between the client and the server. It causes TCP in the client to send a TCP message to TCP in the server (see Section 3.5). TCPServer.py Now let's take a look at the server program. from socket import * serverPort = 12000 serverSocket = socket(AF_INET, SOCK_STREAM) serverSocket.bind((’’, serverPort)) serverSocket.listen(1) print('The server is ready to receive') while True: connectionSocket, addr = serverSocket.accept() sentence = connectionSocket.recv(1024).decode() capitalizedSentence = sentence.upper() connectionSocket.send(capitalizedSentence.encode()) connectionSocket.close() Let's now take a look at the lines that differ significantly from UDPServer and TCPClient. As with TCPClient, the server creates a TCP socket with: serverSocket=socket(AF_INET, SOCK_STREAM) Similar to UDPServer, we associate the server port number, serverPort , with this socket: serverSocket.bind((’’, serverPort)) But with TCP, serverSocket will be our welcoming socket. After establishing this welcoming door, we will wait and listen for some client to knock on the door: serverSocket.listen(1) This line has the server listen for TCP connection requests from the client. The parameter specifies the maximum number of queued connections (at least 1). connectionSocket, addr = serverSocket.accept() When a client knocks on this door, the program invokes the accept() method for serverSocket, which creates a new socket in the server, called connectionSocket , dedicated to this particular client. The client and server then complete the handshaking, creating a TCP connection between the client's clientSocket and the server's connectionSocket . With the TCP connection established, the client and server can now send bytes to each other over the connection. With TCP, all bytes sent from one side not are not only guaranteed to arrive at the other side but also guaranteed arrive in order. connectionSocket.close() In this program, after sending the modified sentence to the client, we close the connection socket. But since serverSocket remains open, another client can now knock on the door and send the server a sentence to modify. This completes our discussion of socket programming in TCP. You are encouraged to run the two programs in two separate hosts, and also to modify them to achieve slightly different goals. You should compare the UDP program pair with the TCP program pair and see how they differ. You should also do many of the socket programming assignments described at the ends of Chapter 2, 4, and 9. Finally, we hope someday, after mastering these and more advanced socket programs, you will write your own popular network application, become very rich and famous, and remember the authors of this textbook! 2.8 Summary In this chapter, we've studied the conceptual and the implementation aspects of network applications. We've learned about the ubiquitous client-server architecture adopted by many Internet applications and seen its use in the HTTP, SMTP, POP3, and DNS protocols. We've studied these important applicationlevel protocols, and their corresponding associated applications (the Web, file transfer, e-mail, and DNS) in some detail. We've learned about the P2P architecture and how it is used in many applications. We've also learned about streaming video, and how modern video distribution systems leverage CDNs. We've examined how the socket API can be used to build network applications. We've walked through the use of sockets for connection-oriented (TCP) and connectionless (UDP) end-to-end transport services. The first step in our journey down the layered network architecture is now complete! At the very beginning of this book, in Section 1.1, we gave a rather vague, bare-bones definition of a protocol: "the format and the order of messages exchanged between two or more communicating entities, as well as the actions taken on the transmission and/or receipt of a message or other event." The material in this chapter, and in particular our detailed study of the HTTP, SMTP, POP3, and DNS protocols, has now added considerable substance to this definition. Protocols are a key concept in networking; our study of application protocols has now given us the opportunity to develop a more intuitive feel for what protocols are all about. In Section 2.1, we described the service models that TCP and UDP offer to applications that

invoke them. We took an even closer look at these service models when we developed simple applications that run over TCP and UDP in Section 2.7. However, we have said little about how TCP and UDP provide these service models. For example, we know that TCP provides a reliable data service, but we haven't said yet how it does so. In the next chapter we'll take a careful look at not only the what, but also the how and why of transport protocols. Equipped with knowledge about Internet application structure and application-level protocols, we're now ready to head further down the protocol stack and examine the transport layer in Chapter 3. Homework Problems and Questions Chapter 2 Review Questions SECTION 2.1 SECTION 2.2–2.5 R1. List five nonproprietary Internet applications and the application-layer protocols that they use. R2. What is the difference between network architecture and application architecture? R3. For a communication session between a pair of processes, which process is the client and which is the server? R4. For a P2P file-sharing application, do you agree with the statement, "There is no notion of client and server sides of a communication session"? Why or why not? R5. What information is used by a process running on one host to identify a process running on another host? R6. Suppose you wanted to do a transaction from a remote client to a server as fast as possible. Would you use UDP or TCP? Why? R7. Referring to Figure 2.4 , we see that none of the applications listed in Figure 2.4 requires both no data loss and timing. Can you conceive of an application that requires no data loss and that is also highly time-sensitive? R8. List the four broad classes of services that a transport protocol can provide. For each of the service classes, indicate if either UDP or TCP (or both) provides such a service. R9. Recall that TCP can be enhanced with SSL to provide process-to-process security services, including encryption. Does SSL operate at the transport layer or the application layer? If the application developer wants TCP to be enhanced with SSL, what does the developer have to do? R10. What is meant by a handshaking protocol? R11. Why do HTTP, SMTP, and POP3 run on top of TCP rather than on UDP? R12. Consider an e-commerce site that wants to keep a purchase record for each of its customers. Describe how this can be done with cookies. R13. Describe how Web caching can reduce the delay in receiving a requested object. Will Web caching reduce the delay for all objects requested by a user or for only some of the objects? SECTION 2.5 SECTION 2.6 Why? R14. Telnet into a Web server and send a multiline request message. Include in the request message the If-modified-since: header line to force a response message with the 304 Not Modified status code. R15. List several popular messaging apps. Do they use the same protocols as SMS? R16. Suppose Alice, with a Web-based e-mail account (such as Hotmail or Gmail), sends a message to Bob, who accesses his mail from his mail server using POP3. Discuss how the message gets from Alice's host to Bob's host. Be sure to list the series of application-layer protocols that are used to move the message between the two hosts. R17. Print out the header of an e-mail message you have recently received. How many Received: header lines are there? Analyze each of the header lines in the message. R18. From a user's perspective, what is the difference between the download-and-delete mode and the download-and-keep mode in POP3? R19. Is it possible for an organization's Web server and mail server to have exactly the same alias for a hostname (for example, foo.com)? What would be the type for the RR that contains the hostname of the mail server? R20. Look over your received e-mails, and examine the header of a message sent from a user with a .edu e-mail address. Is it possible to determine from the header the IP address of the host from which the message was sent? Do the same for a message sent from a Gmail account. R21. In BitTorrent, suppose Alice provides chunks to Bob throughout a 30-second interval. Will Bob necessarily return the favor and provide chunks to Alice in this same interval? Why or why not? R22. Consider a new peer Alice that joins BitTorrent without possessing any chunks. Without any chunks, she cannot become a top-four uploader for any of the other peers, since she has nothing to upload. How then will

