the *de facto* standard inter-AS routing protocol in today's Internet. It is commonly referred to as BGP4 or simply as **BGP**. As an inter-AS routing protocol (see Section 4.5.3), BGP provides each AS a means to

1. Obtain subnet reachability information from neighboring ASs.
2. Propagate the reachability information to all routers internal to the AS.
3. Determine "good" routes to subnets based on the reachability information and on AS policy.



eBGP and iBGP sessions

• AS-PATH. This attribute contains the ASs through which the advertisement for the prefix has passed. When a prefix is passed into an AS, the AS adds its ASN to the AS-

PATH attribute. For example, consider Figure 4.40 and suppose that prefix

138.16.64/24 is first advertised from AS2 to AS1; if AS1 then advertises the prefix to

AS3, AS-PATH would be AS2 AS1. Routers use the AS-PATH attribute to detect and prevent looping advertisements; specifically, if a router sees that its AS is contained in the path list, it will reject the advertisement. As we'll soon discuss, routers also use the AS-PATH attribute in choosing among multiple paths to the same prefix.

• Providing the critical link between the inter-AS and intra-AS routing protocols, the

NEXT-HOP attribute has a subtle but important use. The NEXT-HOP is the router interface that begins the AS-PATH. To gain insight into this attribute, let's again refer to Figure 4.40. Consider what happens when the gateway router 3a in AS3 advertises a route to gateway router 1c in AS1 using eBGP. The route includes the advertised prefix, which we'll call x, and an AS-PATH to the prefix

Uncontrolled Flooding

The most obvious technique for achieving broadcast is a **flooding** approach in which the source node sends a copy of the packet to all of its neighbors. When a node receives a broadcast packet, it duplicates the packet and forwards it to all of its neighbors (except the neighbor from which it received the packet). Clearly, if the graph is connected, this scheme will eventually deliver a copy of the broadcast packet to all nodes in the graph. Although this scheme is simple and elegant, it has a fatal flaw (before you read on, see if you can figure out this fatal flaw): If the graph has cycles, then one or more copies of each broadcast packet will cycle indefinitely.
For example, in Figure 4.43, R2 will flood to R3, R3 will flood to R4, R4 will flood to R2, and R2 will flood (again!) to R3, and so on. This simple scenario results in the endless cycling of two broadcast packets, one clockwise, and one counter clock- wise. But there can be an even more calamitous fatal flaw: When a node is connected to more than two other nodes, it will create and forward multiple copies of the broadcast packet, each of which will create multiple copies of itself (at other nodes with more than two neighbors), and so on. This **broadcast storm**, resulting from the endless multiplication of broadcast packets, would eventually result in so many broadcast packets being created that the network would be rendered useless.
(See the homework questions at the end of the chapter for a problem analyzing the rate at which such a broadcast storm grows.)

Controlled Flooding

The key to avoiding a broadcast storm is for a node to judiciously choose when to flood a packet and (e.g., if it has already received and flooded an earlier copy of a packet) when not to flood a packet. In practice, this can be done in one of several ways.

In **sequence-number-controlled flooding**, a source node puts its address (or other unique identifier) as well as a **broadcast sequence number** into a broadcast packet, then sends the packet to all of its neighbors. Each node maintains a list of the source address and sequence number of each broadcast packet it has already received, duplicated, and forwarded. When a node receives a broadcast packet, it first checks whether the packet is in this list. If so, the packet is dropped; if not, the

Multicast Routing Algorithms

The **multicast routing problem** is illustrated in Figure 4.49. Hosts joined to the multicast group are shaded in color; their immediately attached router is also shaded in color. As shown in Figure 4.49, only a subset of routers (those with attached hosts that are joined to the multicast group) actually needs to receive the multicast traffic. In Figure 4.49, only routers *A*, *B*, *E*, and *F* need to receive the multicast traffic.

Module – 4
Mobile and Multimedia Networks

4.1Cellular Internet Access

In the previous section we examined how an Internet host can access the Internet when inside a WiFi hotspot—that is, when it is within the vicinity of an 802.11 access point. But most WiFi hotspots have a small coverage area of between 10 and 100 meters in diameter. What do we do then when we have a desperate need for wireless Internet access and we cannot access a WiFi hotspot? Given that cellular telephony is now ubiquitous in many areas throughout the world, a natural strategy is to extend cellular networks so that they support not only voice telephony but wireless Internet access as well. Ideally, this Internet access would be at a reasonably high speed and would provide for seamless mobility, allowing users to maintain their TCP sessions while traveling, for example, on a bus or a train. With sufficiently high upstream and downstream bit
rates, the user could even maintain video-conferencing sessions while roaming about. This scenario is not that far-fetched. As of 2012, many cellular telephony providers in the U.S. offer their subscribers a cellular Internet access service for under $50 per month with typical downstream and upstream bit rates in the hundreds of kilobits per second. Data rates of several megabits per second are ecoming available as broadband data services such as those we will cover here become more widely deployed. In this section, we provide a brief overview of current and emerging cellular Internet access technologies. Our focus here will be on both the wireless first hop as well as the network that connects the wireless first hop into the larger telephone network and/or the Internet; in Section 6.7 we'll consider how calls are routed to a user moving between base stations.

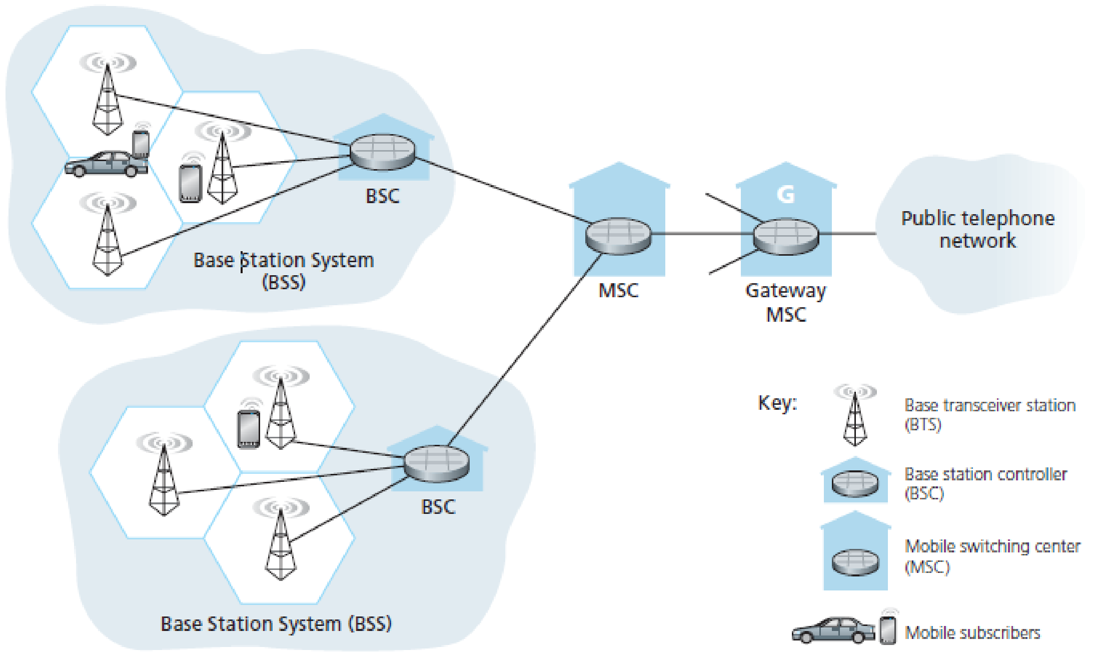
An Overview of Cellular Network Architecture

In our description of cellular network architecture in this section, we'll adopt the terminology of the *Global System for Mobile Communications (**GSM**)* standards. (For history buffs, the GSM acronym was originally derived from *GroupeSpécial Mobile*, until the more anglicized name was adopted, preserving the original acronym letters.) In the 1980s, Europeans recognized the need for a pan-European digital cellular telephony system that would replace the numerous incompatible analog cellular telephony systems, leading to the GSM standard [Mouly 1992]. Europeans deployed GSM technology with great success in the early 1990s, and since then GSM has grown to be the 800-pound gorilla of the cellular telephone world, with more than 80% of all cellular subscribers worldwide using GSM. When people talk about cellular technology, they often classify the technology as belonging to one of several "generations." The earliest generations were designed primarily for voice traffic. First generation (1G) systems were analog FDMA systems designed exclusively for voice-only communication. These 1G systems are almost extinct now, having been replaced by digital 2G systems. The original 2G systems were also designed for voice, but later extended (2.5G) to support data (i.e., Internet) as well as voice service. The 3G systems that currently are being deployed also support voice and data, but with an ever increasing emphasis on data capabilities and higher-speed radio access links.

Cellular Network Architecture, 2G: Voice Connections to the Telephone Network

The term *cellular* refers to the fact that the region covered by a cellular network is partitioned into a number of geographic coverage areas, known as **cells**, shown as hexagons on the left side of Figure 6.18. As with the 802.11WiFi standard we studied in Section 6.3.1, GSM has its own particular nomenclature. Each cell contains a **base transceiver station (BTS)** that transmits signals to and receives signals from the mobile stations in its cell. The coverage area of a cell depends on many factors, including the transmitting power of the BTS, the transmitting power of the user devices, obstructing buildings in the cell, and the height of base station antennas. Although Figure 6.18 shows each cell containing one base transceiver station residing

in the middle of the cell, many systems today place the BTS at corners where three cells intersect, so that a single BTS with directional antennas can service three cells. The GSM standard for 2G cellular systems uses combined FDM/TDM (radio) for the air interface. Recall from Chapter 1 that, with pure FDM, the channel is partitioned into a number of frequency bands with each band devoted to a call. Also recall from Chapter 1 that, with pure TDM, time is partitioned into frames with each frame further partitioned into slots and each call being assigned the use of a particular
slot in the revolving frame. In combined FDM/TDM systems, the channel is partitioned into a number of frequency sub-bands; within each sub-band, time is partitioned into frames and slots. Thus, for a combined FDM/TDM system, if the channel is partitioned into *F* sub-bands and time is partitioned into *T* slots, thenthe channel wIll be able to support *F.T* simultaneous calls. Recall that we saw in Section 5.3.4 that cable access networks also use a combined FDM/TDM approach. GSM systems consist of 200-kHz frequency bands with each band supporting eight
TDM calls. GSM encodes speech at 13 kbps and 12.2 kbps. A GSM network's **base station controller (BSC)** will typically service several tens of base transceiver stations. The role of the BSC is to allocate BTS radio channels to mobile subscribers, perform **paging** (finding the cell in which a mobile user is resident), and perform handoff of mobile users. The base station controller and its controlled base transceiver stations collectively constitute a GSM **base station system (BSS)**.



Mobile switching center (MSC) plays the central role in user authorization and accounting (e.g., determining whether a mobile device is allowed to connect to the cellular network), call establishment and teardown, and handoff. A single MSC will typically contain up to five BSCs, resulting in approximately 200K subscribers per MSC. A cellular provider's network will have a number of MSCs, with special MSCs known as gateway MSCs connecting the provider's cellular network to the larger public telephone network.

3G Cellular Data Networks: Extending the Internet to Cellular Subscribers

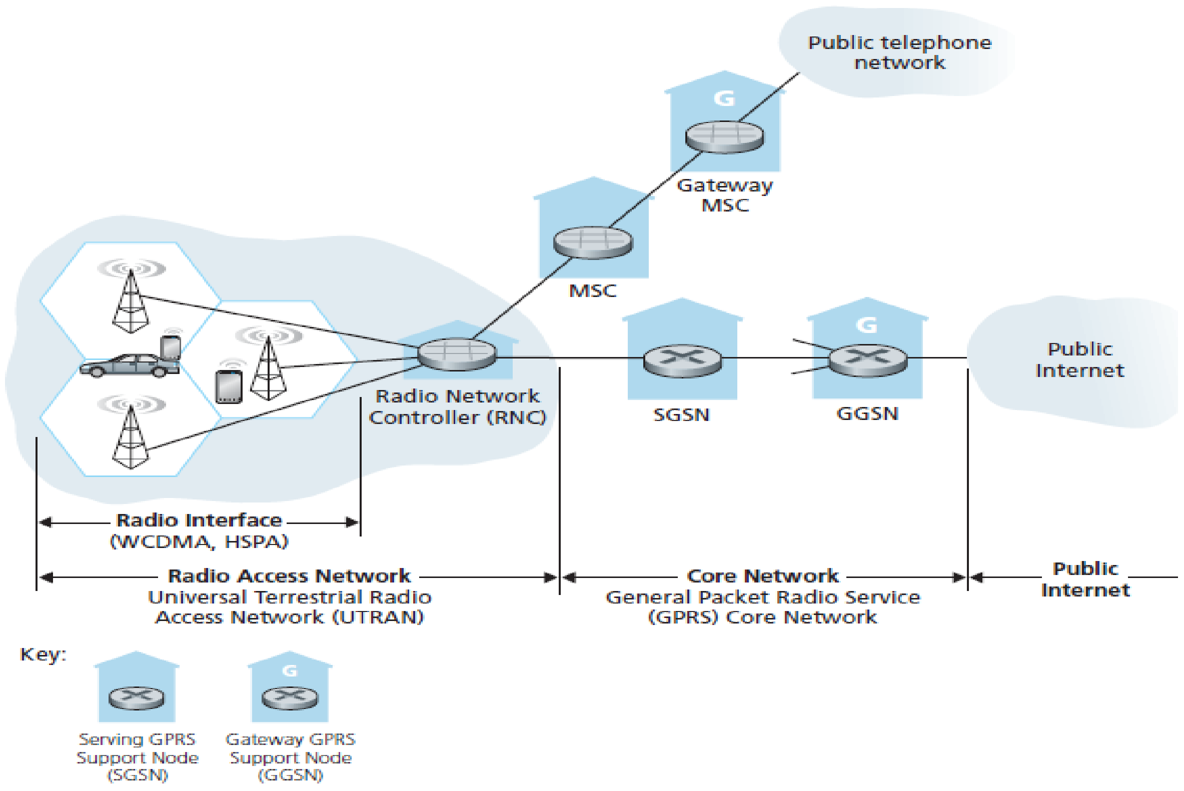
Focused on connecting cellular voice users to the public telephone network. But, of course, when we're on the go, we'd also like to read email, access the Web, get location-dependent services (e.g., maps and restaurant recommendations) and perhaps even watch streaming video. To do this, our smartphone will need to run a full TCP/IP protocol stack (including the physical link, network, transport, and application layers) and connect into the Internet via the cellular data network. The topic of cellular data networks is a rather bewildering collection of competing and ever-evolving standards as one generation (and half-generation) succeeds the former and introduces new technologies and services with new acronyms. To make matters worse, there's no single official body that sets requirements for 2.5G, 3G, 3.5G, or 4G technologies, making it hard to sort out the differences among competing standards.

3G Core Network

The 3G core cellular data network connects radio access networks to the public Internet. The core network interoperates with components of the existing cellular voice network (in particular, the MSC) that we previously encountered in Figure 6.18. Given the considerable amount of existing infrastructure (and profitable services!) in the existing cellular voice network, the approach taken by the designers of 3G data services is clear: *leave the existing core GSM cellular voice network untouched,adding additional cellular data functionality in parallel to the existing cellular voice network*. The alternative—integrating new data services directly into the core of theexisting cellular voice network, where we discussed integrating new (IPv6) and legacy (IPv4) technologies in the Internet.

There are two types of nodes in the 3G core network: **Serving GPRS Support Nodes (SGSNs)** and **Gateway GPRS Support Nodes (GGSNs)**. (GPRS stands for Generalized Packet Radio Service, an early cellular data service in 2G networks; here we discuss the evolved version of GPRS in 3G networks). An SGSN is responsible for delivering datagrams to/from the mobile nodes in the radio access network to which the SGSN is attached. The SGSN interacts with the cellular voice network's MSC for that area, providing user authorization and handoff, maintaining location (cell) information about active mobile nodes, and performing datagram forwarding between mobile nodes in the radio access network and a GGSN. The GGSN acts as a gateway, connecting multiple SGSNs into the larger Internet. A GGSN is thus the last piece of 3G infrastructure that a datagram originating at a mobile node encounters before entering the larger Internet. To the outside world, the GGSN looks like any other gateway router; the mobility

3G Radio Access Network: The Wireless Edge The 3G **radio access network** is the wireless first-hop network that we see as a 3G user. The **Radio Network Controller (RNC)** typically controls several cell base transceiver stations similar to the base stations that we encountered in 2G systems (but officially known in 3G UMTS parlance as a "Node Bs"—a rather non-descriptive name!). Each cell's wireless link operates between the mobile nodes and a base transceiver station, just as in 2G networks. The RNC connectsto both the circuit-switched cellular voice network via an MSC, and to the packet-switched Internet via an SGSN. of the 3G nodes within the GGSN's network is hidden from the outside world behind the GGSN. Thus, while 3G cellular voice and cellular data services use different core networks, they share a common first/last-hop radio access network.A significant change in 3G UMTS over 2G networks is that rather than using GSM's FDMA/TDMA scheme, UMTS uses a CDMA technique known as Direct Sequence Wideband CDMA (DS-WCDMA) [Dahlman 1998] within TDMAslots; TDMA slots, in turn, are available on multiple frequencies—an interesting use of all three dedicated channel-sharing approaches that we earlier identified inChapter 5 and similar to the approach taken in wired cable access networks (see Section


Public telephone network
Gateway MSC
MSC
Radio Network Controller (RNC)
SGSN
GGSN
Public Internet
Radio Interface (WCDMA, HSPA)
Radio Access Network Universal Terrestrial Radio Access Network (UTRAN)
Core Network General Packet Radio Service (GPRS) Core Network
Public Internet
Key:
Serving GPRS Support Node (SGSN)
Gateway GPRS Support Node (GGSN)

5.3.4).