Alice get her first chunk? R23. What is an overlay network? Does it include routers? What are the edges in the overlay network? R24. CDNs typically adopt one of two different server placement philosophies. Name and briefly describe them. R25. Besides network-related considerations such as delay, loss, and bandwidth performance, there are other important factors that go into designing a CDN server selection strategy. What are they? SECTION 2.7 Problems R26. In Section 2.7, the UDP server described needed only one socket, whereas the TCP server needed two sockets. Why? If the TCP server were to support n simultaneous connections, each from a different client host, how many sockets would the TCP server need? R27. For the client-server application over TCP described in Section 2.7 , why must the server program be executed before the client program? For the client-server application over UDP, why may the client program be executed before the server program? P1. True or false? a. A user requests a Web page that consists of some text and three images. For this page, the client will send one request message and receive four response messages. b. Two distinct Web pages (for example, www.mit.edu/research.html and www.mit.edu/students.html) can be sent over the same persistent connection. c. With nonpersistent connections between browser and origin server, it is possible for a single TCP segment to carry two distinct HTTP request messages. d. The Date: header in the HTTP response message indicates when the object in the response was last modified. e. HTTP response messages never have an empty message body. P2. SMS, iMessage, and WhatsApp are all smartphone real-time messaging systems. After doing some research on the Internet, for each of these systems write one paragraph about the protocols they use. Then write a paragraph explaining how they differ. P3. Consider an HTTP client that wants to retrieve a Web document at a given URL. The IP address of the HTTP server is initially unknown. What transport and application-layer protocols besides HTTP are needed in this scenario? P4. Consider the following string of ASCII characters that were captured by Wireshark when the browser sent an HTTP GET message (i.e., this is the actual content of an HTTP GET message). The characters are carriage return and line-feed characters (that is, the italized character string in the text below represents the single carriage-return character that was contained at that point in the HTTP header). Answer the following questions, indicating where in the HTTP GET message below you find the answer. GET /cs453/index.html HTTP/1.1Host: gai a.cs.umass.eduUser-Agent: Mozilla/5.0 (Windows;U; Windows NT 5.1; en-US; rv:1.7.2) Gec ko/20040804 Netscape/7.2 (ax) Accept:ex t/xml, application/xml, application/xhtml+xml, text /html;q=0.9, text/plain;q=0.8, image/png,*/*;q=0.5 Accept-Language: en-us, en;q=0.5AcceptEncoding: zip, deflateAccept-Charset: ISO -8859-1, utf-8;q=0.7,*;q=0.7Keep-Alive: 300 Connection:keep-alive a. What is the URL of the document requested by the browser? b. What version of HTTP is the browser running? c. Does the browser request a non-persistent or a persistent connection? d. What is the IP address of the host on which the browser is running? e. What type of browser initiates this message? Why is the browser type needed in an HTTP request message? P5. The text below shows the reply sent from the server in response to the HTTP GET message in the question above. Answer the following questions, indicating where in the message below you find the answer. HTTP/1.1 200 OKDate: Tue, 07 Mar 2008 12:39:45GMTServer: Apache/2.0.52 (Fedora) Last-Modified: Sat, 10 Dec2005 18:27:46 GMTETag: ”526c3-f22-a88a4c80”AcceptRanges: bytesContent-Length: 3874 Keep-Alive: timeout=max=100Connection: Keep-AliveContent-Type: text/html; charset= ISO-8859-1 a. Was the server able to successfully find the document or not? What time was the document reply provided? b. When was the document last modified? c. How many bytes are there in the document being returned? d. What are the first 5 bytes of the document being returned? Did the server agree to a persistent connection? P6. Obtain the HTTP/1.1 specification (RFC 2616). Answer the following questions: a. Explain the mechanism used for signaling between the client

and server to indicate that a persistent connection is being closed. Can the client, the server, or both signal the close of a connection? b. What encryption services are provided by HTTP? c. Can a client open three or more simultaneous connections with a given server? d. Either a server or a client may close a transport connection between them if either one detects the connection has been idle for some time. Is it possible that one side starts closing a connection while the other side is transmitting data via this connection? Explain. P7. Suppose within your Web browser you click on a link to obtain a Web page. The IP address for the associated URL is not cached in your local host, so a DNS lookup is necessary to obtain the IP address. Suppose that n DNS servers are visited before your host receives the IP address from DNS; the successive visits incur an RTT of Further suppose that the Web page associated with the link contains exactly one object, consisting of a small amount of HTML text. Let RTT denote the RTT between the local host and the server containing the object. Assuming zero transmission time of the object, how much time elapses from when the client clicks on the link until the client receives the object? P8. Referring to Problem P7, suppose the HTML file references eight very small objects on the same server. Neglecting transmission times, how much time elapses with a. Non-persistent HTTP with no parallel TCP connections? b. Non-persistent HTTP with the browser configured for 5 parallel connections? c. Persistent HTTP? P9. Consider Figure 2.12 , for which there is an institutional network connected to the Internet. Suppose that the average object size is 850,000 bits and that the average request rate from the institution's browsers to the origin servers is 16 requests per second. Also suppose that the amount of time it takes from when the router on the Internet side of the access link forwards an HTTP request until it receives the response is three seconds on average (see Section 2.2.5). Model the total average response time as the sum of the average access delay (that is, the delay from Internet router to institution router) and the average Internet delay. For the average access delay, use where Δ is the average time required to send an object over the access link and b is the arrival rate of objects to the access link. a. Find the total average response time. b. Now suppose a cache is installed in the institutional LAN. Suppose the miss rate is 0.4. Find the total response time. RTT1,. . .,RTTn. 0 Δ/(1−Δβ), P10. Consider a short, 10-meter link, over which a sender can transmit at a rate of 150 bits/sec in both directions. Suppose that packets containing data are 100,000 bits long, and packets containing only control (e.g., ACK or handshaking) are 200 bits long. Assume that N parallel connections each get 1/N of the link bandwidth. Now consider the HTTP protocol, and suppose that each downloaded object is 100 Kbits long, and that the initial downloaded object contains 10 referenced objects from the same sender. Would parallel downloads via parallel instances of non-persistent HTTP make sense in this case? Now consider persistent HTTP. Do you expect significant gains over the non-persistent case? Justify and explain your answer. P11. Consider the scenario introduced in the previous problem. Now suppose that the link is shared by Bob with four other users. Bob uses parallel instances of non-persistent HTTP, and the other four users use non-persistent HTTP without parallel downloads. a. Do Bob's parallel connections help him get Web pages more quickly? Why or why not? b. If all five users open five parallel instances of non-persistent HTTP, then would Bob's parallel connections still be beneficial? Why or why not? P12. Write a simple TCP program for a server that accepts lines of input from a client and prints the lines onto the server's standard output. (You can do this by modifying the TCPServer.py program in the text.) Compile and execute your program. On any other machine that contains a Web browser, set the proxy server in the browser to the host that is running your server program; also configure the port number appropriately. Your browser should now send its GET request messages to your server, and your server should display the messages on its standard output. Use this platform to determine whether your browser generates conditional GET messages for objects that are locally cached. P13. What is the

difference between MAIL FROM : in SMTP and From : in the mail message itself? P14. How does SMTP mark the end of a message body? How about HTTP? Can HTTP use the same method as SMTP to mark the end of a message body? Explain. P15. Read RFC 5321 for SMTP. What does MTA stand for? Consider the following received spam e-mail (modified from a real spam e-mail). Assuming only the originator of this spam e-mail is malicious and all other hosts are honest, identify the malacious host that has generated this spam e-mail. From - Fri Nov 07 13:41:30 2008 Return-Path: Received: from barmail.cs.umass.edu (barmail.cs.umass.edu [128.119.240.3]) by cs.umass.edu (8.13.1/8.12.6) for ; Fri, 7 Nov 2008 13:27:10 -0500 Received: from asusus-4b96 (localhost [127.0.0.1]) by barmail.cs.umass.edu (Spam Firewall) for ; Fri, 7 Nov 2008 13:27:07 -0500 (EST) Received: from asusus-4b96 ([58.88.21.177]) by barmail.cs.umass.edu for ; Fri, 07 Nov 2008 13:27:07 -0500 (EST) Received: from [58.88.21.177] by inbnd55.exchangeddd.com; Sat, 8 Nov 2008 01:27:07 +0700 From: ”Jonny” To: Subject: How to secure your savings P16. Read the POP3 RFC, RFC 1939. What is the purpose of the UIDL POP3 command? P17. Consider accessing your e-mail with POP3. a. Suppose you have configured your POP mail client to operate in the download-anddelete mode. Complete the following transaction: C: list S: 1 498 S: 2 912 S: . C: retr 1 S: blah blah ... S: .....blah S: . ? ? b. Suppose you have configured your POP mail client to operate in the download-and-keep mode. Complete the following transaction: C: list S: 1 498 S: 2 912 S: . C: retr 1 S: blah blah ... S: .....blah S: . ? ? c. Suppose you have configured your POP mail client to operate in the download-and-keep mode. Using your transcript in part (b), suppose you retrieve messages 1 and 2, exit POP, and then five minutes later you again access POP to retrieve new e-mail. Suppose that in the five-minute interval no new messages have been sent to you. Provide a transcript of this second POP session. P18. a. What is a whois database? b. Use various whois databases on the Internet to obtain the names of two DNS servers. Indicate which whois databases you used. c. Use nslookup on your local host to send DNS queries to three DNS servers: your local DNS server and the two DNS servers you found in part (b). Try querying for Type A, NS, and MX reports. Summarize your findings. d. Use nslookup to find a Web server that has multiple IP addresses. Does the Web server of your institution (school or company) have multiple IP addresses? e. Use the ARIN whois database to determine the IP address range used by your university. f. Describe how an attacker can use whois databases and the nslookup tool to perform reconnaissance on an institution before launching an attack. g. Discuss why whois databases should be publicly available. P19. In this problem, we use the useful dig tool available on Unix and Linux hosts to explore the hierarchy of DNS servers. Recall that in Figure 2.19 , a DNS server in the DNS hierarchy delegates a DNS query to a DNS server lower in the hierarchy, by sending back to the DNS client the name of that lower-level DNS server. First read the man page for dig, and then answer the following questions. a. Starting with a root DNS server (from one of the root servers [a-m].root-servers.net), initiate a sequence of queries for the IP address for your department’s Web server by using dig. Show the list of the names of DNS servers in the delegation chain in answering your query. b. Repeat part (a) for several popular Web sites, such as google.com, yahoo.com, or amazon.com. P20. Suppose you can access the caches in the local DNS servers of your department. Can you propose a way to roughly determine the Web servers (outside your department) that are most popular among the users in your department? Explain. P21. Suppose that your department has a local DNS server for all computers in the department. You are an ordinary user (i.e., not a network/system administrator). Can you determine if an external Web site was likely accessed from a computer in your department a couple of seconds ago? Explain. P22. Consider distributing a file of Gbits to N peers. The server has an upload rate of Mbps, and each peer has a download rate of Mbps and an upload rate of u. For 100, and 1,000 and 700 Kbps, and 2 Mbps, prepare a chart giving

the minimum distribution time for each of the combinations of N and u for both client-server distribution and P2P distribution. P23. Consider distributing a file of F bits to N peers using a client-server architecture. Assume a fluid model where the server can simultaneously transmit to multiple peers, transmitting to each peer at different rates, as long as the combined rate does not exceed u . a. Suppose that Specify a distribution scheme that has a distribution time of NF/u . b. Suppose that Specify a distribution scheme that has a distribution time of F/d . c. Conclude that the minimum distribution time is in general given by P24. Consider distributing a file of F bits to N peers using a P2P architecture. Assume a fluid model. For simplicity assume that dmin is very large, so that peer download bandwidth is never a bottleneck. a. Suppose that Specify a distribution scheme that has a distribution time of F/u . b. Suppose that Specify a distribution scheme that has a distribution time of c. Conclude that the minimum distribution time is in general given by P25. Consider an overlay network with N active peers, with each pair of peers having an active TCP connection. Additionally, suppose that the TCP connections pass through a total of M routers. How many nodes and edges are there in the corresponding overlay network? P26. Suppose Bob joins a BitTorrent torrent, but he does not want to upload any data to any other peers (so called free-riding). a. Bob claims that he can receive a complete copy of the file that is shared by the swarm. Is Bob's claim possible? Why or why not? b. Bob further claims that he can further make his "free-riding" more efficient by using a collection of multiple computers (with distinct IP addresses) in the computer lab in his department. How can he do that? P27. Consider a DASH system for which there are N video versions (at N different rates and qualities) and N audio versions (at N different rates and qualities). Suppose we want to allow the F=15 us=30 di=2 N=10, u=300 Kbps, s us/N≤dmin. s us/N≥dmin. min max{NF/us, F/dmin}. us≤(us+u1+…+uN)/N. s us≥(us+u1+…+uN)/N. NF/(us+u1+…+uN). max{F/us, NF/(us+u1+…+uN)}. Socket Programming Assignments The Companion Website includes six socket programming assignments. The first four assignments are summarized below. The fifth assignment makes use of the ICMP protocol and is summarized at the end of Chapter 5. The sixth assignment employs multimedia protocols and is summarized at the end of Chapter 9. It is highly recommended that students complete several, if not all, of these assignments. Students can find full details of these assignments, as well as important snippets of the Python code, at the Web site www.pearsonhighered.com/cs-resources. Assignment 1: Web Server player to choose at any time any of the N video versions and any of the N audio versions. a. If we create files so that the audio is mixed in with the video, so server sends only one media stream at given time, how many files will the server need to store (each a different URL)? b. If the server instead sends the audio and video streams separately and has the client synchronize the streams, how many files will the server need to store? P28. Install and compile the Python programs TCPClient and UDPClient on one host and TCPServer and UDPServer on another host. a. Suppose you run TCPClient before you run TCPServer. What happens? Why? b. Suppose you run UDPClient before you run UDPServer. What happens? Why? c. What happens if you use different port numbers for the client and server sides? P29. Suppose that in UDPClient.py, after we create the socket, we add the line: clientSocket.bind(('', 5432)) Will it become necessary to change UDPServer.py? What are the port numbers for the sockets in UDPClient and UDPServer? What were they before making this change? P30. Can you configure your browser to open multiple simultaneous connections to a Web site? What are the advantages and disadvantages of having a large number of simultaneous TCP connections? P31. We have seen that Internet TCP sockets treat the data being sent as a byte stream but UDP sockets recognize message boundaries. What are one advantage and one disadvantage of byte-oriented API versus having the API explicitly recognize and preserve application-defined message boundaries? P32. What is the Apache Web server? How much does it cost? What