This change requires a new 3G cellular wireless-access network operating in parallel with the 2G BSS radio network shown in Figure 6.19. The data service associated with the WCDMA specification is known as HSP (High Speed Packet Access) and promises downlink data rates of up to 14 Mbps. Details regarding 3G networks can be found at the 3rd Generation Partnership Project (3GPP) Web site [3GPP 2012]. With 3G systems now being deployed worldwide, can 4G systems be far behind?Certainly not! Indeed, the design, early testing, and initial deployment of 4G systems are already underway. The 4G Long-Term Evolution (LTE) standard put forward by the 3GPP has two important innovations over 3G systems:

• **Evolved Packet Core (EPC)** [3GPP Network Architecture 2012]. The EPC is a simplified all-IP core network that unifies the separate circuit-switched cellular voice network and the packet-switched cellular data network shown in Figure 6.19. It is an “all-IP” network in that both voice and data will be carried in IP datagrams. As we’ve seen in Chapter 4 and will study in more detail in Chapter 7, IP’s “best effort” service model is not inherently well-suited to the stringent performance requirements of Voice-over-IP (VoIP) traffic unless network resources are carefully managed to avoid (rather than react to) congestion. Thus, a key task of the EPC is to manage network resources to provide this high quality of service. The EPC also makes a clear separation
between the network control and user data planes, with many of the mobility support features that we will study in Section 6.7 being implemented in the control plane. The EPC allows multiple types of radio access networks, including legacy 2G and 3G radio access networks, to attach to the core network. Two very readable introductions to the EPC are [Motorola 2007; Alcatel-Lucent 2009].

• **LTE Radio Access Network.** LTE uses a combination of frequency division multiplexing and time division multiplexing on the downstream channel, known as orthogonal frequency division multiplexing (OFDM) [Rohde 2008; Ericsson 2011]. (The term “orthogonal” comes from the fact the signals being sent on different frequency channels are created so that they interfere very little with each other, even when channel frequencies are tightly spaced). In LTE, each active mobile node is allocated one or more 0.5 ms

time slots in one or more of the channel frequencies. Figure 6.20 shows an allocation of eight time slots over four frequencies. By being allocated increasingly more time slots (whether on the same frequency or on different frequencies), a mobile node is able to achieve increasingly higher transmission rates. Slot (re)allocation among mobile nodes can be performed as often as once every millisecond. Different modulation schemes can also be used to change the transmission rate; see our earlier discussion of Figure 6.3 and dynamic selection of modulation schemes in WiFi networks. Another innovation in the LTE radio network is the use of sophisticated multiple-input, multiple output (MIMO) antennas. The
maximum data rate for an LTE user is 100 Mbps in the downstream direction and 50 Mbps in the upstream direction, when using 20 MHz worth of wireless spectrum. The particular allocation of time slots to mobile nodes is not mandated by the LTE standard.

Instead, the decision of which mobile nodes will be allowed to transmit in a given time slot on a given frequency is determined by the scheduling algorithms provided by the LTE equipment vendor and/or the network operator. With opportunistic scheduling [Bender 2000; Kolding 2003; Kulkarni 2005], matching the physical-layer protocol to the channel conditions between the sender and receiver and choosing the receivers to which packets will be sent based on channel conditions allow the radio network controller to make best use of the wireless medium. In addition, user priorities and contracted levels of service (e.g., silver, gold, or platinum) can be used in scheduling downstream packet transmissions. In addition to the LTE capabilities described above, LTE-Advanced allows for downstream bandwidths of hundreds of Mbps by allocating aggregated channels to a mobile node [Akyildiz 2010]. An additional 4G wireless technology—WiMAX (World Interoperability for Microwave Access)—is a family of IEEE 802.16 standards that differ significantly from LTE. Whether LTE or WiMAX becomes the 4G technology of choice is still to be seen, but at the time of this writing (spring 2012), LTE appears to have significantly6.5

4.2 Mobility Management: Principles

Having covered the *wireless* nature of the communication links in a wireless network, it's now time to turn our attention to the *mobility* that these wireless links enable. In the broadest sense, a mobile node is one that changes its point of attachment into the network over time. Because the term *mobility* has taken on many meanings in both the computer and telephony worlds, it will serve us well first to consider several dimensions of mobility in some detail. • *From the network layer's standpoint, how mobile is a user?* A physically mobile user will present a very different set of challenges to the network layer, depending on how he or she moves between points of attachment to the network. At one end of the spectrum in Figure 6.21, a user may carry a laptop with a wireless network interface card around in a building. As we saw in Section 6.3.4, this user is *not* mobile from a network-layer perspective. Moreover, if the user associates with the same access point regardless of location, the user is not even mobile from the perspective of the link layer. At the other end of the spectrum, consider the user zooming along the autobahn in a BMW at 150 kilometers per hour, passing through multiple wireless access networks and wanting to maintain an uninterrupted TCP connection to a remote application throughout the trip. This user is *definitely* mobile! In between these extremes is a user who takes a laptop from one location (e.g., office or dormitory) into another (e.g., coffeeshop, classroom) and wants to connect into the network in the new location. This user is also mobile (although less so than the BMW driver!) but does not need to maintain an ongoing connection while moving between points of attachment to the network. Figure 6.21 illustrates this spectrum of user mobility from the network layer's perspective.
How important is it for the mobile node's address to always remain the same? With mobile telephony, your phone number—essentially the network-layer address of your phone—remains the same as you travel from one provider's mobile phone network to another. Must a laptop similarly maintain the same IP address while moving between IP networks?the answer to this question will depend strongly on the applications being run.For the BMW driver who wants to maintain an uninterrupted TCP connection to a remote

application while zipping along the autobahn, it would be convenient to maintain the same IP address. Recall from Chapter 3 that an Internet application needs to know the IP address and port number of the remote entity with which it is communicating. If a mobile entity is able to maintain its IP address as it moves, mobility becomes invisible from the application standpoint.

There is great value to this transparency—an application need not be concerned with a potentially changing IP address, and the same application code serves mobile and nonmobile connections alike. We'll see in the following section that mobile IP provides this transparency, allowing a mobile node to maintain its permanent IP address while moving among networks. On the other hand, a less glamorous mobile user might simply want to turn off an office laptop, bring that laptop home, power up, and work from home. If the laptop functions primarily as a client in client-server applications (e.g., send/read e-mail, browse the Web, Telnet to a remote host) from home, the particular IP address used by the laptop is not that important. In particular, one could get by fine with an address that is temporarily allocated to the laptop by the ISP serving the home.

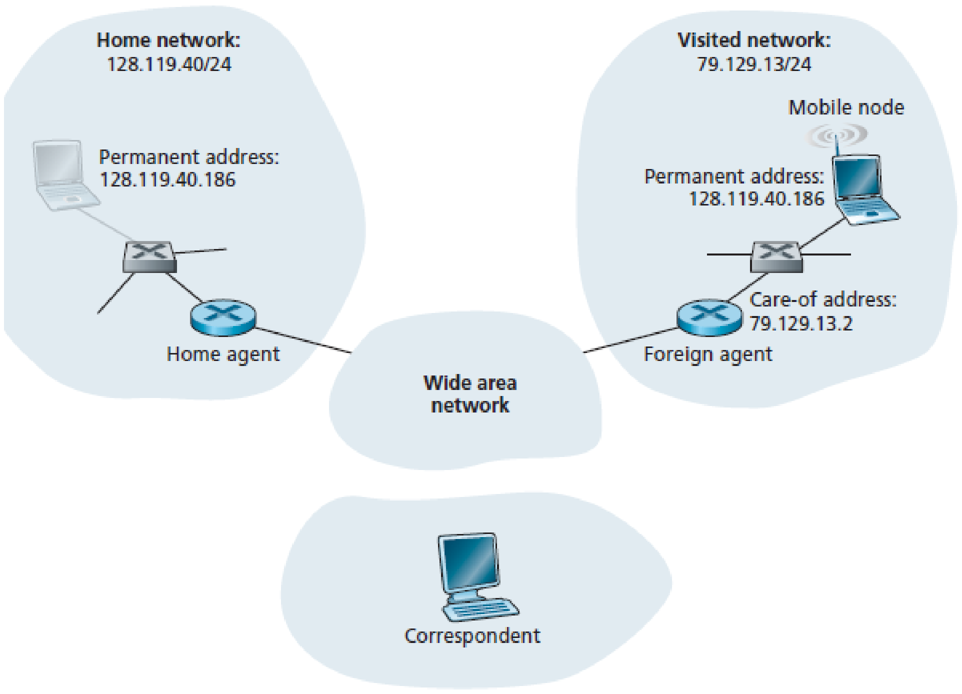
. • *What supporting wired infrastructure is available?* In all of our scenarios above, we've implicitly assumed that there is a fixed infrastructure to which the mobile user can connect—for example, the home's ISP network, the wireless access network in the office, or the wireless access networks lining the autobahn. What if no such infrastructure exists? If two users are within communication proximity of each other, can they establish a network connection in the absence of any other network-layer infrastructure? Ad hoc networking provides precisely these capabilities. This rapidly developing area is at the cutting edge of mobile networking research and is beyond the scope of this book. [Perkins 2000] and the IETF Mobile Ad Hoc Network (manet) working group Web pages [manet 2012] provide thorough treatments of the subject. In order to illustrate the issues involved in allowing a mobile user to maintain ongoing connections while moving between networks, let's consider a human analogy. A twenty-something adult moving out of the family home becomes mobile, living in a series of dormitories and/or apartments, and often changing addresses. If an old friend wants to get in touch, how can that friend find the addressmobile adult will often register his or her current address with the family (if for no other reason than so that the parents can send money to help pay the rent!). The family home, with its permanent address, becomes that one place that others can go as a first step in communicating with the mobile adult. Later communication from the friend may be either indirect (for example, with mail being sent first to the parents' home and then forwarded to the mobile adult) or direct (for example, with the friend using the address obtained from the parents to send mail directly to
her mobile friend). In a network setting, the permanent home of a mobile node (such as a laptop
or smartphone) is k nown as the **home network**, and the entity within the home network that performs the mobility management functions discussed below on behalf of the mobile node is known as the **home agent**. The network in which the mobile node is currently residing is known as the **foreign** (or **visited**) **network**, and the entity within the foreign network that helps the mobile node with the mobility management functions discussed below is known as a **foreign agent**. For mobile professionals, their home network might likely be their company network, while the visited network might be the network of a colleague they are visiting. A **correspondent** is the entity wishing to communicate with the mobile node. Figure 6.22 illustrates these concepts, as well as addressing concepts considered below. In Figure 6.22, note that agents are shown as being collocated with routers (e.g., as processes running on routers), but alternatively they could be executing on other hosts or servers in the network.

4.3 **Addressing**

We noted above that in order for user mobility to be transparent to network applications, it is desirable for a mobile node to keep its address as it moves from one network to another. When a mobile node is resident in a foreign network, all traffic addressed to the node's permanent address now needs to be routed to the foreign network. How can this be done? One option is for the foreign network to advertise to all other networks that the mobile node is resident in its network. This could be via the usual exchange of intradomain and interdomain routing information and would require few changes to the existing routing infrastructure. The foreign network could simply advertise to its neighbors that it has a highly specific route to the mobile node's permanent address (that is, essentially inform other networks that it has the correct path for routing datagrams to the mobile node's permanent address; see Section 4.4).

These neighbors would then propagate this routing information throughout the network as part of the normal procedure of updating routing information and forwarding tables. When the mobile node leaves one foreign network and joins another, the new foreign network would advertise a new, highly specific route to the mobile node, and the old foreign network would withdraw its routing information regarding the mobile node. This solves two problems at once, and it does so without making significant changes to the network-layer infrastructure. Other networks know the location of the mobile node, and it is easy to route datagrams to the mobile node, since the forwarding tables will direct datagrams to the foreign network. A significant drawback, however, is that of scalability. If mobility management were to be the responsibility of network routers, the routers would have to maintain forwarding table entries for potentially millions of mobile nodes, and update these entries as nodes move. Some additional drawbacks at the end of this chapter. An alternative approach (and one that has been adopted in practice) is to push mobility functionality from the network core to the network edge—a recurring theme in our study of Internet architecture. A natural way to do this is via the mobile node's home network. In much the same way that parents of the mobile twenty something track their child's location, the home agent in the mobile node's home network can track the foreign network in which the A protocol between the mobile node (or a foreign agent representing the mobile node) and the home agent will certainly be needed to update the mobile node's location. Let's now consider the foreign agent in more detail. The conceptually simplest

approach, mobile node resides. are explored in the problems



shown in above Figure , is to locate foreign agents at the edge routers in the foreign network. One role of the foreign agent is to create a so-called **care-of address (COA)** for the mobile node, with the network portion of the COA matching that of the foreign network. There are thus two addresses associated with a mobile node, its **permanent address** (analogous to our mobile youth's family's home address) and its COA, sometimes known as a **foreign address** (analogous to the address of the house in which our mobile youth is currently residing). In the example in Figure 6.22, the permanent address of the mobile node is 128.119.40.186. When visiting network 79.129.13/24, the mobile node has a COA of 79.129.13.2. A second role of the foreign agent is to inform the home agent that the mobile node is resident in its (the foreign agent's) network and has the given COA. We'll see shortly that the COA will be used to "reroute" datagrams to the mobile node via its foreign agent. Although we have separated the functionality of the mobile node and the foreign agent, it is worth noting that the mobile node can also assume the responsibilities of the foreign agent. For example, the mobile node could obtain a COA in the foreign network (for example, using a protocol such as DHCP) and itself inform the agent of its COA.

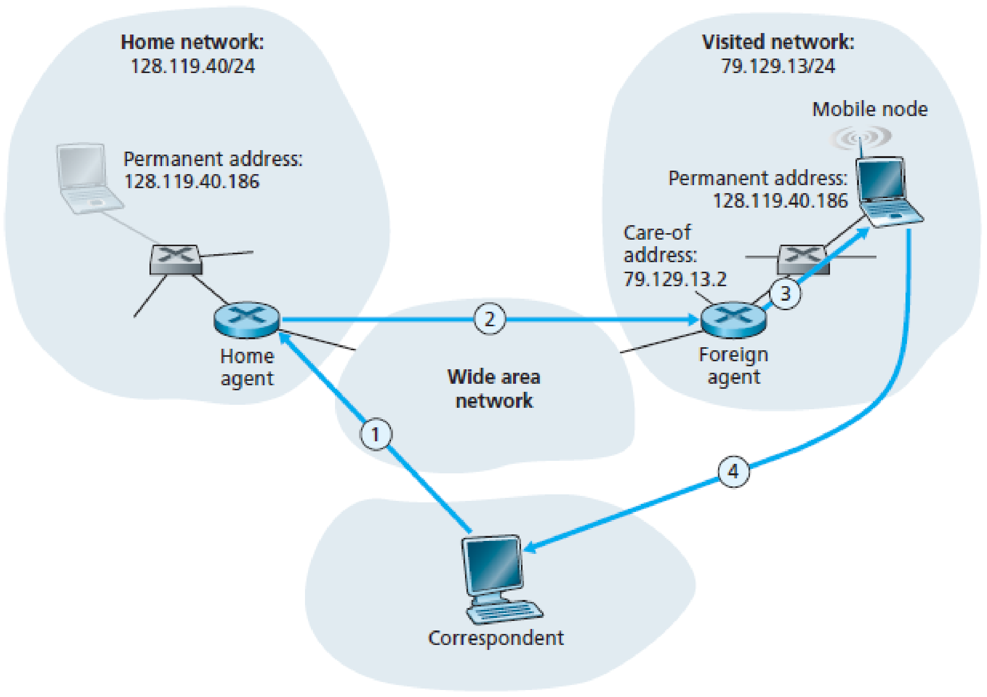
Routing to a Mobile Node

We have now seen how a mobile node obtains a COA and how the home agent can be informed of that address. But having the home agent know the COA solves only part of the problem. How should datagrams be addressed and forwarded to the mobile node? Since only the home agent (and not network-wide routers) knows the location of the mobile node, it will no longer suffice to simply address a datagram to the mobile node's permanent address and send it into the network-layer infrastructure. Something more must be done. Two approaches can be identified, which we
will refer to as indirect and direct routing.

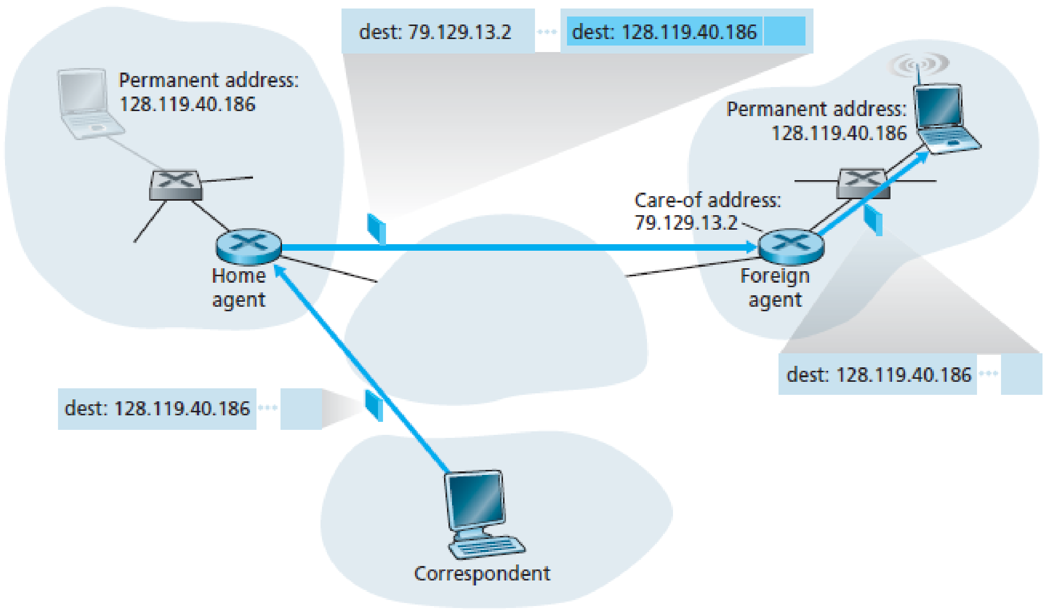
Indirect Routing to a Mobile Node

Let's first consider a correspondent that wants to send a datagram to a mobile node. In the **indirect routing** approach, the correspondent simply addresses the datagram to the mobile node's permanent address and sends the datagram into the network, blissfully unaware of whether the mobile node is resident in its home network or is visiting a foreign network; mobility is thus completely transparent to the correspondent. Such datagrams are first routed, as usual, to the mobile node's home network. This is illustrated in step 1 in Figure 6.23. Let's now turn our attention to the home agent. In addition to being responsible for interacting with a foreign agent to track the mobile node's COA, the home agenthas another very important function. Its second job is to be on the lookout for arriving datagrams addressed to nodes whose home network is that of the home agent but that are currently resident in a foreign network.

The home agent intercepts these datagrams and then forwards them to a mobile node in a two-step process. The datagram is first forwarded to the foreign agent, using the mobile node's COA (step 2 in Figure 6.23), and then forwarded from the foreign agent to the mobile node (step 3 in Figure 6.23). It is instructive to consider this rerouting in more detail. The home agent will need to address the datagram using the mobile node's COA, so that the network layer will route the datagram to the foreign network. On the other hand, it is desirable to leave the correspondent's datagram intact, since the application receiving the datagram should be unaware that the datagram was forwarded via the home agent.



Both goals can be satisfied by having the home agent **encapsulate** the correspondent's original complete datagram within a new (larger) datagram. This larger datagram is addressed and delivered to the mobile node's COA. The foreign agent, who "owns" the COA, will receive and decapsulate the datagram—that is, remove the correspondent's original datagram from within the larger encapsulating datagram and forward (step 3 in Figure 6.23) the original datagram to the mobile node. Figure 6.24 shows a correspondent's original datagram being sent to the home network, an encapsulated datagram being sent to the foreign agent, and the original datagram being delivered to the mobile node. The sharp reader will note that the encapsulation/decapsulation described here is identical to the notion of tunneling
discussed in Chapter 4 in the context of IP multicast and IPv6. Let's next consider how a mobile node sends datagrams to a correspondent. This is quite simple, as the mobile node can address its datagram *directly* to the correspondent (using its own permanent address as the source address, and the correspondent's address as the destination address). Since the mobile node knows the
correspondent's address , there is no need to route the datagram back through the home agent. This is shown as step 4 in Figure .



Let’s summarize our discussion of indirect routing by listing the new networklayer functionality required to support mobility. *A mobile-node–to–foreign-agent protocol.* The mobile node will register with the foreign agent when attaching to the foreign network. Similarly, a mobile node will deregister with the foreign agent when it leaves the foreign network.

• *A foreign-agent–to–home-agent registration protocol.* The foreign agent will register the mobile node’s COA with the home agent. A foreign agent need not explicitly deregister a COA when a mobile node leaves its network, because the subsequent registration of a new COA, when the mobile node moves to a new network, will take care of this.
• *A home-agent datagram encapsulation protocol.* Encapsulation and forwarding of the correspondent’s original datagram within a datagram addressed to the COA.
• *A foreign-agent decapsulation protocol.* Extraction of the correspondent’s original datagram from the encapsulating datagram, and the forwarding of the original datagram to the mobile node.
The previous discussion provides all the pieces—foreign agents, the home agent, and indirect forwarding—needed for a mobile node to maintain an ongoing connection while moving among networks. As an example of how these pieces fit together, assume the mobile node is attached to foreign network A, has registered a COA in network A with its home agent, and is receiving datagrams that are being indirectly routed through its home agent. The mobile node now moves to foreign network B and registers with the foreign agent in network B, which informs the home agent of the mobile node’s new COA. From this point on, the home agent will reroute datagrams to foreign network B. As far as a correspondent is concerned, mobility is transparent—datagrams are routed via the same home agent both before and after the move. As far as the home agent is concerned, there is no disruption in the flow of datagrams—arriving datagrams are first forwarded to foreign network A; after the change in COA, datagrams are forwarded to foreign network B. But will the mobile node see an interrupted flow of datagrams as it moves between networks? As long as the time between the mobile node’s disconnection from network A (at which point it can no longer receive datagrams via A) and its attachment to network B (at which point it will register a new COA with its home agent) is small, few datagrams will be lost. Recall from Chapter 3 that end-to-end connections can suffer datagram loss due to network congestion. Hence occasional datagram loss within a connection when a node moves between networks is by no means a catastrophic problem. If loss-free communication is required,

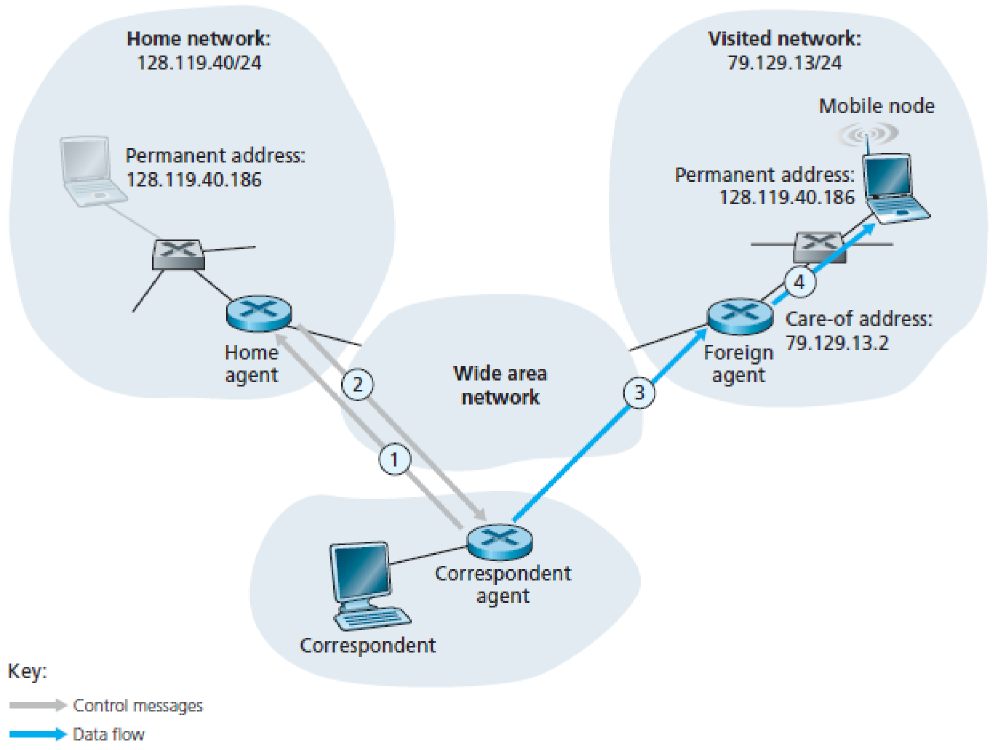
upper-layer mechanisms will recover from datagram loss, whether such loss results from network congestion or from user mobility.

Direct Routing to a Mobile Node

The indirect routing approach illustrated in Figure 6.23 suffers from inefficiency known as the **triangle routing problem**—datagrams addressed to the mobile node must be routed first to the home agent and then to the foreign network, even when a much more efficient route exists between the correspondent and the mobile node. In the worst case, imagine a mobile user who is visiting the foreign network of a colleague. The two are sitting side by side and exchanging data over the network. Datagrams from the correspondent (in this case the colleague of the visitor) are routed to the mobile user's home agent and then back again to the foreign network! **Direct routing** overcomes the inefficiency of triangle routing, but does so at the cost of additional complexity. In the direct routing approach, a **correspondent agent** in the correspondent's network first learns the COA of the mobile node. This can be done by having the correspondent agent query the home agent, assuming that (as in the case of indirect routing) the mobile node has an up-to-date value for its COA registered with its home agent. It is also possible for the correspondent itself to perform the function of the correspondent agent, just as a mobile node could perform the function of the foreign agent. This is shown as steps 1 and 2 in Figure 6.25. The correspondent agent then tunnels datagrams directly to the mobile node's COA, in a manner analogous to the tunneling performed by the home agent, steps 3 and 4 in Figure 6.25.While direct routing overcomes the triangle routing problem, it introduces two important additional challenges:

• A **mobile-user location protocol** is needed for the correspondent agent to query the home agent to obtain the mobile node's COA (steps 1 and 2 in Figure 6.25).

• When the mobile node moves from one foreign network to another, how will data now be forwarded to the new foreign network? In the case of indirect routing, this problem was easily solved by updating the COA maintained by the home agent.

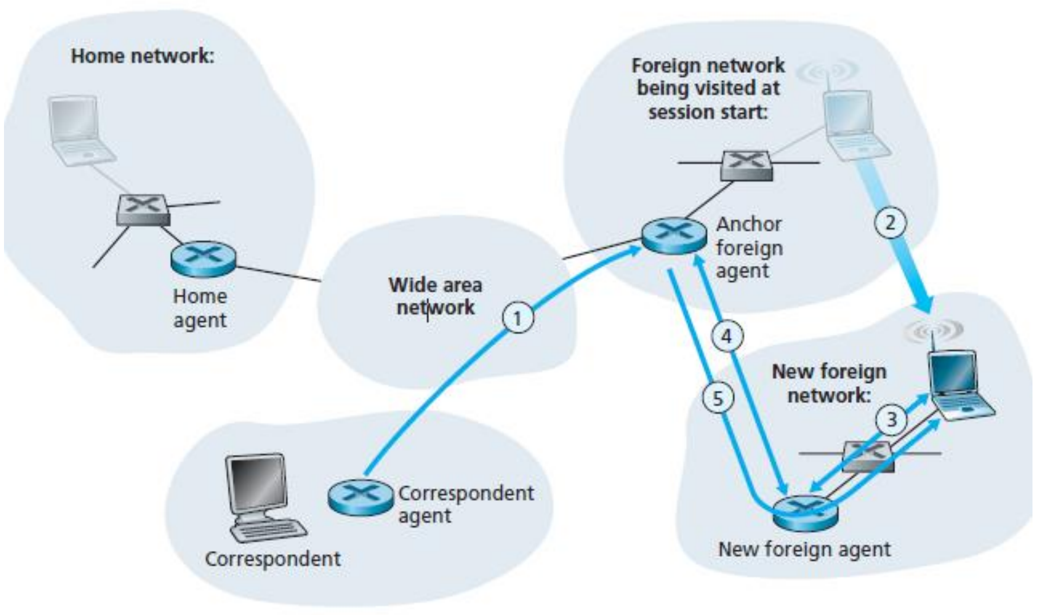


However, with direct routing, the home agent is queried for the COA by the correspondent agent only once, at the beginning of the session. Thus, updating the COA at the home agent, while necessary, will not be enough to solve the problem of routing data to the mobile node's new foreign network. One solution would be to create a new protocol to notify the correspondent of the changing COA. An alternate solution, and one that we'll see adopted in practice in GSM networks, works as follows. Suppose data is currently being forwarded to the mobile node in the foreign network where the mobile node was located when the session first started (step 1 in Figure 6.26). We'll identify the foreign agent in that foreign network where the mobile node was first found as the **anchor foreign agent**. When the mobile node moves to a new foreign network (step 2 in Figure 6.26), the mobile node registers with the new foreign agent (step 3), and the new foreign agent provides the anchor foreign agent with the mobile node's new COA (step 4).

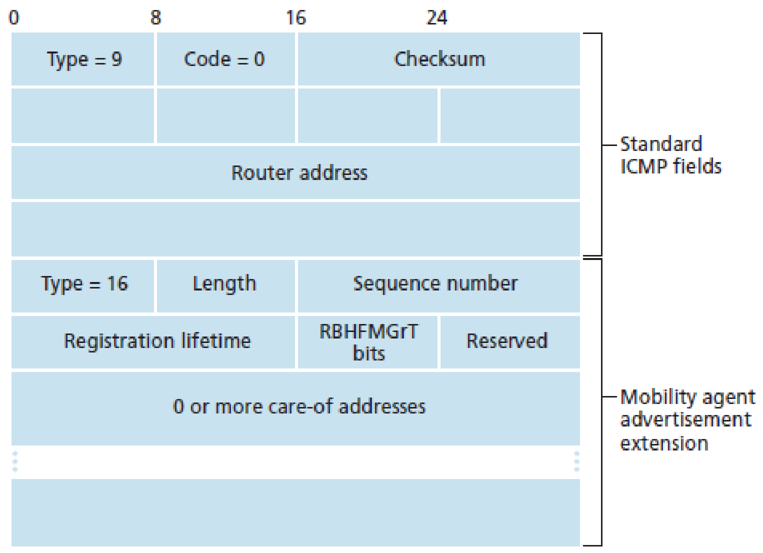
When the anchor foreign agent receives an encapsulated datagram for a departed mobile node, it can then re-encapsulate the datagram and forward it to the mobile node (step 5) using the new COA. If the mobile node later moves yet again to a new foreign network, the foreign agent in that new visited network would then contact the anchor foreign agent in order to set up forwarding to this new foreign network.

4.4 **Mobile IP**

The Internet architecture and protocols for supporting mobility, collectively known as mobile IP, are defined primarily in RFC 5944 for IPv4. Mobile IP is a flexible standard, supporting many different modes of operation (for example, operation with or without a foreign agent), multiple ways for agents and mobile nodes to discover each other, use of single or multiple COAs, and multiple forms of encapsulation. As such, mobile IP is a complex standard, and would require an entire book to describe in detail; indeed one such book is [Perkins 1998b]. Our modest goal here is to provide an overview of the most important aspects of mobile IP and to illustrate its use in a few common-case scenarios.



The mobile IP architecture contains many of the elements we have considered above, including the concepts of home agents, foreign agents, care-of addresses, and
encapsulation/decapsulation. The current standard [RFC 5944] specifies the use of indirect routing to the



mobile node.
The mobile IP standard consists of three main pieces:
• *Agent discovery.* Mobile IP defines the protocols used by a home or foreign agent to advertise its services to mobile nodes, and protocols for mobile nodes to solicit the services of a foreign or home agent. *Registration with the home agent.* Mobile IP defines the protocols used by the mobile node and/or foreign agent to register and deregister COAs with a mobile node's home agent.
• *Indirect routing of datagrams.* The standard also defines the manner in which datagrams are forwarded to mobile nodes by a home agent, including rules for forwarding datagrams, rules for handling error conditions, and several forms of encapsulation [RFC 2003, RFC 2004]. Security considerations are prominent throughout the mobile IP standard. For example, authentication of a mobile node is clearly

needed to ensure that a malicious user does not register a bogus care-of address with a home agent, which could cause all datagrams addressed to an IP address to be redirected to the malicious user. Mobile IP achieves security using many of the mechanisms that

Agent Discovery

A mobile IP node arriving to a new network, whether attaching to a foreign network or returning to its home network, must learn the identity of the corresponding foreign or home agent. Indeed it is the discovery of a new foreign agent, with a new network address, that allows the network layer in a mobile node to learn that it has moved into a new foreign network. This process is known as **agent discovery**. Agent discovery can be accomplished in one of two ways: via agent advertisement or via agent solicitation. With **agent advertisement**, a foreign or home agent advertises its services using an extension to the existing router discovery protocol [RFC 1256]. The agent periodically broadcasts an ICMP message with a type field of 9 (router discovery) on all links to which it is connected. The router discovery message contains the IP address of the router (that is, the agent), thus allowing a mobile node to learn the agent's IP address. The router discovery message also contains a mobility agent advertisement extension that contains additional information needed by the mobile node. Among the more important fields in the extension are the following:

• *Home agent bit (H).* Indicates that the agent is a home agent for the network in which it resides.

• *Foreign agent bit (F).* Indicates that the agent is a foreign agent for the network in which it resides.

• *Registration required bit (R).* Indicates that a mobile user in this network *must* register with a foreign agent. In particular, a mobile user cannot obtain a care of address in the foreign network (for example, using DHCP) functionality of the foreign agent for itself, without registering with the foreign agent.

• *M, G encapsulation bits.* Indicate whether a form of encapsulation other than IPin- IP encapsulation will be used.

• *Care-of address (COA) fields.* A list of one or more care-of addresses provided by the foreign agent. In our example below, the COA will be associated with the foreign agent, who will receive datagrams sent to the COA and then forward them to the appropriate mobile node. The mobile user will select one of these addresses as its COA when registering with its home agent.

Figure 6.27 illustrates some of the key fields in the agent advertisement message. With **agent solicitation**, a mobile node wanting to learn about agents without waiting to receive an agent advertisement can broadcast an agent solicitation message, which is simply an ICMP message with type value 10. An agent receiving the solicitation wills unicast an agent advertisement directly to the mobile node, which can then proceed as if it had received an unsolicited advertisement.

Registration with the Home Agent

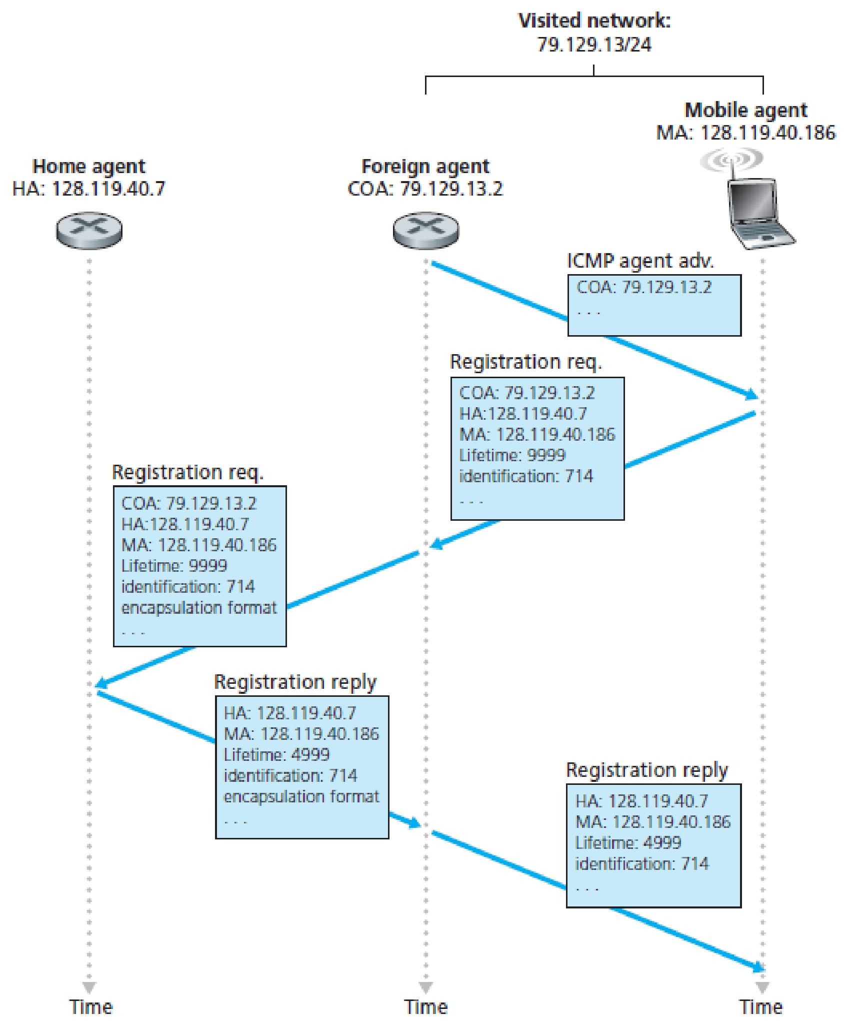
Once a mobile IP node has received a COA, that address must be registered with the home agent. This can be done either via the foreign agent (who then registers the COA with the home agent) or directly by the mobile IP node itself. We consider the former case below. Four steps are involved.