functionality does it currently have? You may want to look at Wikipedia to answer this question. In this assignment, you will develop a simple Web server in Python that is capable of processing only one request. Specifically, your Web server will (i) create a connection socket when contacted by a client (browser); (ii) receive the HTTP request from this connection; (iii) parse the request to determine the specific file being requested; (iv) get the requested file from the server's file system; (v) create an HTTP response message consisting of the requested file preceded by header lines; and (vi) send the response over the TCP connection to the requesting browser. If a browser requests a file that is not present in your server, your server should return a "404 Not Found" error message. In the Companion Website, we provide the skeleton code for your server. Your job is to complete the code, run your server, and then test your server by sending requests from browsers running on different hosts. If you run your server on a host that already has a Web server running on it, then you should use a different port than port 80 for your Web server. Assignment 2: UDP Pinger In this programming assignment, you will write a client ping program in Python. Your client will send a simple ping message to a server, receive a corresponding pong message back from the server, and determine the delay between when the client sent the ping message and received the pong message. This delay is called the Round Trip Time (RTT). The functionality provided by the client and server is similar to the functionality provided by standard ping program available in modern operating systems. However, standard ping programs use the Internet Control Message Protocol (ICMP) (which we will study in Chapter 5). Here we will create a nonstandard (but simple!) UDP-based ping program. Your ping program is to send 10 ping messages to the target server over UDP. For each message, your client is to determine and print the RTT when the corresponding pong message is returned. Because UDP is an unreliable protocol, a packet sent by the client or server may be lost. For this reason, the client cannot wait indefinitely for a reply to a ping message. You should have the client wait up to one second for a reply from the server; if no reply is received, the client should assume that the packet was lost and print a message accordingly. In this assignment, you will be given the complete code for the server (available in the Companion Website). Your job is to write the client code, which will be very similar to the server code. It is recommended that you first study carefully the server code. You can then write your client code, liberally cutting and pasting lines from the server code. Assignment 3: Mail Client The goal of this programming assignment is to create a simple mail client that sends e-mail to any recipient. Your client will need to establish a TCP connection with a mail server (e.g., a Google mail server), dialogue with the mail server using the SMTP protocol, send an e-mail message to a recipient (e.g., your friend) via the mail server, and finally close the TCP connection with the mail server. For this assignment, the Companion Website provides the skeleton code for your client. Your job is to complete the code and test your client by sending e-mail to different user accounts. You may also try sending through different servers (for example, through a Google mail server and through your university mail server). Assignment 4: Multi-Threaded Web Proxy In this assignment, you will develop a Web proxy. When your proxy receives an HTTP request for an object from a browser, it generates a new HTTP request for the same object and sends it to the origin server. When the proxy receives the corresponding HTTP response with the object from the origin server, it creates a new HTTP response, including the object, and sends it to the client. This proxy will be multi-threaded, so that it will be able to handle multiple requests at the same time. For this assignment, the Companion Website provides the skeleton code for the proxy server. Your job is to complete the code, and then test it by having different browsers request Web objects via your proxy. Wireshark Lab: HTTP Having gotten our feet wet with the Wireshark packet sniffer in Lab 1, we're now ready to use Wireshark to investigate protocols in operation. In this lab, we'll explore several aspects of the HTTP protocol: the basic GET/reply

interaction, HTTP message formats, retrieving large HTML files, retrieving HTML files with embedded URLs, persistent and non-persistent connections, and HTTP authentication and security. As is the case with all Wireshark labs, the full description of this lab is available at this book's Web site, www.pearsonhighered.com/cs-resources. Wireshark Lab: DNS In this lab, we take a closer look at the client side of the DNS, the protocol that translates Internet hostnames to IP addresses. Recall from Section 2.5 that the client's role in the DNS is relatively simple —a client sends a query to its local DNS server and receives a response back. Much can go on under the covers, invisible to the DNS clients, as the hierarchical DNS servers communicate with each other to either recursively or iteratively resolve the client's DNS query. From the DNS client's standpoint, however, the protocol is quite simple—a query is formulated to the local DNS server and a response is received from that server. We observe DNS in action in this lab. As is the case with all Wireshark labs, the full description of this lab is available at this book's Web site, www.pearsonhighered.com/cs-resources. An Interview With... Marc Andreessen Marc Andreessen is the co-creator of Mosaic, the Web browser that popularized the World Wide Web in 1993. Mosaic had a clean, easily understood interface and was the first browser to display images in-line with text. In 1994, Marc Andreessen and Jim Clark founded Netscape, whose browser was by far the most popular browser through the mid-1990s. Netscape also developed the Secure Sockets Layer (SSL) protocol and many Internet server products, including mail servers and SSL-based Web servers. He is now a co-founder and general partner of venture capital firm Andreessen Horowitz, overseeing portfolio development with holdings that include Facebook, Foursquare, Groupon, Jawbone, Twitter, and Zynga. He serves on numerous boards, including Bump, eBay, Glam Media, Facebook, and Hewlett-Packard. He holds a BS in Computer Science from the University of Illinois at Urbana-Champaign. How did you become interested in computing? Did you always know that you wanted to work in information technology? The video game and personal computing revolutions hit right when I was growing up—personal computing was the new technology frontier in the late 70's and early 80's. And it wasn't just Apple and the IBM PC, but hundreds of new companies like Commodore and Atari as well. I taught myself to program out of a book called "Instant Freeze-Dried BASIC" at age 10, and got my first computer (a TRS-80 Color Computer—look it up!) at age 12. Please describe one or two of the most exciting projects you have worked on during your career. What were the biggest challenges? Undoubtedly the most exciting project was the original Mosaic web browser in '92–'93—and the biggest challenge was getting anyone to take it seriously back then. At the time, everyone thought the interactive future would be delivered as "interactive television" by huge companies, not as the Internet by startups. What excites you about the future of networking and the Internet? What are your biggest concerns? The most exciting thing is the huge unexplored frontier of applications and services that programmers and entrepreneurs are able to explore—the Internet has unleashed creativity at a level that I don't think we've ever seen before. My biggest concern is the principle of unintended consequences—we don't always know the implications of what we do, such as the Internet being used by governments to run a new level of surveillance on citizens. Is there anything in particular students should be aware of as Web technology advances? The rate of change—the most important thing to learn is how to learn—how to flexibly adapt to changes in the specific technologies, and how to keep an open mind on the new opportunities and possibilities as you move through your career. What people inspired you professionally? Vannevar Bush, Ted Nelson, Doug Engelbart, Nolan Bushnell, Bill Hewlett and Dave Packard, Ken Olsen, Steve Jobs, Steve Wozniak, Andy Grove, Grace Hopper, Hedy Lamarr, Alan Turing, Richard Stallman. What are your recommendations for students who want to pursue careers in computing and information technology? Go as deep as you possibly can on understanding how technology is