1. Following the receipt of a foreign agent advertisement, a mobile node sends a mobile IP registration message to the foreign agent. The registration message is carried within a UDP datagram and sent to port 434. The registration message carries a COA advertised by the foreign agent, the address of the home agent (HA), the permanent address of the mobile node (MA), the requested lifetime of the registration, and a 64-bit registration identification. The requested registration lifetime is the number of seconds that the registration is to be valid. If the registration is not renewed at the home agent within the specified lifetime, the registration will become invalid. The registration identifier acts like a sequence number and serves to match a received registration reply with a registration request, as discussed below.

2. The foreign agent receives the registration message and records the mobile node's permanent IP address. The foreign agent now knows that it should be looking for datagrams containing an encapsulated datagram whose destination address matches the permanent address of the mobile node. The foreign agent then sends

a mobile IP registration message (again, within a UDP datagram) to port 434 of the home agent. The message contains the COA, HA, MA, encapsulation format requested, requested registration lifetime, and registration identification.
3. The home agent receives the registration request and checks for authenticity and correctness. The home agent binds the mobile node's permanent IP address with the COA; in the future, datagrams arriving at the home agent and addressed to the mobile node will now be encapsulated and tunneled to the COA. The home agent sends a mobile IP registration reply containing the HA, MA, actual registration lifetime, and the registration identification of the datagrams sent to its permanent address. Figure 6.28 illustrates these steps. Note that the home agent specifies a lifetime that is smaller than the lifetime requested by the mobile node. A foreign agent need not explicitly deregister a COA when a mobile node leaves its network. This will occur automtically, when the mobile node moves to a new network (whether another foreign network or its home network) and registers a new COA.

Managing Mobility in Cellular Networks
Having examined how mobility is managed in IP networks, let's now turn our attention to networks with an even longer history of supporting mobility—cellular telephony networks. Whereas we focused on the first-hop wireless link in cellular networks in Section 6.4, we'll focus here on mobility, using the GSM cellular network architecture [Goodman 1997; Mouly 1992; Scourias 2012; Kaaranen 2001; Korhonen 2003; Turner 2012] as our case study, since it is a mature and widely deployed technology. As in the case of mobile IP, we'll see that a number of the fundamental principles we identified in Section 6.5 are embodied in GSM's network architecture. Like mobile IP, GSM adopts an indirect routing approach (see Section 6.5.2), first routing the correspondent's call to the mobile user's home network and from there to the visited network. In GSM terminology, the mobile users's home network is referred to as the mobilerequest that is being satisfied with this reply. 4. The foreign agent receives the registration reply and then forwards it to the mobile node. At this point, registration is complete, and the



user's **home public land mobile network (home PLMN)**. Since the PLMN acronym is a bit of a mouthful, and mindful of our quest to avoid an alphabet soup of acronyms, we'll refer to the GSM home PLMN simply as the **home network**. The home network is the cellular provider with which the mobile user has a subscription (i.e., the provider that bills the user for monthly cellular
service). The visited PLMN, which we'll refer to simply as the **visited network**, is the network in which the mobile user is currently residing. As in the case of mobile IP, the responsibilities of the home and visited networks are quite different.

• The home network maintains a database known as the **home location register** (**HLR**), which contains the permanent cell phone number and subscriber profile information for each of its subscribers. Importantly, the HLR also contains information about the current locations of these subscribers. That is, if a mobile user is currently roaming in another provider's cellular network, the HLR contains enough information to obtain (via a process we'll describe shortly) an address in the visited network to which a call to the mobile user should be routed. As we'll see, a special switch in the home network, known as the **Gateway Mobile services Switching Center (GMSC)** is contacted by a correspondent when a call is placed to a mobile user. Again, in our quest to avoid an alphabet soup of acronyms, we'll refer to the GMSC here by a more descriptive term, **home MSC**.

• The visited network maintains a database known as the **visitor location register (VLR)**. The VLR contains an entry for each mobile user that is *currently* in the portion of the network served by the VLR. VLR entries thus come and go as mobile users enter and leave the network. A VLR is usually co-located with the mobile switching center (MSC) that coordinates the setup of a call to and from the visited network.

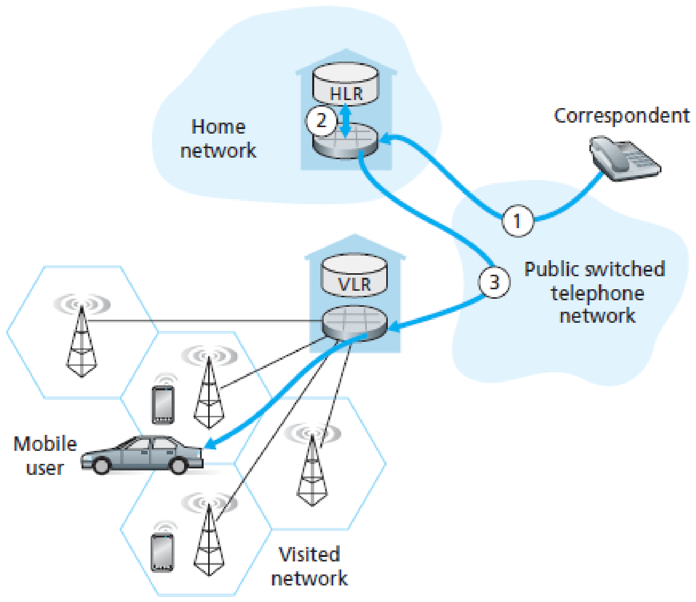
4.5 MANAGING MOBILITY IN CELLULAR NETWORKS

In practice, a provider's cellular network will serve as a home network for its subscribers and as a visited network for mobile users whose subscription is with a different cellular provider.

Routing Calls to a Mobile User

We're now in a position to describe how a call is placed to a mobile GSM user in a visited network. We'll consider a simple example below; more complex scenarios are described in [Mouly 1992]. The steps, as illustrated in Figure 6.29, are as follows:

The correspondent dials the mobile user's phone number. This number itself does not refer to a particular telephone line or location (after all, the phone number is fixed and the user is mobile!). The leading digits in the number are sufficient to globally identify the mobile's home network. The call is routed from the correspondent through the PSTN to the home MSC in the mobile's home network. This is the first leg of the call.

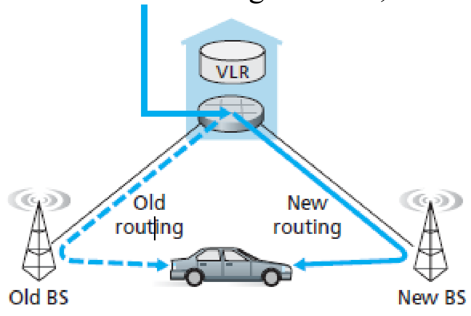


2. The home MSC receives the call and interrogates the HLR to determine the location of the mobile user. In the simplest case, the HLR returns the **mobilestation roaming number (MSRN)**, which we will refer to as the **roaming number**. Note that this number is different from the mobile's permanent phone number, which is associated with the mobile's home network. The roaming number is ephemeral: It is temporarily assigned to a mobile when it enters a visited network. The roaming number serves a role similar to that of the care-of address in mobile IP and, like the COA, is invisible to the correspondent and the mobile. If HLR does not have the roaming number, it returns the address of the VLR in the visited network. In this case (not shown in Figure), the home MSC will need to query the VLR to obtain the roaming number of the mobile node. But how does the HLR get the roaming number or the VLR address in the first place? What happens to these values when the mobile user moves to another visited network? We'll consider these important questions shortly.

3. Given the roaming number, the home MSC sets up the second leg of the call through the network to the MSC in the visited network. The call is completed, being routed from the correspondent to the home MSC, and from there to the visited MSC, and from there to the base station serving the mobile user. An unresolved question in step 2 is how the HLR obtains information about the location of the mobile user. When a mobile telephone is switched on or enters a part of a visited network that is covered by a new VLR, the mobile must register with the visited network. This is done through the exchange of signaling messages between the mobile and the VLR. The visited VLR, in turn, sends a location update request message to the mobile's HLR. This message informs the HLR of either the roaming number at which the mobile can be contacted, or the address of the VLR (which can then later be queried to obtain the mobile number). As part of this exchange, the VLR also obtains subscriber information from the HLR about the mobile and determines what services (if any) should be accorded the mobile user by the visited network.

Handoffs in GSM

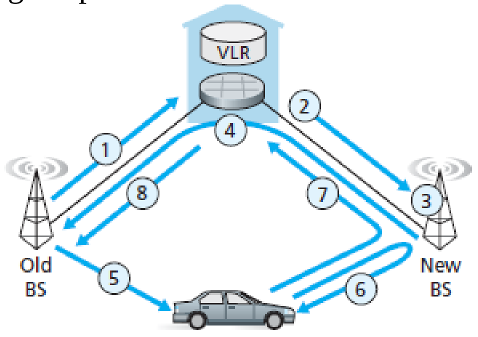
A **handoff** occurs when a mobile station changes its association from one base station to another during a call. As shown in Figure 6.30, a mobile's call is initially (before handoff) routed to the mobile



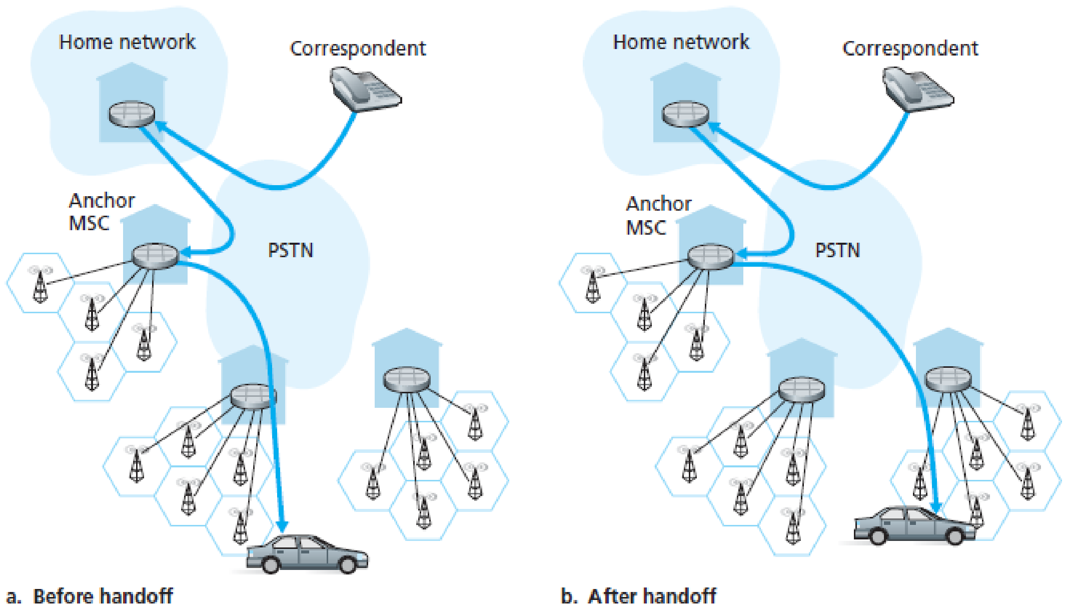
t

hrough one base station (which we'll refer to as the old base station), and after handoff is routed to the mobile through another base station (which we'll refer to as the new base station). Note that a handoff between base stations results not only in the mobile transmitting/receiving to/from a new base station, but also in the rerouting of the ongoing call from a switching point
within the network to the new base station. Let's initially assume that the old and new base stations share the same MSC, and that the rerouting occurs at this MSC. There may be several reasons for handoff to occur, including (1) the signal between the current base station and the mobile may have deteriorated to such an extent that the call is in danger of being dropped, and (2) a cell may have become overloaded, handling a large number of calls. This congestion may be alleviated by handing off mobiles to less congested nearby cells. While it is associated with a base station, a mobile periodically measures the strength of a beacon signal from its current base station as well as beacon signals from thearby base stations that it can "hear." These measurements are reported once or twice a second to the mobile's current base station. Handoff in GSM is initiated by the old base station based on these measurements, the current loads of mobiles in nearby cells, and other factors [Mouly 1992]. The GSM standard does not specify the specific algorithm to be used by a base station to determine whether or not to perform handoff.
Figure 6.31 illustrates the steps involved when a base station does decide to hand off a mobile user:
1. The old base station (BS) informs the visited MSC that a handoff is to be performed and the BS (or possible set of BSs) to which the mobile is to be handed off.
2. The visited MSC initiates path setup to the new BS, allocating the resources needed to carry the rerouted call, and signaling the new BS that a handoff is about to occur.
3. The new BS allocates and activates a radio channel for use by the mobile.
4. The new BS signals back to the visited MSC and the old BS that the visited- MSC-to-new-BS path has been established and that the mobile should be informed of the impending handoff. The new BS provides all of the information that the mobile will need to associate with the new BS.

5. The mobile is informed that it should perform a handoff. Note that up until this point, the mobile has been blissfully unaware that the network has been laying the groundwork (e.g., allocating a channel in the new BS and allocating a path from the visited MSC to the new BS) for a handoff.
6. The mobile and the new BS exchange one or more messages to fully activate the new channel in the new BS.
7. The mobile sends a handoff complete message to the new BS, which is forwarded up to the visited MSC. The visited MSC then reroutes the ongoing call to the mobile via the new BS.
8. The resources allocated along the path to the old BS are then released.



Let’s conclude our discussion of handoff by considering what happens when the mobile moves to a BS that is associated with a *different* MSC than the old BS, and what happens when this inter-MSC handoff occurs more than once. As shown in Figure 6.32, GSM defines the notion of an **anchor MSC**. The anchor MSC is the MSC visited by the mobile when a call first begins; the anchor MSC thus remains unchanged during the call. Throughout the call’s duration and regardless of the number of inter-MSC transfers performed by the mobile, the call is routed from the home MSC to the anchor MSC, and then from the anchor MSC to the visited MSC where the mobile is currently located. When a mobile moves from the coverage area of one MSC to another, the ongoing call is rerouted from the anchor MSC to the new visited MSC containing the new base station. Thus, at all times there are at most three MSCs (the home MSC, the anchor MSC, and the visited MSC) between the correspondent and the mobile. Figure 6.32 illustrates the routing of a call among the MSCs visited by a mobile user. Rather than maintaining a single MSC hop from the anchor MSC to the current MSC, an alternat ive approach would have been to simply chain the MSCs visited by the mobile, having an old MSC forward the ongoing call to the new MSC each time the mobile moves to a new MSC. Such MSC chaining can in fact occur in IS-41 cellular networks, with an optional path minimization step to remove MSCs between the anchor MSC and the current visited MSC [Lin 2001]. Let’s wrap up our discussion of GSM mobility management with a comparison of mobility management in GSM and Mobile IP. The comparison in Table 6.2 indicates that although IP and cellular networks are fundamentally different in many ways, they share a surprising number of common functional elements and overall approaches in handling mobility.



4.6 **Wireless and Mobility: Impact on Higher- Layer Protocols**

In this chapter, we've seen that wireless networks differ significantly from their wired counterparts at both the link layer (as a result of wireless channel characteristics such as fading, multipath, and hidden terminals) and at the network layer (as a result of mobile users who change their points of attachment to the network). But are there important differences at the transport and application layers? It's tempting to think that these differences will be minor, since the network layer provides the same best-effort delivery service model to upper layers in both wired and wireless networks. Similarly, if protocols such as TCP or UDP are used to provide transport-layer services to applications in both wired and wireless networks, then the application layer should remain unchanged as well. In one sense our intuition is right—TCP and UDP can (and do) operate in networks

with wireless links. On the other hand, transport protocols in general, and TCP in particular, can sometimes have very different performance in wired and wireless networks, and it is here, in terms of performance, that differences are manifested. Let's see why. Recall that TCP retransmits a segment that is either lost or corrupted on the path between sender and receiver. In the case of mobile users, loss can result from either network congestion (router buffer overflow) or from handoff (e.g., from delays in rerouting segments to a mobile's new point of attachment to the network). In all cases, TCP's receiver-to-sender ACK indicates only that a segment was not received intact; the sender is unaware of whether the segment was lost due to congestion,

during hando ff, or due to detected bit errors. In all cases, the sender's response is the same—to retransmit the segment. TCP's congestion-control response is *also* the same in all cases—TCP decreases its congestion window, as discussed in Section 3.7. By unconditionally decreasing its congestion window, TCP implicitly assumes that segment loss results from congestion rather than corruption or handoff. We saw in Section 6.2 that bit errors are much more common in wireless networks than in wired networks. When such bit errors occur or when handoff loss occurs, there's really no reason for the TCP sender to decrease its congestion window (and thus

decrease its sending rate). Indeed, it may well be the case that router buffers are empty and packets are flowing along the end-to-end path unimpeded by congestion. Researchers realized in the early to mid1990s that given high bit error rates on wireless links and the possibility of handoff loss, TCP's congestion-control response could be problematic in a wireless setting. Three broad classes of approaches are possible for dealing with this problem:

• *Local recovery*. Local recovery protocols recover from bit errors when and where (e.g., at the wireless link) they occur, e.g., the 802.11 ARQ protocol we studied in Section 6.3, or more sophisticated approaches that use both ARQ and FEC [Ayanoglu 1995].

• *TCP sender awareness of wireless links.* In the local recovery approaches, the TCP sender is blissfully unaware that its segments are traversing a wireless link. An alternative approach is for the TCP sender and receiver to be aware of the existence of a wireless link, to distinguish between congestive losses occurring in the wired network and corruption/loss occurring at the wireless link, and to invoke congestion control only in response to congestive wired-network losses. [Balakrishnan 1997] investigates various types of TCP, assuming that end systems can make this distinction. [Liu 2003] investigates techniques for distinguishing between losses on the wired and wireless segments of an end-to-end path.

• *Split-connection approaches.* In a split-connection approach [Bakre 1995], the end-to-end connection between the mobile user and the other end point is broken into two transport-layer connections: one from the mobile host to the wireless access point, and one from the wireless access point to the other communication end point (which we'll assume here is a wired host). The end-to-end connection is thus formed by the concatenation of a wireless part and a wired part. The transport layer over the wireless segment can be a standard TCP connection [Bakre 1995], or a specially tailored error recovery protocol on top of UDP. [Yavatkar 1994] investigates the use of a transport-layer selective repeat protocol over the wireless connection. Measurements reported in [Wei 2006] indicate that split TCP connections are widely used in cellular data networks, and that significant improvements can indeed be made through the use of split TCP connections.

Module – 5
Multimedia Networking Applications

Properties of Video

Perhaps the most salient characteristic of video is its **high bit rate**. Video distributed over the Internet typically ranges from 100 kbps for low-quality video conferencing to over 3 Mbps for streaming high-definition movies. To get a sense of how video bandwidth demands compare with those of other Internet applications, let's briefly consider three different users, each using a different Internet application. Our first user, Frank, is going quickly through photos posted on his friends' Facebook pages. Let's assume that Frank is looking at a new photo every 10 seconds, and that photos are on average 200 Kbytes in size. (As usual, throughout this discussion we make the simplifying assumption that 1 Kbyte = 8,000 bits.) Our second user, Martha, is streaming music from the Internet ("the cloud") to her smartphone. Let's assume Martha is listening to many MP3 songs, one after the other, each encoded at a rate of 128 kbps. Our third user, Victor, is watching a video that has been encoded at 2 Mbps.
Finally, let's suppose that the session length for all three users is 4,000 seconds (approximately 67 minutes). Table 7.1 compares the bit rates and the total bytes transferred for these three users. We see that video streaming consumes by far the most bandwidth, having a bit rate of more than ten times greater than that of the Facebook and music-streaming applications. Therefore, when designing networked
video applications, the first thing we must keep in mind is the high bit-rate requirements of video. Given the popularity of video and its high bit rate, it is perhaps not surprising that Cisco predicts [Cisco 2011] that streaming and stored video will be approximately 90 percent of global consumer Internet traffic by 2015.

	Bit rate	Bytes transferred in 67 min
Facebook Frank	160 kbps	80 Mbytes
Martha Music	128 kbps	64 Mbytes
Victor Video	2 Mbps	1 Gbyte

Table 7.1 ♦ Comparison of bit-rate requirements of three Internet applications

Another important characteristic of video is that it can be compressed, therebytrading off video quality with bit rate. A video is a sequence of images, typically being displayed at a constant rate, for example, at 24 or 30 images per second. An uncompressed, digitally encoded image consists of an array of pixels, with each pixel encoded into a number of bits to represent luminance and color. There are two types of redundancy in video, both of which can be exploited by **video compression**. *Spatial redundancy* is the redundancy within a given image. Intuitively, an image that consists of mostly white space has a high degree of redundancy and can be efficiently compressed without significantly sacrificing image quality. *Temporal redundancy* reflects repetition from image to subsequent image. If, for example, an image and the subsequent image are exactly the same, there is no reason to reencode the subsequent image; it is instead more efficient simply to indicate during encoding that the subsequent image is exactly the same. Today's off-the-shelf compression algorithms can compress a video to essentially any bit rate desired. Of course, the higher the bit rate, the better the image quality and the better the overall

user viewing experience.
We can also use compression to create **multiple versions** of the same video, each at a different quality level. For example, we can use compression to create, say, three versions of the same video, at rates of 300 kbps, 1 Mbps, and 3 Mbps. Users can then decide which version they want to watch as a function of their current available bandwidth. Users with high-speed Internet connections might choose the 3 Mbps version; users watching the video over 3G with a smartphone might choosethe 300 kbps version. Similarly, the video in a video conference application can be compressed "on-the-fly" to provide the best video quality given the available end to end bandwidth between conversing users.

Properties of Audio

Digital audio (including digitized speech and music) has significantly lower bandwidth requirements than video. Digital audio, however, has its own unique properties that must be considered when designing multimedia network applications. To understand these properties, let's first consider how analog audio (which humans and musical instruments generate) is converted to a digital signal:
• The analog audio signal is sampled at some fixed rate, for example, at 8,000 samples per second. The value of each sample is an arbitrary real number.
• Each of the samples is then rounded to one of a finite number of values. This operation is referred to as **quantization**. The number of such finite values called quantization values—is typically a power of two, for example, 256 quantization values.
• Each of the quantization values is represented by a fixed number of bits. For example, if there are 256 quantization values, then each value—and hence each audio sample—is represented by one byte. The bit representations of all the samples are then concatenated together to form the digital representation of the signal.
As an example, if an analog audio signal is sampled at 8,000 samples per second and each sample is quantized and represented by 8 bits, then the resulting
digital signal will have a rate of 64,000 bits per second. For playback through audio speakers, the digital signal can then be converted back—that is, decoded— to an analog signal. However, the decoded analog signal is only an approximation of the original signal, and the sound quality may be noticeably degraded (for example, high-frequency sounds may be missing in the decoded signal). By increasing the sampling rate and the number of quantization values, the decoded
signal can better approximate the original analog signal. Thus (as with video), there is a trade-off between the quality of the decoded signal and the bit-rate and storage requirements of the digital signal.
The basic encoding technique that we just described is called **pulse code modulation (PCM)**. Speech encoding often uses PCM, with a sampling rate of 8,000 samples per second and 8 bits per sample, resulting in a rate of 64 kbps. The audio compact disk (CD) also uses PCM, with a sampling rate of 44,100 samples per second with 16 bits per sample; this gives a rate of 705.6 kbps for mono and 1.411 Mbps for stereo.
PCM-encoded speech and music, however, are rarely used in the Internet. Instead, as with video, compression techniques are used to reduce the bit rates of the stream. Human speech can be compressed to less than 10 kbps and still be intelligible. A popular compression technique for near CD-quality stereo music is **MPEG 1 layer 3**, more commonly known as **MP3**. MP3 encoders can compress to many different rates; 128 kbps is the most common encoding rate and produces very little sound degradation. A related standard is **Advanced Audio Coding (AAC)**, which has been popularized by Apple. As with video, multiple versions of a prerecorded
audio stream can be created, each at a different bit rate.

Types of Multimedia Network Applications

The Internet supports a large variety of useful and entertaining multimedia applications. In this subsection, we classify multimedia applications into three broad categories: *(i) streaming stored audio/video*, *(ii) conversational voice/video-over-IP*, and *(iii) streaming live audio/video*. As we will soon see, each of these application categories has its own set of service requirements and design issues.

Streaming Stored Audio and Video

To keep the discussion concrete, we focus here on streaming stored video, which typically combines video and audio components. Streaming stored audio (such as streaming music) is very similar to streaming stored video, although the bit rates are typically much lower. In this class of applications, the underlying medium is prerecorded video, such as a movie, a television show, a prerecorded sporting event, or a prerecorded usergenerated video (such as those commonly seen on YouTube). These prerecorded videos are placed on servers, and users send requests to the servers to view the videos *on demand*. Many Internet companies today provide streaming video, including YouTube (Google), Netflix, and Hulu. By some estimates, streaming stored video makes up over 50 percent of the downstream traffic in the Internet access networkstoday [Cisco 2011]. Streaming stored video has three key distinguishing features.

• *Streaming.* In a streaming stored video application, the client typically begins video playout within a few seconds after it begins receiving the video from the server. This means that the client will be playing out from one location in the video while at the same time receiving later parts of the video from the server. This technique, known as **streaming**, avoids having to download the entire video file (and incurring a potentially long delay) before playout begins.

• *Interactivity*. Because the media is prerecorded, the user may pause, reposition forward, reposition backward, fast-forward, and so on through the video content. The time from when the user makes such a request until the action manifests itself at the client should be less than a few seconds for acceptable responsiveness.

• *Continuous playout.* Once playout of the video begins, it should proceed according to the original timing of the recording. Therefore, data must be received from the server in time for its playout at the client; otherwise, users experience video frame freezing (when the client waits for the delayed frames) or frame skipping (when the client skips over delayed frames).

Conversational Voice- and Video-over-IP

Real-time conversational voice over the Internet is often referred to as **Internet telephony**, since, from the user's perspective, it is similar to the traditional circuitswitched telephone service. It is also commonly called **Voice-over-IP (VoIP)**. Conversational video is similar, except that it includes the video of the participants as well as their voices. Most of today's voice and video conversational systems allow users to create conferences with three or more participants. Conversational voice and video are widely used in the Internet today, with the Internet companies Skype, QQ, and Google Talk boasting hundreds of millions of daily users.

we identified a number of axes along which application requirements can be classified. Two of these axes—timing considerations and tolerance of data loss—are particularly important for conversational voice and video applications. Timing considerations are important because audio and video conversational

applications are highly **delay-sensitive**. For a conversation with two or more interacting speakers, the delay from when a user speaks or moves until the action is manifested at the other end should be less than a few hundred milliseconds. For voice, delays smaller than 150 milliseconds are not perceived by a human listener, delays between 150 and 400 milliseconds can be acceptable, and delays exceeding 400 milliseconds can result in frustrating, if not completely unintelligible, voice conversations. On the other hand, conversational multimedia applications are **loss-tolerant**— occasional loss only causes occasional glitches in audio/video playback, and these losses can often be partially or fully concealed. These delay-sensitive but loss-tolerant characteristics are clearly different from those of elastic data applications such as Web browsing, e-mail, social networks, and remote login. For elastic applications, long delays are annoying but not particularly harmful; the completeness and integrity of the transferred data, however, are of paramount importance. We will explore conversational voice and video in more depth in Section 7.3, paying particular attention to how
adaptive playout, forward error correction, and error concealment can mitigate against network-induced packet loss and delay.

Streaming Live Audio and Video

This third class of applications is similar to traditional broadcast radio and television, except that transmission takes place over the Internet. These applications allow a user to receive a *live* radio or television transmission—such as a live sporting event or an ongoing news event—transmitted from any corner of the world. Today, thousands of radio and television stations around the world are broadcasting content over the Internet. Live, broadcast-like applications often have many users who receive the same audio/video program at the same time. Although the distribution of live audio/video to many receivers can be efficiently accomplished using the IP multicasting techniques described in Section 4.7, multicast distribution is more often accomplished today via application-layer multicast (using P2P networks or CDNs) or through multiple separate unicast streams. As with streaming stored multimedia, the network must provide each live multimedia flow with an average throughput that is larger than the video consumption rate. Because the event is live, delay can also be an issue, although the timing constraints are much less stringent than those for conversational voice. Delays of up to ten seconds or so from when the user chooses to view a live transmission to when playout begins can be tolerated. We will not cover streaming live media in this book because many of the techniques used for streaming live media—initial buffering delay, adaptive bandwidth use, and CDN distribution—are similar to those for streaming stored media.

Streaming Stored Video

For streaming video applications, prerecorded videos are placed on servers, and users send requests to these servers to view the videos on demand. The user may watch the video from beginning to end without interruption, may stop watching the video well before it ends, or interact with the video by pausing or repositioning to a future or past scene. Streaming video systems can be classified into three categories: **UDP streaming**, **HTTP streaming**, and **adaptive HTTP streaming**. Although all three types of systems are used in practice, the majority of today's systems employ HTTP streaming and adaptive HTTP streaming. A common characteristic of all three forms of video streaming is the extensive use of client-side application buffering to mitigate the effects of varying end-to-end delays and varying amounts of available bandwidth between server and client. For streaming video (both stored and live), users generally can tolerate a small

several second initial delay between when the client requests a video and when video playout begins at the client. Consequently, when the video starts to arrive at the client, the client need not immediately begin playout, but can instead build up a reserve of video in an application buffer. Once the client has built up a reserve of several seconds of buffered-but-not-yet-played video, the client can then begin video playout. There are two important advantages provided by such **client buffering**. First, clientside buffering can absorb variations in server-to-client delay. If a particular piece of video data is delayed, as long as it arrives before the reserve of received-but-notyet- played video is exhausted, this long delay will not be noticed. Second, if the server-to-client bandwidth briefly drops below the video consumption rate, a user
can continue to enjoy continuous playback, again as long as the client application buffer does not become completely drained. Figure 7.1 illustrates client-side buffering. In this simple example, suppose that video is encoded at a fixed bit rate, and thus each video block contains video frames
that are to be played out over the same fixed amount of time, . The server transmits the first video block at , the second block at , the third block at , and so on. Once the client begins playout, each block should be played out time units after the previous block in order to reproduce the timing of the original recorded video. Because of the variable end-to-end network delays, different video blocks experience different delays. The first video block arrives at the client at *t*1
and the second block arrives at . The network delay for the *i*th block is the horizontal distance between the time the block was transmitted by the server and the time it is received at the client; note that the network delay varies from one video block to another. In this example, if the client were to begin playout as soon as the first block arrived at , then the second block would not have arrived in time to be played out at out at . In this case, video playout would either have to stall waiting for block 1 to arrive) or block 1 could be skipped—both resulting in undesirable playout impairments. Instead, if the client were to delay the start of playout until , when blocks 1 through 6 have all arrived, periodic playout can proceed with *all* blocks having been received before their playout time.

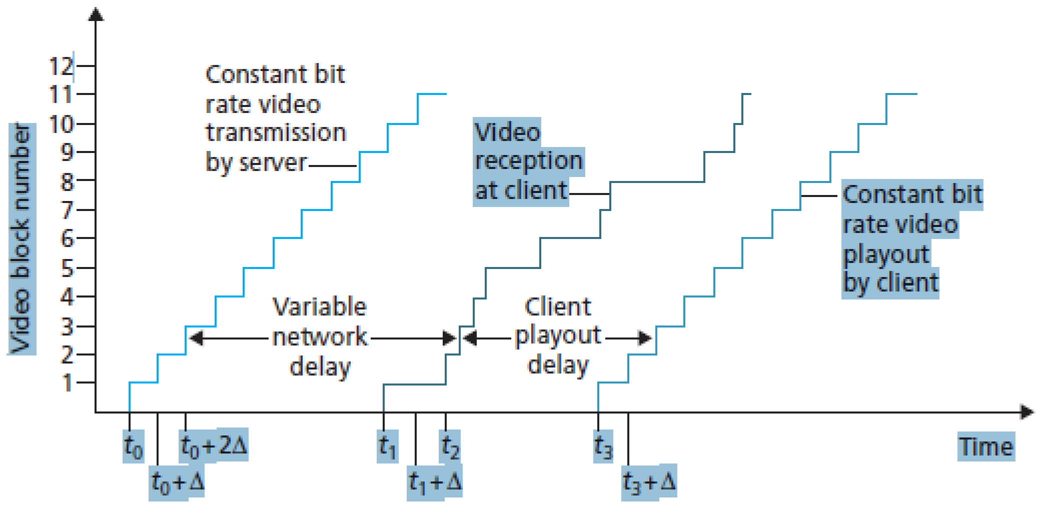


Figure 7.1 ♦ Client playout delay in video streaming

UDP Streaming

We only briefly discuss UDP streaming here, referring the reader to more in-depth discussions of the protocols behind these systems where appropriate. With UDP streaming,the server transmits video at a rate

that matches the client's video consumption rate by clocking out the video chunks over UDP at a steady rate. For example, if the video consumption rate is 2 Mbps and each UDP packet carries 8,000 bits of video, then the server would transmit one UDP packet into its socket every (8000 bits)/(2 Mbps) = 4 msec. As we learned in Chapter 3, because UDP does not employ a congestion-control mechanism, the server can push packets into the network at the consumption rate of the video without the rate-control restrictions of TCP. UDP streaming typically uses a small client-side buffer, big enough to hold less than a second of video. Before passing the video chunks to UDP, the server will encapsulate the video chunks within transport packets specially designed for transporting audio and video, using the Real-Time Transport Protocol (RTP) [RFC 3550] or a similar (possibly proprietary) scheme. We delay our coverage of RTP until Section 7.3, where we discuss RTP in the context of conversational voice and video systems. Another distinguishing property of UDP streaming is that in addition to the serverto- client video stream, the client and server also maintain, in parallel, a separate control connection over which the client sends commands regarding session state changes (such as pause, resume, reposition, and so on). This control connection is in many ways analogous to the FTP control connection we studied in Chapter 2. The Real-Time Streaming Protocol (RTSP) [RFC 2326], explained in some detail in the companion Web site for this textbook, is a popular open protocol for such a control connection. Although UDP streaming has been employed in many open-source systems and proprietary products, it suffers from three significant drawbacks. First, due to the unpredictable and varying amount of available bandwidth between server and client, constant-rate UDP streaming can fail to provide continuous playout. For example, consider the scenario where the video consumption rate is 1 Mbps and the serverto- client available bandwidth is usually more than 1 Mbps, but every few minutes the available bandwidth drops below 1 Mbps for several seconds. In such a scenario, a UDP streaming system that transmits video at a constant rate of 1 Mbps over RTP/UDP would likely provide a poor user experience, with freezing or skipped frames soon after the available bandwidth falls below 1 Mbps. The second drawback of UDP streaming is that it requires a media control server, such as an RTSP server,
to process client-to-server interactivity requests and to track client state (e.g., the client's playout point in the video, whether the video is being paused or played, and so on) for *each* ongoing client session. This increases the overall cost and complexity of deploying a large-scale video-on-demand system. The third drawback is that many firewalls are configured to block UDP traffic, preventing the users behind these firewalls from receiving UDP video

HTTP Streaming

In HTTP streaming, the video is simply stored in an HTTP server as an ordinary file with a specific URL. When a user wants to see the video, the client establishes a TCP connection with the server and issues an HTTP GET request for that URL. The server then sends the video file, within an HTTP response message, as quickly as possible, that is, as quickly as TCP congestion control and flow control will allow. On the client side, the bytes are collected in a client application buffer. Once the number of bytes in this buffer exceeds a predetermined threshold, the client application begins playback—specifically, it periodically grabs video frames from the client application buffer, decompresses the frames, and displays them on the user's screen We learned in Chapter 3 that when transferring a file over TCP, the server-toclient transmission rate can vary significantly due to TCP's congestion control mechanism. In particular, it is not uncommon for the transmission rate to vary in a "saw-tooth" manner (for example, Figure 3.53) associated with TCP congestion control. Furthermore, packets can also be significantly delayed due to TCP's retransmission mechanism. Because of these characteristics of TCP, the conventional wisdom in the 1990s was that video streaming would never work well over TCP. Over time, however, designers of streaming video systems learned that TCP's congestion control and reliable-data transfer mechanisms do not necessarily preclude continuous playout when client buffering and prefetching (discussed in the next section) are used.