created, and then complement with learning how business works. Can technology solve the world's problems? No, but we advance the standard of living of people through economic growth, and most economic growth throughout history has come from technology—so that's as good as it gets. Chapter 3 Transport Layer Residing between the application and network layers, the transport layer is a central piece of the layered network architecture. It has the critical role of providing communication services directly to the application processes running on different hosts. The pedagogic approach we take in this chapter is to alternate between discussions of transport-layer principles and discussions of how these principles are implemented in existing protocols; as usual, particular emphasis will be given to Internet protocols, in particular the TCP and UDP transport-layer protocols. We'll begin by discussing the relationship between the transport and network layers. This sets the stage for examining the first critical function of the transport layer—extending the network layer's delivery service between two end systems to a delivery service between two application-layer processes running on the end systems. We'll illustrate this function in our coverage of the Internet's connectionless transport protocol, UDP. We'll then return to principles and confront one of the most fundamental problems in computer networking—how two entities can communicate reliably over a medium that may lose and corrupt data. Through a series of increasingly complicated (and realistic!) scenarios, we'll build up an array of techniques that transport protocols use to solve this problem. We'll then show how these principles are embodied in TCP, the Internet's connection-oriented transport protocol. We'll next move on to a second fundamentally important problem in networking—controlling the transmission rate of transport-layer entities in order to avoid, or recover from, congestion within the network. We'll consider the causes and consequences of congestion, as well as commonly used congestion-control techniques. After obtaining a solid understanding of the issues behind congestion control, we'll study TCP's approach to congestion control. 3.1 Introduction and Transport-Layer Services In the previous two chapters we touched on the role of the transport layer and the services that it provides. Let's quickly review what we have already learned about the transport layer. 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. Figure 3.1 illustrates the notion of logical communication. As shown in Figure 3.1, transport-layer protocols are implemented in the end systems but not in network routers. On the sending side, the transport layer converts the application-layer messages it receives from a sending application process into transport-layer packets, known as transport-layer segments in Internet terminology. This is done by (possibly) breaking the application messages into smaller chunks and adding a transport-layer header to each chunk to create the transport-layer segment. The transport layer then passes the segment to the network layer at the sending end system, where the segment is encapsulated within a network-layer packet (a datagram) and sent to the destination. It's important to note that network routers act only on the network-layer fields of the datagram; that is, they do not examine the fields of the transport-layer segment encapsulated with the datagram. On the receiving side, the network layer extracts the transport-layer segment from the datagram and passes the segment up to the transport layer. The transport layer then processes the received segment, making the data in the segment available to the receiving application. More than one transport-layer protocol may be available to network applications. For example, the Internet has two protocols—TCP and UDP. Each of

these protocols provides a different set of transportlayer services to the invoking application. 3.1.1 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 Figure 3.1 The transport layer provides logical rather than physical communication between application processes 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) Continuing with this analogy, note that Ann and Bill do all their work within their respective homes; they are not involved, for example, in sorting mail in any intermediate mail center or in moving mail from one mail center to another. Similarly, transport-layer protocols live in the end systems. Within an end system, a transport protocol moves messages from application processes to the network edge (that is, the network layer) and vice versa, but it doesn't have any say about how the messages are moved within the network core. In fact, as illustrated in Figure 3.1, intermediate routers neither act on, nor recognize, any information that the transport layer may have added to the application messages. Continuing with our family saga, suppose now that when Ann and Bill go on vacation, another cousin pair—say, Susan and Harvey—substitute for them and provide the household-internal collection and delivery of mail. Unfortunately for the two families, Susan and Harvey do not do the collection and delivery in exactly the same way as Ann and Bill. Being younger kids, Susan and Harvey pick up and drop off the mail less frequently and occasionally lose letters (which are sometimes chewed up by the family dog). Thus, the cousin-pair Susan and Harvey do not provide the same set of services (that is, the same service model) as Ann and Bill. In an analogous manner, a computer network may make = = = = = available multiple transport protocols, with each protocol offering a different service model to applications. The possible services that Ann and Bill can provide are clearly constrained by the possible services that the postal service provides. For example, if the postal service doesn't provide a maximum bound on how long it can take to deliver mail between the two houses (for example, three days), then there is no way that Ann and Bill can guarantee a maximum delay for mail delivery between any of the cousin pairs. In a similar manner, the services that a transport protocol can provide

are often constrained by the service model of the underlying network-layer protocol. If the network-layer protocol cannot provide delay or bandwidth guarantees for transport-layer segments sent between hosts, then the transport-layer protocol cannot provide delay or bandwidth guarantees for application messages sent between processes. Nevertheless, certain services can be offered by a transport protocol even when the underlying network protocol doesn't offer the corresponding service at the network layer. For example, as we'll see in this chapter, a transport protocol can offer reliable data transfer service to an application even when the underlying network protocol is unreliable, that is, even when the network protocol loses, garbles, or duplicates packets. As another example (which we'll explore in Chapter 8 when we discuss network security), a transport protocol can use encryption to guarantee that application messages are not read by intruders, even when the network layer cannot guarantee the confidentiality of transport-layer segments. 3.1.2 Overview of the Transport Layer in the Internet Recall that the Internet 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, 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. (We'll learn about the network layer in detail in Chapters 4 and 5.) The Internet's network-layer protocol has a name—IP, for Internet Protocol. IP provides logical communication between hosts. 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 no guarantees. 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 data in the segments. For these reasons, IP is said to be an unreliable service. We also mention here that every host has at least one networklayer address, a so-called IP address. We'll examine IP addressing in detail in Chapter 4; for this chapter we need only keep in mind that each host has an IP address. Having taken a glimpse at the IP service model, let's now summarize the service models provided by UDP and TCP. The most fundamental responsibility of UDP and TCP is to extend IP's delivery service between two end systems to a delivery service between two processes running on the end systems. Extending host-to-host delivery to process-to-process delivery is called transport-layer multiplexing and demultiplexing. We'll discuss transport-layer multiplexing and demultiplexing in the next section. UDP and TCP also provide integrity checking by including error-detection fields in their segments' headers. These two minimal transport-layer services—process-to-process data delivery and error checking—are the only two services that UDP provides! In particular, like IP, UDP is an unreliable service—it does not guarantee that data sent by one process will arrive intact (or at all!) to the destination process. UDP is discussed in detail in Section 3.3. TCP, on the other hand, offers several additional services to applications. First and foremost, it provides reliable data transfer. Using flow control, sequence numbers, acknowledgments, and timers

(techniques we'll explore in detail in this chapter), TCP ensures that data is delivered from sending process to receiving process, correctly and in order. TCP thus converts IP's unreliable service between end systems into a reliable data transport service between processes. TCP also provides congestion control. Congestion control is not so much a service provided to the invoking application as it is a service for the Internet as a whole, a service for the general good. Loosely speaking, TCP congestion control prevents any one TCP connection from swamping the links and routers between communicating hosts with an excessive amount of traffic. TCP strives to give each connection traversing a congested link an equal share of the link bandwidth. This is done by regulating the rate at which the sending sides of TCP connections can send traffic into the network. UDP traffic, on the other hand, is unregulated. An application using UDP transport can send at any rate it pleases, for as long as it pleases. A protocol that provides reliable data transfer and congestion control is necessarily complex. We'll need several sections to cover the principles of reliable data transfer and congestion control, and additional sections to cover the TCP protocol itself. These topics are investigated in Sections 3.4 through 3.8. The approach taken in this chapter is to alternate between basic principles and the TCP protocol. For example, we'll first discuss reliable data transfer in a general setting and then discuss how TCP specifically provides reliable data transfer. Similarly, we'll first discuss congestion control in a general setting and then discuss how TCP performs congestion control. But before getting into all this good stuff, let's first look at transport-layer multiplexing and demultiplexing. 3.2 Multiplexing and Demultiplexing In this section, we discuss transport-layer multiplexing and demultiplexing, that is, extending the host-tohost 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. You therefore have four network application processes running—two Telnet processes, one FTP process, and one HTTP process. When the transport layer in your computer receives data from the network layer below, it needs to direct the received data to one of these four processes. Let's now examine how this is done. First recall from Section 2.7 that a process (as part of a network application) can have one or more sockets, doors through which data passes from the network to the process and through which data passes from the process to the network. Thus, as shown in Figure 3.2, the transport layer in the receiving host does not actually deliver data directly to a process, but instead to an intermediary socket. Because at any given time there can be more than one socket in the receiving host, each socket has a unique identifier. The format of the identifier depends on whether the socket is a UDP or a TCP socket, as we'll discuss shortly. 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 Figure 3.2 Transport-layer multiplexing and demultiplexing in Figure 3.2

must demultiplex segments arriving from the network layer below to either process P or P 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 transportlayer segments, and pass these segments down to the network layer. Although we have introduced multiplexing and demultiplexing in the context of the Internet transport protocols, it's important to realize that they are concerns whenever a single protocol at one layer (at the transport layer or elsewhere) is used by multiple protocols at the next higher layer. To illustrate the demultiplexing job, recall the household analogy in the previous section. Each of the kids is identified by his or her name. When Bill receives a batch of mail from the mail carrier, he performs a demultiplexing operation by observing to whom the letters are addressed and then hand delivering the mail to his brothers and sisters. Ann performs a multiplexing operation when she collects letters from her brothers and sisters and gives the collected mail to the mail person. Now that we understand the roles of transport-layer multiplexing and demultiplexing, let us examine how it is actually done in a host. From the discussion above, we know that transport-layer multiplexing requires (1) that sockets have unique identifiers, and (2) that each segment have special fields that indicate the socket to which the segment is to be delivered. These special fields, illustrated in Figure 3.3, are the source port number field and the destination port number field. (The UDP and TCP segments have other fields as well, as discussed in the subsequent sections of this chapter.) Each port number is a 16-bit number, ranging from 0 to 65535. The port numbers ranging from 0 to 1023 are called well-known port numbers and are restricted, which means that they are reserved for use by well-known 1 2 Figure 3.3 Source and destination port-number fields in a transport-layer segment application protocols such as HTTP (which uses port number 80) and FTP (which uses port number 21). The list of well-known port numbers is given in RFC 1700 and is updated at http://www.iana.org [RFC 3232]. When we develop a new application (such as the simple application developed in Section 2.7), we must assign the application a port number. It should now be clear how the transport layer could implement the demultiplexing service: Each socket in the host could be assigned a port number, and when a segment arrives at the host, the transport layer examines the destination port number in the segment and directs the segment to the corresponding socket. The segment's data then passes through the socket into the attached process. As we'll see, this is basically how UDP does it. However, we'll also see that multiplexing/demultiplexing in TCP is yet more subtle. Connectionless Multiplexing and Demultiplexing Recall from Section 2.7.1 that the Python program running in a host can create a UDP socket with the line clientSocket = socket(AF_INET, 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. With port numbers assigned to UDP sockets, we can now precisely describe UDP multiplexing/demultiplexing. Suppose a process in Host A, with UDP port 19157, wants to send a chunk of application data to a process with UDP port 46428 in Host B. The transport layer in Host A creates a transport-layer segment that includes the application data, the source port number (19157), the destination port number (46428), and two other values (which will be