The use of HTTP over TCP also allows the video to traverse firewalls and NATs more easily (which are often configured to block most UDP traffic but to allow most HTTP traffic). Streaming over HTTP also obviates the need for a media control server, such as an RTSP server, reducing the cost of a large-scale deployment over the Internet. Due to all of these advantages, most video streaming applications today—including YouTube and Netflix—use HTTP streaming (over TCP) as its underlying streaming protocol.

Prefetching Video

We just learned, client-side buffering can be used to mitigate the effects of varying end-to-end delays and varying available bandwidth. In our earlier example in Figure 7.1, the server transmits video at the rate at which the video is to be played out. However, for streaming *stored* video, the client can attempt to download the video at a rate *higher* than the consumption rate, thereby **prefetching** video frames that are to be consumed in the future. This prefetched video is naturally stored in the client application buffer. Such prefetching occurs naturally with TCP streaming, since TCP's congestion avoidance mechanism will attempt to use all of the available bandwidth between server and client. To gain some insight into prefetching, let's take a look at a simple example. Suppose the video consumption rate is 1 Mbps but the network is capable of delivering the video from server to client at a constant rate of 1.5 Mbps. Then the client will not only be able to play out the video with a very small playout delay, but will also be able to increase the amount of buffered video data by 500 Kbits every second. In this manner, if in the future the client receives data at a rate of less than 1 Mbps for a brief period of time, the client will be able to continue to provide continuous playback due to the reserve in its buffer. [Wang 2008] shows that when the average TCP throughput is roughly twice the media bit rate, streaming over TCP results in minimal starvation and low buffering delays.

Client Application Buffer and TCP Buffers

Figure 7.2 illustrates the interaction between client and server for HTTP streaming. At the server side, the portion of the video file in white has already been sent into the server's socket, while the darkened portion is what remains to be sent. After "passing through the socket door," the bytes are placed in the TCP send buffer before being transmitted into the Internet, as described in Chapter 3. In Figure 7.2 because the TCP send buffer is shown to be full, the server is momentarily prevented from sending more bytes from the video file into the socket. On the client side, the client application (media player) reads bytes from the TCP receive buffer (through its client socket) and places the bytes into the client application buffer. At the same time, the client application periodically grabs video frames from the client application buffer, decompresses the frames, and displays them on the user's screen. Note that if the client application buffer is larger than the video file, then the whole process of moving bytes from the server's storage to the client's application buffer is equivalent to an ordinary file download over HTTP—the client simply pulls the video off the server as fast as TCP will allow! Consider now what happens when the user pauses the video during the streaming process. During the pause period, bits are not removed from the client application buffer, even though bits continue to enter the buffer from the server. If the client application buffer is finite, it may eventually become full, which will cause "back pressure" all the way back to the server. Specifically, once the client application buffer becomes full, bytes can no longer be removed from the client TCP receive buffer, so it too becomes full. Once the client receive TCP buffer becomes full, bytes can no longer be removed from the client TCP send buffer, so it also becomes full. Once the TCP send buffer becomes full, the server cannot send any more bytes into the socket. Thus, if the user pauses the video, the server may be forced to stop transmitting, in which case the server will be blocked until the user resumes the video.

In fact, even during regular playback (that is, without pausing), if the client application buffer becomes full, back pressure will cause the TCP buffers to become full, which will force the server to reduce its rate. To determine the

resulting rate, note that when the client application removes *f* bits, it creates room for *f* bits in the client application buffer, which in turn allows the server to send *f* additional bits. Thus, the server send rate can be no higher than the video consumption rate at the client. Therefore, *a full client application buffer indirectl imposes a limit on the rate that video can be sent from server to client when streaming over HTTP.*

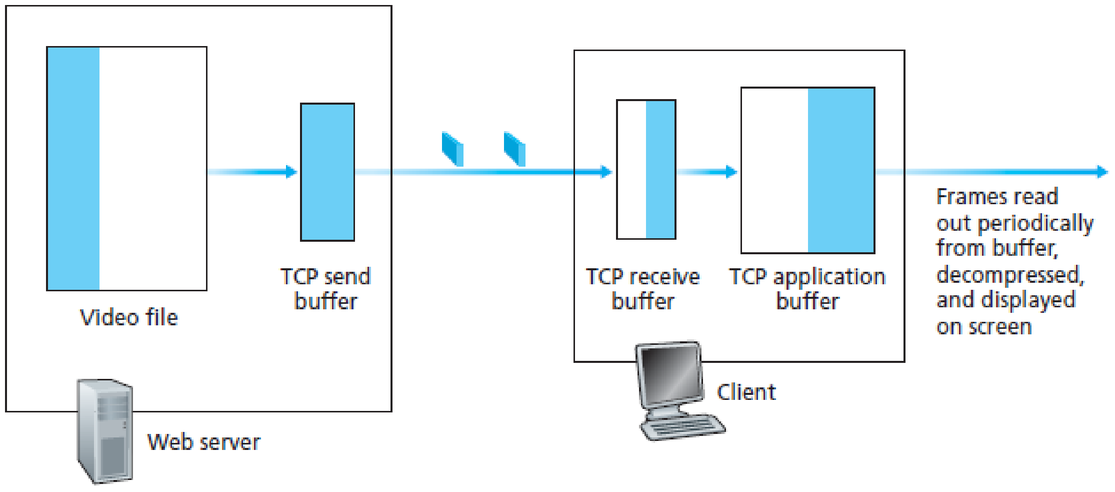


Figure 7.2 ♦ Streaming stored video over HTTP/TCP

Analysis of Video Streaming

Some simple modeling will provide more insight into initial playout delay and freezing due to application buffer depletion. As shown in Figure 7.3, let B denote the size (in bits) of the client's application buffer, and let Q denote the number of bits that must be buffered before the client application begins playout. (Of course, $Q < B$.) Let r denote the video consumption rate—the rate at which the client draws bits out of the client application buffer during playback. So, for example, if the video's frame rate is 30 frames/sec, and each (compressed) frame is 100,000 bits,

then $r = 3$ Mbps. To see the forest through the trees, we'll ignore TCP's send and receive buffers.

Let's assume that the server sends bits at a constant rate x whenever the client buffer is not full. (This is a gross simplification, since TCP's send rate varies due to congestion control; we'll examine more realistic time-dependent rates $x(t)$ in the problems at the end of this chapter.) Suppose at time $t = 0$, the application buffer is empty and video begins arriving to the client application buffer. We now ask at what time does playout begin? And while we are at it, at what time does the client application buffer become full? First,

let's determine , the time when Q bits have entered the application buffer and playout begins. Recall that bits arrive to the client applicati on buffer at rate x and *no* bits are removed from this buffer before playout begins. Thus, the amount of time required to build up Q bits (the initial buffering delay) is Q/x. Now let's determine , the point in time when the client application buffer becomes full. We first observe that if $x < r$ (that is, if the server send rate is less than the video consumption rate), then the client buffer will never become full! Indeed, starting at time , the buffer will be depleted at rate r and will only be filled at rate
$x < r$. Eventually the client buffer will empty out entirely, at which time the video will freeze on the screen while the client buffer waits another seconds to build up Q bits of video. *Thus, when the available rate in the network is less than the video rate, playout will alternate between periods of continuous playout and periods of freezing.* In a homework problem, you will be asked to determine the length of each continuous playout and freezing period as a function of Q, r, and x. Now let's determine for when $x > r$. In this case, starting at time , the buffer increases from Q to B at rate x $r$since bits are being depleted at rate r but are arriving at rate x, as shown in Figure 7.3. Given these hints, you will be asked in a homework problem to determine , the time the client buffer becomes full. Note that *when the available rate in the network is more than the video rate, after the initial buffering delay, the user will enjoy continuous playout until the video ends.*

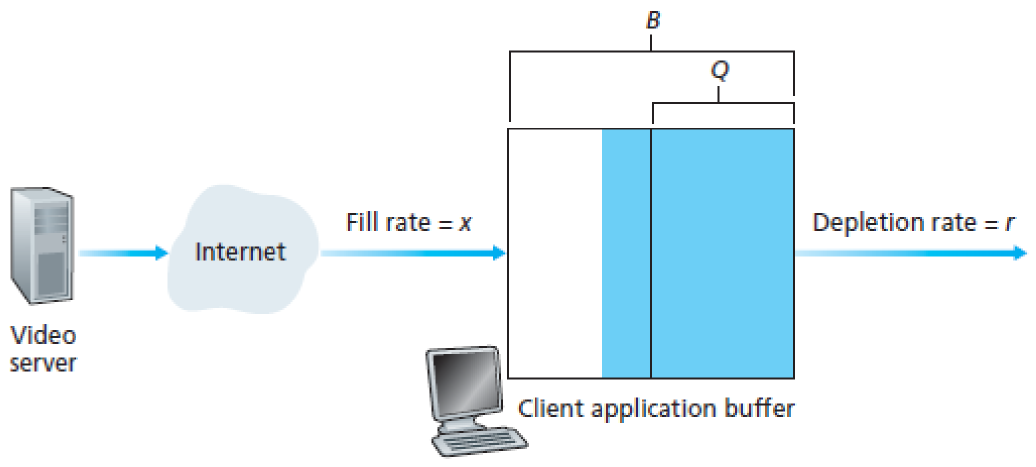


Figure 7.3 ♦ Analysis of client-side buffering for video streaming

Early Termination and Repositioning the Video

HTTP streaming systems often make use of the **HTTP byte-range header** in the HTTP GET request message, which specifies the specific range of bytes the client currently wants to retrieve from the desired video. This is particularly useful when the user wants to reposition (that is, jump) to a future point in time in the video.

When the user repositions to a new position, the client sends a new HTTP request,
indicating with the byte-range header from which byte in the file should the server send data. When the server receives the new HTTP request, it can forget about any earlier request and instead send bytes beginning with the byte indicated in the byterange request. While we are on the subject of repositioning, we briefly mention that when a user repositions to a future point in the video or terminates the video early, some prefetched-but-not-yet-viewed data transmitted by the server will go unwatched—a waste of network

bandwidth and server resources. For example, suppose that the client buffer is full with *B* bits at some time *t*0 into the video, and at this time the user repositions
to some instant $t > t0 + B/r$ into the video, and then watches the video to completion from that point on. In this case, all *B* bits in the buffer will be unwatched and the bandwidth and server resources that were used to transmit those *B* bits have been completely wasted. There is significant wasted bandwidth in the Internet due to early termination,
which can be quite costly, particularly for wireless links [Ihm 2011]. For this reason, many streaming systems use only a moderate-size client application buffer, or will limit the amount of prefetched video using the byte-range header in HTTP.

Adaptive Streaming and DASH

Although HTTP streaming, as described in the previous subsection, has been extensively deployed in practice (for example, by YouTube since its inception), it has a major shortcoming: All clients receive the same encoding of the video, despite the large variations in the amount of bandwidth available to a client, both across different clients and also over time for the same client. This has led to the development of a new type of HTTP-based streaming, often referred to as **Dynamic Adaptive Streaming over HTTP (DASH)**. In DASH, the video is encoded into several different versions, with each version having a different bit rate and, correspondingly, a different quality level. The client dynamically requests chunks of video segments of a few seconds in length from the different versions. When the amount of available bandwidth is high, the client naturally selects chunks from a high-rate version; and
when the available bandwidth is low, it naturally selects from a low-rate version.
The client selects different chunks one at a time with HTTP GET request message. On one hand, DASH allows clients with different Internet access rates to stream in video at different encoding rates. Clients with low-speed 3G connections can receive a low bit-rate (and low-quality) version, and clients with fiber connections can receive a high-quality version. On the other hand, DASH allows a client to adapt to the available bandwidth if the end-to-end bandwidth changes during the session. This feature is particularly important for mobile users, who typically see their bandwidth availability fluctuate as they move with respect to the base stations. Comcast, for example, has deployed an adaptive streaming system in which each video source file is encoded into 8 to 10 different MPEG-4 formats, allowing the highest quality video format to be streamed to the client, with adaptation being performed in
response to changing network and device conditions.
With DASH, each video version is stored in the HTTP server, each with a different URL. The HTTP server also has a **manifest file**, which provides a URL for each version along with its bit rate. The client first requests the manifest file and learns about the various versions. The client then selects one chunk at a time by specifying a URL and a byte range in an HTTP GET request message for each chunk. While downloading chunks, the client also measures the received bandwidth and runs a *rate determination algorithm* to select the chunk to request next. Naturally, if the client has a lot of video buffered and if the measured receive bandwidth is high, it will choose a chunk from a high-rate version. And naturally if the client has little video buffered and the measured received bandwidth is low, it will choose a chunk from a low-rate version. DASH therefore allows the client to freely switch among different quality levels.
Since a sudden drop in bit rate by changing versions may result in noticeable visual quality degradation, the bit-rate reduction may be achieved using multiple intermediate versions to smoothly transition to a rate where the client’s consumption rate drops below its available receive bandwidth. When the network conditions improve, the client can then later choose chunks from higher bit-rate versions.

Content Distribution Networks

Today, many Internet video companies are distributing on-demand multi-Mbps
streams to millions of users on a daily basis. YouTube, for example, with a library
of hundreds of millions of videos, distributes hundreds of millions of video streams to users around the world every day [Ding 2011]. Streaming all this traffic to locations all over the world while providing continuous playout and high interactivity is clearly a challenging task. For an Internet video company, perhaps the most straightforward approach to providing streaming video service is to build a single massive data center, store all of its videos in the data center, and stream the videos directly from the data center to clients worldwide. But there are three major problems with this approach. First, if the client is far from the data center, server-to-client packets will cross many communication links and likely pass through many ISPs, with some of the ISPs possibly located on different continents. If one of these links provides a throughput that is less than the video consumption rate, the end-to-end throughput will also be below the consumption rate, resulting in annoying freezing delays for the user. (Recall from Chapter 1 that the end-to-end throughput of a stream is governed by the throughput in the bottleneck link.) The likelihood of this happening increases as the number of links in the end-to-end path increases. A second drawback is that a popular video will likely be sent many times over the same communication links. Not only does this waste network bandwidth, but the Internet video company itself will be paying its provider ISP (connected to the data center) for sending the *same* bytes into the Internet over and over again. A third problem with this solution is that a single data center represents a single point of failure—if the data center or its links to the Internet goes down, it would not be able to distribute *any* video streams. In order to meet the challenge of distributing massive amounts of video data to users distributed around the world, almost all major video-streaming companies make use of **Content Distribution Networks (CDNs)**. A CDN manages servers in multiple geographically distributed locations, stores copies of the videos (and other types of Web content, including documents, images, and audio) in its servers, and
attempts to direct each user request to a CDN location that will provide the best user experience. The CDN may be a **private CDN**, that is, owned by the content provider itself; for example, Google's CDN distributes YouTube videos and other types of content. The CDN may alternatively be a **third-party CDN** that distributes content on behalf of multiple content providers; Akamai's CDN, for example, is a thirdparty CDN that distributes Netflix and Hulu content, among others. A very readable overview of modern CDNs is [Leighton 2009].

Case Studies: Netflix, YouTube, and KankanvNetflix

Generating almost 30 percent of the downstream U.S. Internet traffic in 2011, Netflix has become the leading service provider for online movies and TV shows in the United States [Sandvine 2011]. In order to rapidly deploy its large-scale service, Netflix has made extensive use of third-party cloud services and CDNs. Indeed, Netflix is an interesting example of a company deploying a large-scale online service by renting servers, bandwidth, storage, and database services from third parties while using hardly any infrastructure of its own. The following discussion is adapted from a very readable measurement study of the Netflix architecture [Adhikari 2012]. As we'll see, Netflix
employs many of the techniques covered earlier in this section, including video distribution using a CDN (actually multiple CDNs) and adaptive streaming over HTTP. Figure 7.6 shows the basic architecture of the Netflix video-streaming platform. It has four major components: the registration and payment servers, the Amazon cloud, multiple CDN providers, and clients. In its own hardware infrastructure, Netflix maintains registration and payment servers, which handle registration of new

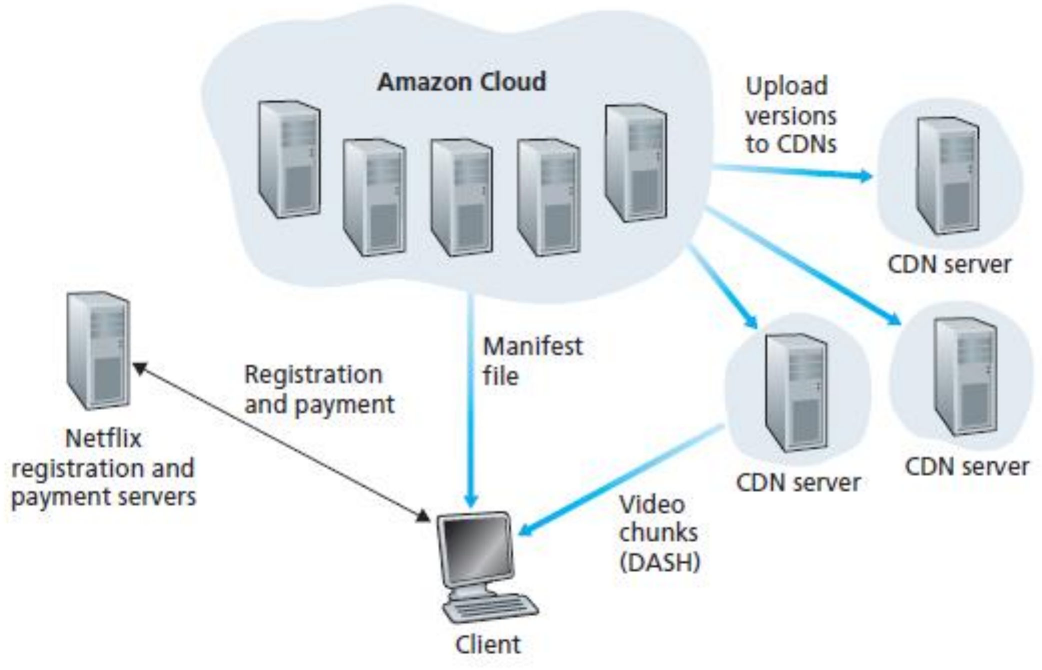


Figure 7.6 ♦ Netflix video streaming platform

accounts and capture credit-card payment information. Except for these basic functions, Netflix runs its online service by employing machines (or virtual machines) in the Amazon cloud. Some of the functions taking place in the Amazon cloud include:

• *Content ingestion.* Before Netflix can distribute a movie to its customers, it must first ingest and process the movie.

Netflix receives studio master versions of movies and uploads them to hosts in the Amazon cloud.

• *Content processing.* The machines in the Amazon cloud create many different formats for each movie, suitable for
a diverse array of client video players running on desktop computers, smartphones, and game consoles connected
to televisions. A different version is created for each of these formats and at multiple bit rates, allowing for
adaptive streaming over HTTP using DASH.

• *Uploading versions to the CDNs.* Once all of the versions of a movie have been created, the hosts in the Amazon cloud upload the versions to the CDNs To deliver the movies to its customers on demand, Netflix makes extensive use of CDN technology. In fact, as of this writing in 2012, Netflix employs not one but *three* third-party CDN companies at the same time—Akamai, Limelight, and Level-3. Having described the components of the Netflix architecture, let's take a closer look at the interaction between the client and the various servers that are involved in
movie delivery. The Web pages for browsing the Netflix video library are served from servers in the Amazon cloud. When the user selects a movie to "Play Now," the user's client obtains a manifest file, also from servers in the Amazon cloud. The manifest file includes a variety of information, including a ranked list of CDNs and the URLs for the different versions of the movie, which are used for DASH playback. The ranking of the CDNs is determined by Netflix, and may change from one streaming session to the next.

Typically the client will select the CDN that is ranked highest in the manifest file. After the client selects a CDN, the CDN leverages DNS to redirect the client to a specific CDN server, as described in Section 7.2.4. The client and that CDN server then interact using DASH. Specifically, as described in Section 7.2.3, the client uses the byte-range header in HTTP GET request messages,
to request chunks from the different versions of the movie. Netflix uses chunks that are approximately four-seconds long [Adhikari 2012]. While the chunks are being downloaded, the client measures the received throughput and runs a rate-determination algorithm to determine the quality of the next chunk to request.

YouTube

With approximately half a billion videos in its library and half a billion video views per day [Ding 2011], YouTube is indisputably the world's largest video-sharing site. YouTube began its service in April 2005 and was acquired by Google in November 2006. Although the Google/YouTube design and protocols are proprietary, through several independent measurement efforts we can gain a basic understanding about how YouTube operates [Zink 2009; Torres 2011; Adhikari 2011a]. As with Netflix, YouTube makes extensive use of CDN technology to distribute
its videos [Torres 2011]. Unlike Netflix, however, Google does not employ third-party CDNs but instead uses its own private CDN to distribute YouTube videos. Google has installed server clusters in many hundreds of different
locations. From a subset of about 50 of these locations, Google distributes YouTube videos [Adhikari 2011a]. Google uses DNS to redirect a customer request to a specific cluster, as described in Section 7.2.4. Most of the time,
Google's cluster selection strategy directs the client to the cluster for which the RTT between client and cluster is the lowest; however, in order to balance the load across clusters, sometimes the client is directed (via DNS) to a more distant cluster [Torres 2011]. Furthermore, if a cluster does not have the requested video, instead of fetching it from somewhere else and relaying it to the client, the cluster may return an HTTP redirect message, thereby redirecting the client to another cluster [Torres 2011].
YouTube employs HTTP streaming, as discussed in Section 7.2.2. YouTube often makes a small number of different versions available for a video, each with a different bit rate and corresponding quality level. As of 2011, YouTube does not employ adaptive streaming (such as DASH), but instead requires the user to manually
select a version. In order to save bandwidth and server resources that would be wasted by repositioning or early termination, YouTube uses the HTTP byte range request to limit the flow of transmitted data after a target amount of video is prefetched. A few million videos are uploaded to YouTube every day. Not only are YouTube videos streamed from server to client over HTTP, but YouTube uploaders also upload their videos from client to server over HTTP. YouTube processes each video it receives, converting it to a YouTube video format and creating multiple versions at different bit rates. This processing takes place entirely within Google data enters. Thus, in stark contrast to Netflix, which runs its service almost entirely on third-party infrastructures, Google runs the entire YouTube service within its own vast infrastructure of data centers, private CDN, and private global network
interconnecting its data centers and CDN clusters.

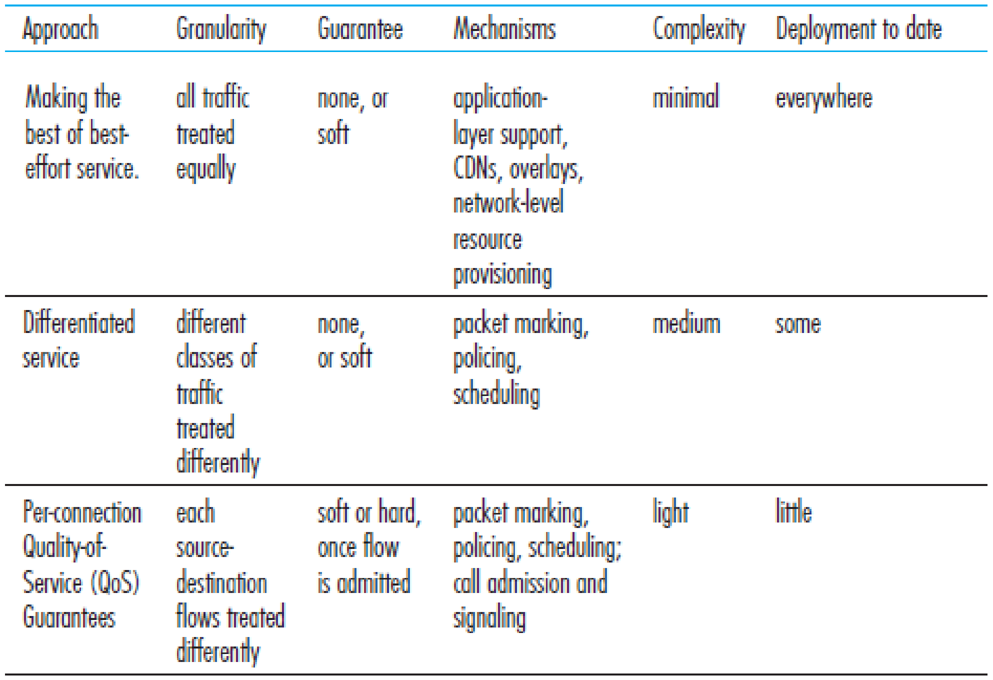
Kankan

We just saw that for both the Netflix and YouTube services, servers operated by CDNs (either third-party or private

CDNs) stream videos to clients. Netflix and YouTube not only have to pay for the server hardware (either directly through ownership or indirectly through rent), but also for the bandwidth the servers use to distribute the videos. Given the scale of these services and the amount of bandwidth they are consuming, such a “client-server” deployment is extremely costly. We conclude this section by describing an entirely different approach for providing
video on demand over the Internet at a large scale—one that allows the service provider to significantly reduce its infrastructure and bandwidth costs. As you might suspect, this approach uses P2P delivery instead of client-server (via CDNs) delivery. P2P video delivery is used with great success by several companies in China, including Kankan (owned and operated by Xunlei), PPTV (formerly PPLive), and PPs (formerly PPstream). Kankan, currently the leading P2P-based video-on-demand provider in China, has over 20 million unique users viewing its videos every month. At a high level, P2P video streaming is very similar to BitTorrent file downloading (discussed in Chapter 2). When a peer wants to see a video, it contacts a tracker (which may be centralized or peer-based using a DHT) to discover other peers in the system that have a copy of that video. This peer then requests chunks of the video file in parallel from these other peers that have the file. Different from downloading with BitTorrent, however, requests are preferentially made for chunks that are to be played back in the near future in order to ensure continuous playback.
The Kankan design employs a tracker and its own DHT for tracking content. Swarm sizes for the most popular content involve tens of thousands of peers, typically larger than the largest swarms in BitTorrent [Dhungel 2012]. The Kankan protocols— for communication between peer and tracker, between peer and DHT, and among peers—are all proprietary.

Network Support for Multimedia

we learned how application-level mechanisms such as client buffering, prefetching, adapting media quality to available bandwidth, adaptive playout, and loss mitigation techniques can be used by multimedia applications to improve a multimedia application’s performance. We also learned how content distribution networks and P2P overlay networks can be used to provide a *systemlevel* approach for delivering multimedia content. These techniques and approaches are all designed to be used in today’s best-effort Internet. Indeed, they are in use today precisely because the Internet provides only a single, best-effort class of service. But as designers of computer networks, we can’t help but ask whether the *network* (rather than the applications or application-level infrastructure alone) might
provide mechanisms to support multimedia content delivery. As we’ll see shortly, the answer is, of course, “yes”! But we’ll also see that a number of these new network- level mechanisms have yet to be widely deployed. This may be due to their complexity and to the fact that application-level techniques together with best-effort service and properly dimensioned network resources (for example, bandwidth) can indeed provide a “good-enough” (even if not-always-perfect) end-to-end multimedia delivery service.
Table 7.4 summarizes three broad approaches towards providing network-level support for multimedia applications.• *Making the best of best-effort service.* The application-level mechanisms and infrastructure that we studied in Sections 7.2 through 7.4 can be successfully used in a well-dimensioned network where packet loss and excessive end-to-end



Approach	Granularity	Guarantee	Mechanisms	Complexity	Deployment to date
Making the best of best-effort service.	all traffic treated equally	none, or soft	application-layer support, CDNs, overlays, network-level resource provisioning	minimal	everywhere
Differentiated service	different classes of traffic treated differently	none, or soft	packet marking, policing, scheduling	medium	some
Per-connection Quality-of-Service (QoS) Guarantees	each source-destination flows treated differently	soft or hard, once flow is admitted	packet marking, policing, scheduling; call admission and signaling	light	little

Table 7.4 ♦ Three network-level approaches to supporting multimedia applications

delay rarely occur. When demand increases are forecasted, the ISPs deploy additional bandwidth and switching capacity to continue to ensure satisfactory delay and packet-loss performance [Huang 2005]. We'll discuss such **network dimensioning**.

• *Differentiated service.* Since the early days of the Internet, it's been envisioned that different types of traffic (for example, as indicated in the Type-of-Service field in the IP4v packet header) could be provided with different classes of service, rather than a single one-size-fits-all best-effort service. With **differentiated service**, one type of traffic might be given strict priority over another class of traffic when both types of traffic are queued at a router. For example, packets belonging to a real-time conversational application might be given priority over other packets due to their stringent delay constraints. Introducing differentiated service into the network will require new mechanisms for packet marking (indicating a packet's class of service), packet scheduling, and more. We'll cover differentiated service, and new network mechanisms needed to implement this service

• *Per-connection Quality-of-Service (QoS) Guarantees.* With per-connection QoS guarantees, each instance of an application explicitly reserves end-to-end bandwidth and thus has a guaranteed end-to-end performance. A **hard guarantee** means the application will receive its requested quality of service (QoS) with certainty. A **soft guarantee** means the application will receive its requested quality of service with high probability. For example, if a user wants to make aVoIP call from Host A to Host B, the user's VoIP application reserves bandwidth explicitly in each link along a route between the two hosts. But permitting applications to make reservations and requiring the network to honor the reservations requires some big changes. First, we need a protocol that, on behalf of the applications, reserves link bandwidth on the paths from the senders to their receivers. Second, we'll need new scheduling policies in the router queues so that per-connection bandwidth reservations can be honored. Finally, in order to make a reservation, the applications must give the network a description of the traffic that they intend to send into the network and the network will need to police each application's traffic to make sure that it abides by that description. These mechanisms, when combined, require new and complex software in hosts and routers.

Dimensioning Best-Effort Networks

Fundamentally, the difficulty in supporting multimedia applications arises from their stringent performance requirements—low end-to-end packet delay, delay jitter, and loss—and the fact that packet delay, delay jitter, and loss occur whenever the network becomes congested. A first approach to improving the quality of multimedia applications—an approach that can often be used to solve just about any problem where resources are constrained—is simply to "throw money at the problem" and thus simply avoid resource contention. In the case of networked
multimedia, this means providing enough link capacity throughout the network so that network congestion, and its consequent packet delay and loss, never (or only very rarely) occurs. With enough link capacity, packets could zip
through today's Internet without queuing delay or loss. From many perspectives this is an ideal situation—multimedia applications would perform perfectly, users would be happy, and this could all be achieved with no changes to Internet's besteffort architecture.

The question, of course, is how much capacity is "enough" to achieve this nirvana, and whether the costs of providing "enough" bandwidth are practical from a business standpoint to the ISPs. The question of how much capacity to provide at network links in a given topology to achieve a given level of performance is often known as **bandwidth provisioning**. The even more complicated problem of how to design a network topology (where to place routers, how to interconnect routers with links, and what capacity to assign to links) to achieve a given level of end-to-end performance is a network design problem often referred to as **network dimensioning**. Both bandwidth provisioning and network dimensioning are complex topics, well beyond the scope of this textbook. We note here,
however, that the following issues must be addressed in order to predict application- level performance between two network end points, and thus provision enough capacity to meet an application's performance requirements. *Models of traffic demand between network end points.* Models may need to be specified at both the call level (for example, users "arriving" to the network and starting up end-to-end applications) and at the packet level (for example, packets
being generated by ongoing applications). Note that workload may change over time.

• *Well-defined performance requirements.* For example, a performance requirement for supporting delay-sensitive traffic, such as a conversational multimedia application, might be that the probability that the end-to-end delay of the packet is greater than a maximum tolerable delay be less than some small value.

• *Models to predict end-to-end performance for a given workload model, and techniques to find a minimal cost bandwidth allocation that will result in all user requirements being met.* Here, researchers are busy developing performance models that can quantify performance for a given workload, and optimization techniques to find minimal-cost bandwidth allocations meeting performance requirements.

Providing Multiple Classes of Service

Perhaps the simplest enhancement to the one-size-fits-all best-effort service in today's Internet is to divide traffic into classes, and provide different levels of service to these different classes of traffic. For example, an ISP might well want to provide a higher class of service to delay-sensitive Voice-over-IP or teleconferencing traffic (and charge more for this service!) than to elastic traffic such as email or HTTP. Alternatively, an ISP may simply want to provide a higher quality of service to customers willing to pay

more for this improved service. A number of residential wired-access ISPs and cellular wireless-access ISPs have adopted such tiered levels
of service—with platinum-service subscribers receiving better performance than gold- or silver-service subscribers. We're all familiar with different classes of service from our everyday lives—
first-class airline passengers get better service than business-class passengers, who in turn get better service than those of us who fly economy class; VIPs are provided immediate entry to events while everyone else waits in line; elders are revered in some countries and provided seats of honor and the finest food at a table. It's important to note that such differential service is provided among aggregates of traffic, that is, among classes of traffic, not among individual connections. For example, all first-class passengers are handled the same (with no first-class passenger receiving any better treatment than any other first-class passenger), just as all VoIP packets would receive the same treatment within the network, independent of the particular
end-to-end connection to which they belong. As we will see, by dealing with a small number of traffic aggregates, rather than a large number of individual connections, the new network mechanisms required to provide better-than-best service can be kept relatively simple.
The early Internet designers clearly had this notion of multiple classes of service in mind. Recall the type-of-service (ToS) field in the IPv4 header in Figure 4.13. IEN123 [ISI 1979] describes the ToS field also present in an ancestor of the IPv4 datagram as follows: "The Type of Service [field] provides an indication of the abstract parameters of the quality of service desired. These parameters are to be used to guide the selection of the actual service parameters when transmitting a datagram through a particular network. Several networks offer service precedence, which somehow treats high precedence traffic as more important that other traffic." More
than four decades ago, the vision of providing different levels of service to different classes of traffic was clear! However, it's taken us an equally long period of time to realize this vision.

Motivating Scenarios

Let's begin our discussion of network mechanisms for providing multiple classes of service with a few motivating scenarios. Figure 7.14 shows a simple network scenario in which two application packet flows originate on Hosts H1 and H2 on one LAN and are destined for Hosts H3 and H4 on another LAN. The routers on the two LANs are connected by a 1.5 Mbps link. Let's assume the LAN speeds are significantly higher than 1.5 Mbps, and focus on the output queue of router R1; it is here that packet delay and packet loss will occur if the aggregate sending rate of H1 and H2 exceeds 1.5 Mbps. Let's further suppose that a 1 Mbps audio application (for example, a CD-quality audio call) shares the 1.5 Mbps link between R1 and R2 with an HTTPWeb-browsing application that is downloading a Web page from H2 to H4.