discussed later, but are unimportant for the current discussion). The transport layer then passes the resulting segment to the network layer. The network layer encapsulates the segment in an IP datagram and makes a best-effort attempt to deliver the segment to the receiving host. If the segment arrives at the receiving Host B, the transport layer at the receiving host examines the destination port number in the segment (46428) and delivers the segment to its socket identified by port 46428. Note that Host B could be running multiple processes, each with its own UDP socket and associated port number. As UDP segments arrive from the network, Host B directs (demultiplexes) each segment to the appropriate socket by examining the segment's destination 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. You may be wondering now, what is the purpose of the source port number? As shown in Figure 3.4, in the A-to-B segment the source port number serves as part of a "return address"—when B wants to send a segment back to A, the destination port in the B-to-A segment will take its value from the source port value of the A-to-B segment. (The complete return address is A's IP address and the source port number.) As an example, recall the UDP server program studied in Section 2.7. In UDPServer.py , the server uses the recvfrom() method to extract the client-side (source) port number from the segment it receives from the client; it then sends a new segment to the client, with the extracted source port number serving as the destination port number in this new segment. 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. Figure 3.4 The inversion of source and destination port numbers 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 connectionestablishment request) be directed to two different sockets. To gain further insight, let's reconsider the TCP client-server programming example in Section 2.7.2: The TCP server application has a "welcoming socket," that waits for connection-establishment requests from TCP clients (see Figure 2.29) 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)) A connection-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. The server host may support many simultaneous TCP connection sockets, with each socket attached to a process, and with each socket identified by its own four-tuple. When a TCP segment arrives at the host, all four fields (source IP address, source port, destination IP address, destination port) are used to direct (demultiplex) the segment to the appropriate socket. FOCUS ON SECURITY Port Scanning We've seen that a server process waits patiently on an open port for contact by a remote client. Some ports are reserved for well-known applications (e.g., Web, FTP, DNS, and SMTP servers); other ports are used by convention by popular applications (e.g., the Microsoft 2000 SQL server listens for requests on UDP port 1434). Thus, if we determine that a port is open on a host, we may be able to map that port to a specific application running on the host. This is very useful for system administrators, who are often interested in knowing which network applications are running on the hosts in their networks. But attackers, in order to "case the joint," also want to know which ports are open on target hosts. If a host is found to be running an application with a known security flaw (e.g., a SQL server listening on port 1434 was subject to a buffer overflow, allowing a remote user to execute arbitrary code on the vulnerable host, a flaw exploited by the Slammer worm [CERT 2003–04]), then that host is ripe for attack. Determining which applications are listening on which ports is a relatively easy task. Indeed there are a number of public domain programs, called port scanners, that do just that. Perhaps the most widely used of these is nmap, freely available at http://nmap.org and included in most Linux distributions. For TCP, nmap sequentially scans ports, looking for ports that are accepting TCP connections. For UDP, nmap again sequentially scans ports, looking for UDP ports that respond to transmitted UDP segments. In both cases, nmap returns a list of open, closed, or unreachable ports. A host running nmap can attempt to scan any target host anywhere in the Internet. We'll revisit nmap in Section 3.5.6, when we discuss TCP connection management. Figure 3.5 Two clients, using the same destination port number (80) to communicate with the same Web server application The situation is illustrated in Figure 3.5, in which Host C initiates two HTTP sessions to server B, and Host A initiates one HTTP session to B. Hosts A and C and server B each have their own unique IP address—A, C, and B, respectively. Host C assigns two different source port numbers (26145 and 7532) to its two HTTP connections. Because Host A is choosing source port numbers independently of C, it might also assign a source port of 26145 to its HTTP connection. But this is not a problem—server B will still be able to correctly demultiplex the two connections having the same source port number, since the two connections have different source IP addresses. Web Servers and TCP Before closing this discussion, it's instructive to say a few additional words about Web servers and how they use port numbers. Consider a host running a Web server, such as an Apache Web server, on port 80. When clients (for example, browsers) send segments to the server, all segments will have destination port 80. In particular, both the initial connection-establishment segments and the segments carrying HTTP request messages will have destination port 80. As we have just described, the server distinguishes the segments from the different clients using source IP addresses and source port numbers. 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 (although a number of operating system tricks can be used to mitigate the problem). Readers interested in the operating system issues surrounding persistent and non-persistent HTTP are encouraged to see [Nielsen 1997; Nahum 2002]. Now that we've discussed transport-layer multiplexing and demultiplexing, let's move on and discuss one of the Internet's transport protocols, UDP. In the next section we'll see that UDP adds little more to the network-layer protocol than a multiplexing/demultiplexing service. 3.3 Connectionless Transport: UDP In this section, we'll take a close look at UDP, how it works, and what it does. We encourage you to refer back to Section 2.1, which includes an overview of the UDP service model, and to Section 2.7.1, which discusses socket programming using UDP. To motivate our discussion about UDP, suppose you were interested in designing a no-frills, bare-bones transport protocol. How might you go about doing this? You might first consider using a vacuous transport protocol. In particular, on the sending side, you might consider taking the messages from the application process and passing them directly to the network layer; and on the receiving side, you might consider taking the messages arriving from the network layer and passing them directly to the application process. But as we learned in the previous section, we have to do a little more than nothing! At the very least, the transport layer has to provide a multiplexing/demultiplexing service in order to pass data between the network layer and the correct application-level process. 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), it might try resending the query, try sending the query to another name server, or inform the invoking application that it can't get a reply. Now you might be wondering why an application developer would ever choose to build an application over UDP rather than over TCP. Isn't TCP always preferable, since TCP provides a reliable data transfer service, while UDP does not? The answer is no, as some 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 congestioncontrol 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. As discussed below, these applications can use UDP and implement, as part of the application, any additional functionality that is needed beyond UDP's no-frills segment-delivery service. 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. Indeed, the QUIC protocol (Quick UDP Internet Connection, [Iyengar 2015]), used in Google's Chrome browser, uses UDP as its underlying transport protocol and implements reliability in an application-layer protocol on top of UDP. 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. Figure 3.6 lists popular Internet applications and the transport protocols that they use. As we expect, email, remote terminal access, the Web, and file transfer run over TCP—all these applications need the reliable data transfer service of TCP. Nevertheless, many important applications run over UDP rather than TCP. For example, UDP is used to carry network management (SNMP; see Section 5.7) data. UDP is preferred to TCP in this case, since network management applications must often run when the network is in a stressed state—precisely when reliable, congestion-controlled data transfer is difficult to achieve. Also, as we mentioned earlier, DNS runs over UDP, thereby avoiding TCP's connectionestablishment delays. As shown in Figure 3.6, both UDP and TCP are somtimes used today with multimedia applications, such as Internet phone, real-time video conferencing, and streaming of stored audio and video. We'll take a close look at these applications in Chapter 9. We just mention now that all of these applications can tolerate a small amount of packet loss, so that reliable data transfer is not absolutely critical for the application's success. Furthermore, real-time applications, like Internet phone and video conferencing, react very poorly to TCP's congestion control. For these reasons, developers of multimedia applications may choose to run their applications over UDP instead of TCP. When packet loss rates are low, and with some organizations blocking UDP traffic for security reasons (see Chapter 8), TCP becomes an increasingly attractive protocol for streaming media transport. Figure 3.6 Popular Internet applications and their underlying transport protocols Although commonly done today, running multimedia applications over UDP is controversial. As we mentioned above, UDP has no congestion control. But congestion control is needed to prevent the network from entering a congested state in which very little useful work is done. If

everyone were to start streaming high-bit-rate video without using any congestion control, there would be so much packet overflow at routers that very few UDP packets would successfully traverse the source-to-destination path. Moreover, the high loss rates induced by the uncontrolled UDP senders would cause the TCP senders (which, as we'll see, do decrease their sending rates in the face of congestion) to dramatically decrease their rates. Thus, the lack of congestion control in UDP can result in high loss rates between a UDP sender and receiver, and the crowding out of TCP sessions—a potentially serious problem [Floyd 1999]. Many researchers have proposed new mechanisms to force all sources, including UDP sources, to perform adaptive congestion control [Mahdavi 1997; Floyd 2000; Kohler 2006: RFC 4340]. Before discussing the UDP segment structure, we mention that it is possible for an application to have reliable data transfer when using UDP. This can be done if reliability is built into the application itself (for example, by adding acknowledgment and retransmission mechanisms, such as those we'll study in the next section). We mentioned earlier that the QUIC protocol [Iyengar 2015] used in Google's Chrome browser implements reliability in an application-layer protocol on top of UDP. But this is a nontrivial task that would keep an application developer busy debugging for a long time. Nevertheless, building reliability directly into the application allows the application to "have its cake and eat it too. That is, application processes can communicate reliably without being subjected to the transmission-rate constraints imposed by TCP's congestion-control mechanism. 3.3.1 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. But we ignore this detail in order to see the forest through the trees. We'll discuss the checksum calculation below. Basic principles of error detection are described in Section 6.2. The length field specifies the length of the UDP segment, including the header, in bytes. 3.3.2 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. Figure 3.7 UDP segment structure 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. You may wonder why UDP provides a checksum in the first place, as many link-layer protocols (including the popular Ethernet protocol) also provide error checking. The reason is that there is no guarantee that all the links between source and destination provide error checking; that is, one of the links may use a link-layer protocol that does not provide error checking. Furthermore, even if segments are correctly transferred across a link, it's possible that bit errors could be introduced when a segment is stored in a router's memory. Given that neither link-by-link reliability nor in-memory error detection is guaranteed, UDP must provide error detection at the transport layer, on an end-end basis, if the endend data transfer service is to provide error detection. This is an example of the celebrated end-end principle in system design [Saltzer 1984], which states that since certain functionality (error detection, in this case) must be implemented on an end-end basis: "functions placed at the lower levels may be redundant or of little value when compared to the cost of providing them at the higher level." Because IP is supposed to run over just about any layer-2 protocol, it is useful for the transport layer to provide error checking as a safety measure. Although UDP provides error checking, it does not do anything to recover from an error. Some implementations of UDP simply discard the damaged segment; others pass the damaged segment to the application with a warning. That wraps up our discussion of UDP. We will soon see that TCP offers reliable data transfer to its applications as well as other services that UDP doesn't offer. Naturally, TCP is also more complex than UDP. Before discussing TCP, however, it will be useful to step back and first discuss the underlying principles of reliable data transfer. 3.4 Principles of Reliable Data Transfer In this section, we consider the problem of reliable data transfer in a general context. This is appropriate since the problem of implementing reliable data transfer occurs not only at the transport layer, but also at the link layer and the application layer as well. The general problem is thus of central importance to networking. Indeed, if one had to identify a "top-ten" list of fundamentally important problems in all of networking, this would be a candidate to lead the list. In the next section we'll examine TCP and show, in particular, that TCP exploits many of the principles that we are about to describe. Figure 3.8 illustrates the framework for our study of reliable data transfer. The service abstraction provided to the upper-layer entities is that of a reliable channel through which data can be transferred. With a reliable channel, no transferred data bits are corrupted (flipped from 0 to 1, or vice versa) or lost, and all are delivered in the order in which they were sent. This is precisely the service model offered by TCP to the Internet applications that invoke it. 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. In this section, we will incrementally develop the sender and receiver sides of a reliable data transfer protocol, considering increasingly complex models of the underlying channel. For example, we'll consider what protocol mechanisms are Figure 3.8 Reliable data transfer: Service model and service implementation needed when the underlying channel can corrupt bits or lose entire packets. One assumption we'll adopt throughout our discussion here is that packets will be delivered in the order in which they were sent, with some packets possibly being lost; that is, the underlying channel will not reorder packets. Figure 3.8(b) illustrates the interfaces for our data transfer

protocol. The sending side of the data transfer protocol will be invoked from above by a call to rdt_send() . It will pass the data to be delivered to the upper layer at the receiving side. (Here rdt stands for reliable data transfer protocol and _send indicates that the sending side of rdt is being called. The first step in developing any protocol is to choose a good name!) On the receiving side, rdt_rcv() will be called when a packet arrives from the receiving side of the channel. When the rdt protocol wants to deliver data to the upper layer, it will do so by calling deliver_data() . In the following we use the terminology “packet” rather than transport-layer “segment.” Because the theory developed in this section applies to computer networks in general and not just to the Internet transport layer, the generic term “packet” is perhaps more appropriate here. In this section we consider only the case of unidirectional data transfer, that is, data transfer from the sending to the receiving side. The case of reliable bidirectional (that is, full-duplex) data transfer is conceptually no more difficult but considerably more tedious to explain. Although we consider only unidirectional data transfer, it is important to note that the sending and receiving sides of our protocol will nonetheless need to transmit packets in both directions, as indicated in Figure 3.8. We will see shortly that, in addition to exchanging packets containing the data to be transferred, the sending and receiving sides of rdt will also need to exchange control packets back and forth. Both the send and receive sides of rdt send packets to the other side by a call to udt_send() (where udt stands for unreliable data transfer). 3.4.1 Building a Reliable Data Transfer Protocol We now step through a series of protocols, each one becoming more complex, arriving at a flawless, reliable data transfer protocol. Reliable Data Transfer over a Perfectly Reliable Channel: rdt1.0 We first consider the simplest case, in which the underlying channel is completely reliable. The protocol itself, which we’ll call rdt1.0 , is trivial. The finite-state machine (FSM) definitions for the rdt1.0 sender and receiver are shown in Figure 3.9. The FSM in Figure 3.9(a) defines the operation of the sender, while the FSM in Figure 3.9(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 3.9 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 3.9 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. When no action is taken on an event, or no event occurs and an action is taken, we’ll use the symbol Λ 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. Figure 3.9 rdt1.0 – A protocol for a completely reliable channel 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 lower-layer protocol. In this simple protocol, there is no difference between a unit of data and a packet. Also, all packet flow is from the sender to receiver; with a perfectly reliable channel there is no need for the receiver side to provide any feedback to the sender since nothing can go wrong! Note that we have also assumed that the receiver is able to receive data as fast as the sender happens to send data. Thus, there is no