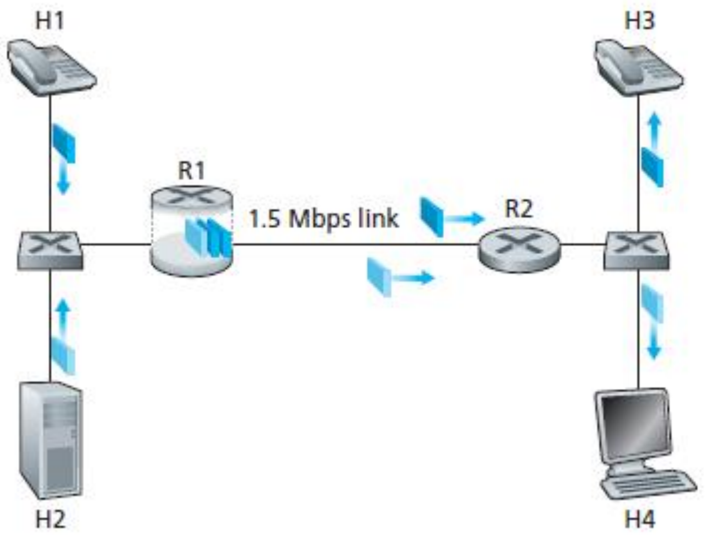


Figure 7.14 ♦ Competing audio and HTTP applications

In the best-effort Internet, the audio and HTTP packets are mixed in the output queue at R1 and (typically) transmitted in a first-in-first-out (FIFO) order. In this scenario, a burst of packets from the Web server could potentially fill up the queue, causing IP audio packets to be excessively delayed or lost due to buffer overflow at R1. How should we solve this potential problem? Given that the HTTP Web-browsing application does not have time constraints, our intuition might be to give strict priority to audio packets at R1. Under a strict priority scheduling discipline, an audio packet in the R1 output buffer would always be transmitted before any HTTP packet in the R1 output buffer. The link from R1 to R2 would look like a dedicated link of 1.5 Mbps to the audio traffic, with HTTP traffic using the R1-to-R2 link only when no audio traffic is queued. In order for R1 to distinguish between the audio and HTTP packets in its queue, each packet
must be marked as belonging to one of these two classes of traffic. This was the original goal of the type-of-service (ToS) field in IPv4. As obvious as this might seem, this then is our first insight into mechanisms needed to provide multiple classes.

Diffserv

Having seen the motivation, insights, and specific mechanisms for providing multiple classes of service, let's wrap up our study of approaches toward proving multiple classes of service with an example—the Internet Diffserv architecture [RFC 2475; RFC Kilkki 1999]. Diffserv provides service differentiation—that is, the abilityto handle different classes of traffic in different ways within the Internet in a scalable manner. The need for scalability arises from the fact that millions of simultaneous source-destination traffic flows may be present at a backbone router.
We'll see shortly that this need is met by placing only simple functionality within the network core, with more complex control operations being implemented at the network's edge.

Let’s begin with the simple network shown in Figure 7.25. We’ll describe one possible use of Diffserv here; other variations are possible, as described in RFC 2475. The Diffserv architecture consists of two sets of functional elements:

• *Edge functions: packet classification and traffic conditioning.* At the incoming edge of the network (that is, at either a Diffserv-capable host that generates traffic or at the first Diffserv-capable router that the traffic passes through), arriving packets are marked. More specifically, the differentiated service (DS) field in the IPv4 or IPv6 packet header is set to some value [RFC 3260]. The definition of the DS field is intended to supersede the earlier definitions of the IPv4 typeof- service field and the IPv6 traffic.

For example, in Figure 7.25, packets being sent from H1 to H3 might be marked

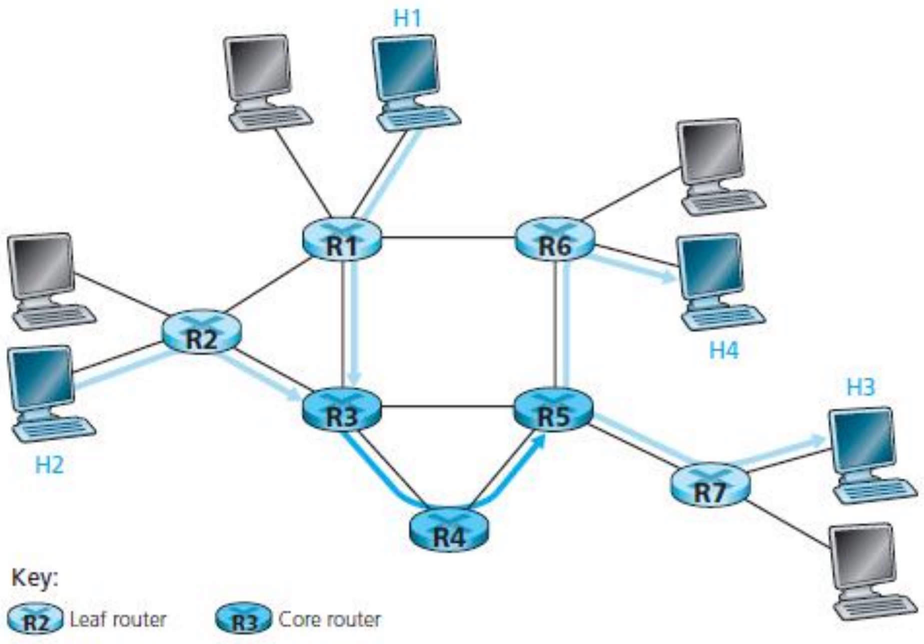


Figure 7.25 ♦ A simple Diffserv network example

at R1, while packets being sent from H2 to H4 might be marked at R2. The mark that a packet receives identifies the class of traffic to which it belongs. Different classes of traffic will then receive different service within the core network. • *Core function: forwarding.* When a DS-marked packet arrives at a Diffservcapable router, the packet is forwarded onto its next hop according to the so-called per-hop behavior (PHB) associated with that packet’s class. The per-hop behavior influences how a router’s buffers and link bandwidth are shared among the competing classes of traffic. Acrucial tenet of the Diffserv architecture is that a router’s perhop
behavior will be based only on packet markings, that is, the class of traffic to which a packet belongs. Thus, if packets being sent from H1 to H3 in Figure 7.25 receive the same marking as packets being sent from H2 to H4, then the network routers treat these packets as an aggregate, without distinguishing whether the packets originated at H1 or H2. For example, R3 would not distinguish between packets from H1 and H2 when forwarding these packets on to R4. Thus, the Diffserv architecture obviates the need to keep router state for individual source-destination

pairs—a critical consderation in making Diffserv scalable. An analogy might prove useful here. At many large-scale social events (for example, a large public reception, a large dance club or discothèque, a concert, or a football game), people entering the event receive a pass of one type or another: VIP passes for Very Important People; over-21 passes for people who are 21 years old or older (for example, if alcoholic drinks are to be served); backstage passes at concerts; press passes for reporters; even an ordinary pass for the Ordinary Person. These passes are typically distributed upon entry to the event, that is, at the edge of the event. It is here at the edge
where computationally intensive operations, such as paying for entry, checking for the appropriate type of invitation, and matching an invitation against a piece of identification,
are performed. Furthermore, there may be a limit on the number of people of a given type that are allowed into an event. If there is such a limit, people may have to wait before entering the event. Once inside the event, one's pass allows one to receive differentiated service at many locations around the event—a VIP is provided with free drinks, a better table, free food, entry to exclusive rooms, and fawning service. Conversely, an ordinary person is excluded from certain areas, pays for drinks, and receives only basic service. In both cases, the service received within the event depends solely on the type of one's pass. Moreover, all people within a class are treated alike. Figure 7.26 provides a logical view of the classification and marking functions within the edge router. Packets arriving to the edge router are first classified. The classifier selects packets based on the values of one or more packet header fields (for example, source address, destination address, source port, destination port, and protocol ID) and steers the packet to the appropriate marking function. As noted above, a packet's marking is carried in the DS field in the packet header. In some cases, an end user may have agreed to limit its packet-sending rate to conform
to a declared **traffic profile**. The traffic profile might contain a limit on the peak rate, as well as the burstiness of the packet flow, as we saw previously with the leaky bucket mechanism. As long as the user sends packets into the network in a way that conforms to the negotiated traffic profile, the packets receive their priority marking and are forwarded along their route to the destination. On the other hand, if the traffic profile is violated, out-of-profile packets might be marked differently, might be shaped (for example, delayed so that a maximum rate constraint would be observed), or might be dropped at the network edge. The role of the **metering function**, shown in Figure 7.26, is to compare the incoming packet flow with the negotiated traffic profile and to determine whether a packet is within the negotiated traffic profile. The actual decision about whether to immediately remark, forward, delay, or drop a packet is a policy issue determined by the network administrator and is *not* specified in the Diffserv architecture.
So far, we have focused on the marking and policing functions in the Diffserv architecture. The second key component of the Diffserv architecture involves the per-hop behavior (PHB) performed by Diffserv-capable routers. PHB is rather cryptically, but carefully, defined as "a description of the externally observable forwarding behavior of a Diffserv node applied to a particular Diffserv behavior aggregate" [RFC 2475]. Digging a little deeper into this definition, we can see several important considerations embedded within:
• A PHB can result in different classes of traffic receiving different performance

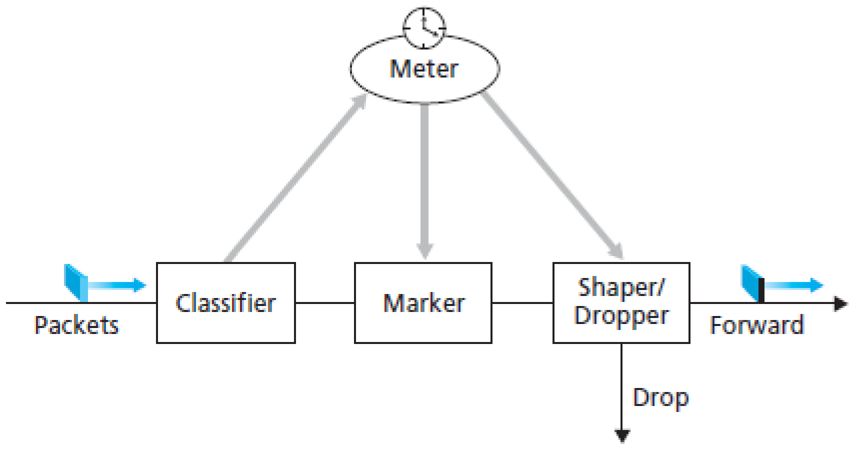


Figure 7.26 ♦ A simple Diffserv network example

While a PHB defines differences in performance (behavior) among classes, it does not mandate any particular mechanism for achieving these behaviors. As long as the externally observable performance criteria are met, any implementation mechanism and any buffer/bandwidth allocation policy can be used. For example, a PHB would not require that a particular packet-queuing discipline (for example, a priority queue versus a WFQ queue versus a FCFS queue) be
used to achieve a particular behavior. The PHB is the end, to which resource allocation and implementation mechanisms are the means.

• Differences in performance must be observable and hence measurable. Two PHBs have been defined: an expedited forwarding (EF) PHB [RFC 3246] and an assured forwarding (AF) PHB [RFC 2597]. The **expedited forwarding** PHB specifies that the departure rate of a class of traffic from a router must equal or exceed a configured rate. The **assured forwarding** PHB divides traffic into four classes, where each AF class is guaranteed to be provided with some minimum
amount of bandwidth and buffering.

Let’s close our discussion of Diffserv with a few observations regarding its service model. First, we have implicitly assumed that Diffserv is deployed within a single administrative domain, but typically an end-to-end service must be fashioned from multiple ISPs sitting between communicating end systems. In order to provide end-to-end Diffserv service, all the ISPs between the end systems must not only provide this service, but most also cooperate and make settlements in order to offer end customers true end-to-end service. Without this kind of cooperation, ISPs directly selling Diffserv service to customers will find themselves repeatedly saying: “Yes, we know you paid extra, but we don’t have a service agreement with the ISP that
dropped and delayed your traffic. I’m sorry that there were so many gaps in your VoIP call!” Second, if Diffserv were actually in place and the network ran at only moderate load, most of the time there would be no perceived difference between a best-effort service and a Diffserv service. Indeed, end-to-end delay is usually dominated by access rates and router hops rather than by queuing delays in the routers. Imagine the unhappy Diffserv customer who has paid more for premium service but finds that the best-effort service being provided to others almost always has the same performance as premium service.

Per-Connection Quality-of-Service (QoS) Guarantees: Resource Reservation and Call Admission

In the previous section, we have seen that packet marking and policing, traffic isolation, and link-level scheduling can provide one class of service with better performance han another. Under certain scheduling disciplines, such as priority scheduling, the lower classes of traffic are essentially “invisible” to the highest-priority class of traffic. With proper network dimensioning, the highest class of service can indeed achieve extremely low packet loss and delay—essentially circuit-like performance. But can the network *guarantee* that an ongoing flow in a high-priority traffic class will continue to receive such service throughout the flow’s duration using only the mechanisms that we have described so far? It cannot. In this section, we’ll see why yet additional network mechanisms and protocols are required when a hard service guarantee is provided to individual connections. Let’s return to our scenario from Section 7.5.2 and consider two 1 Mbps audio applications transmitting their packets over the 1.5 Mbps link, as shown in Figure 7.27. The combined data rate of the two flows (2 Mbps) exceeds the link

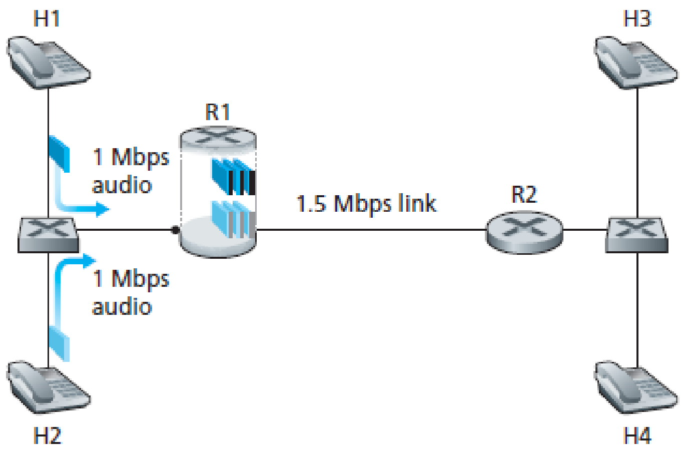


Figure 7.27 ♦ Two competing audio applications overloading the R1-to-R2 link

capacity. Even with classification and marking, isolation of flows, and sharing of unused bandwidth (of which there is none), this is clearly a losing proposition. There is simply not enough bandwidth to accommodate the needs of both applications at the same time. If the two applications equally share the bandwidth, each application would lose 25 percent of its transmitted packets. This is such an unacceptably low QoS that both audio applications are completely unusable; there’s no need even to transmit any audio packets in the first place. Given that the two applications in Figure 7.27 cannot both be satisfied simultaneously, what should the network do? Allowing both to proceed with an unusable QoS wastes network resources on application flows that ultimately provide no utility to the end user. The answer is hopefully clear—one of the application flows should be blocked (that is, denied access to the network), while the other should be allowed to proceed on, using the full 1 Mbps needed by the application. The telephone network is an example of a network that performs such call blocking—if the required resources (an end-to-end circuit in the case of the telephone network) cannot be allocated to the call, the call is blocked (prevented from entering the network) and a busy signal is returned to the user. In our example, there is no gain in allowing a flow into the network if it will not receive a sufficient QoS to be considered usable. Indeed, there is a cost to admitting a flow that does not receive its needed QoS, as network resources are being used to support a flow that provides no utility to the end user.

By explicitly admitting or blocking flows based on their resource requirements, and the source requirements of already-admitted flows, the network can guarantee that admitted flows will be able to receive their requested QoS. Implicit in the need to provide a guaranteed QoS to a flow is the need for the flow to declare its QoS requirements. This process of having a flow declare its QoS requirement, and then having the network either accept the flow (at the required QoS) or block the flow is referred to as the **call admission** process. This then is our fourth insight (in addition to the three earlier insights from Section 7.5.2) into the mechanisms needed to provide

QoS. **Insight 4:** If sufficient resources will not always be available, and QoS is to be *guaranteed*, a call admission process is needed in which flows declare their QoS requirements and are then either admitted to the network (at the required QoS) or blocked from the network (if the required QoS cannot be provided by the network). Our motivating example in Figure 7.27 highlights the need for several new network mechanisms and protocols if a call (an end-to-end flow) is to be guaranteed a given quality of service once it begins:

• *Resource reservation.* The only way to *guarantee* that a call will have theresources (link bandwidth, buffers) needed to meet its desired QoS is to explicitly allocate those resources to the call—a process known in networking parlance as **resource reservation**. Once resources are reserved, the call has on-demand access to these resources throughout its duration, regardless of the demands of all other calls. If a call reserves and receives a guarantee of *x* Mbps of link bandwidth, and never transmits at a rate greater than *x*, the call will see loss- and delay-free performance.

• *Call admission.* If resources are to be reserved, then the network must have a mechanism for calls to request and reserve resources. Since resources are not infinite, a call making a call admission request will be denied admission, that is, be blocked, if the requested resources are not available. Such a call admission is performed by the telephone network—we request resources when we dial a number. If the circuits (TDMA slots) needed to complete the call are available, the circuits are allocated and the call is completed. If the circuits are not available, then the call is blocked, and we receive a busy signal. A blocked call can try again to gain admission to the network, but it is not allowed to send traffic into the network until it has successfully completed the call admission process. Of course, a router that allocates link bandwidth should not allocate more than is available at that link. Typically, a call may reserve only a fraction of the link's bandwidth, and so a router may allocate link bandwidth to more than one call.However, the sum of the allocated bandwidth to all calls should be less than the link capacity if hard quality of service guarantees are to be provided.

• *Call setup signaling.* The call admission process described above requires that a call be able to reserve sufficient resources at each and every network router on its source-to-destination path to ensure that its end-to-end QoS requirement is met. Each router must determine the local resources required by the session, consider the amounts of its resources that are already committed to other ongoing sessions, and determine whether it has sufficient resources to

satisfy the per-hop QoS requirement of the session at this router without violating local QoS guarantees made to an already-admitted session. A signaling protocol is needed to coordinate these various activities—the per-hop allocation of local resources, as well as the overall end-to-end decision of whether or not the call has been able to reserve sufficient resources at each and every router on the end-to-end path. This is the job of the **call setup protocol**, asshown in Figure 7.28. The **RSVP protocol** [Zhang 1993, RFC 2210] was proposed for this purpose within an Internet architecture for providing qualityof- service guarantees. In ATM networks, the Q2931b protocol [Black 1995] carries this information among the ATM network's switches and end point.