# CHAPTER 2

## PRACTICE SET

### Questions

**Q2-1.** No changes are needed. The new protocol needs to use the services provided by one of the transport-layer protocols.

**Q2-3.** A server should always be on because a client may need to access it at any time. A client is normally the initializer of the connection; it can be run when it is needed.

**Q2-5.** Probably Alice turned off her desktop, which stopped the FTP server, when she left the office. A server process should be running all the time, waiting for clients to access it.

**Q2-7.** In this case, UDP is more appropriate because it does not have the overhead of TCP in connection establishment and teardown.

**Q2-9.** The IP address identifies the source computer; the port number identifies the source process.

**Q2-11.** The program is an example of a dynamic document because it needs to be run at the server site and only the result is to be downloaded to the client site.

**Q2-13.** If Bob posts his clip on his website, Alice can get it by running an HTTP client (a browser) using a GET message. Since Alice is not running an HTTP server, she needs to use the PUT command and post her clip on Bob's site.

**Q2-15.** If the message has no body section, the lack of any character after the blank line is an indication that the message is over. If the message has a body, either the body needs to have an end-of-file marker or the message should have the Content-Length header to define the size of the body.

**Q2-17.** The answer is yes. Two separate connections are needed for setup and teardown to use FTP.

**Q2-19.** In FTP, the control connection is actively opened by the client. The server is running all the time waiting for a client to make a control connection. The data-transfer connection, on the other hand, is actively opened by the server. The client issues a passive open and sends the ephemeral port (to be used for the data-transfer connection) to the server, which is done using a control-connection command. The server now issues an active open using the ephemeral port received from the client.

**Q2-21.** The task can be done using only one control connection, but two data-transfer connections are needed, one for retrieving and one for storing. Although the data-transfer connection is a two-way connection, one is used for data transfer, the other for acknowledging.

**Q2-23.** The answer is no. Only the client can get the list of files from the server. In the client-server paradigm, the request is from the client; the server only can respond.

**Q2-25.** FTP has a specific format for exchanging commands and responses during the control connection. We can say that these formats can be defined as shown below:

| | |
|---|---|
| **Command:** | **<command> [<space> <parameter>** …] |
| **Response:** | **<code> [<space> <text>**] **<CRLF>** |

**Q2-27.** We can have a control connection without a data-transfer connection in FTP. As a matter of fact, some tasks in FTP can be done without a data-transfer connection. For example, if a client needs to rename a file at the server site, there is no need for a data-transfer connection.

**Q2-29.** The file needs to be transferred as an *image file* (binary), which means the downloaded file should be the bit-by-bit copy of the audio file on the server.

**Q2-31.** The HELO command is needed so that the client can identify itself using its domain name during connection establishment. The MAIL FROM message is needed to supply the server with a return mail address for returning errors and reporting messages.

**Q2-33.** Mail servers normally allocate a limited amount of storage for each mail-box. If the e-mails or attachments are not retrieved, the mail-box may become full and some old e-mails may be discarded.

**Q2-35.** TELNET allows a host to log into a remote computer that can offer application programs. After the user logs in, she can use any services provided by the remote computer. For example, the user can create a program in any computer language supported by the remote computer, compile it, run it, and see the result.

**Q2-37.**

**a.** PQDN (It does not end with a dot.)

**b.** PQDN (It does not end with a dot.)

**c.** FQDN (It does end with a dot.)

**Q2-39.**

**a.** In the direct method, the file is stored in node 20.

**b.** In the indirect method, the file is stored in node 4, but a reference is given in node 20.

**Q2-41.** For this network, $m = \log_2 1024 = 10$. This means each identifier has 10 bits. The height of the tree is also 10. The number of leaves is 1024. Each node has 10 subtrees. Each routing table also has 10 rows.

## Problems

**P2-1.** The following shows a possible request and response:

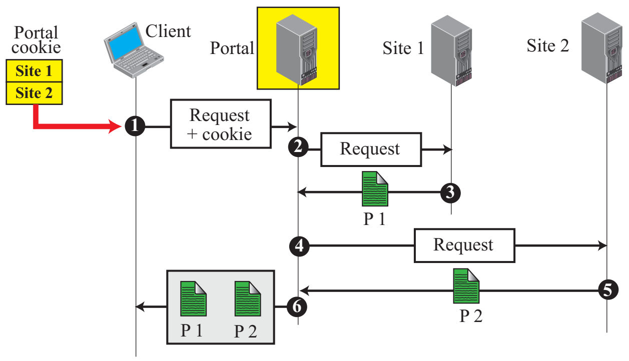
**a.** A possible request

```
GET /usr/users/doc HTTP /1.1
Date: Fri, 26-Nov-04 16:46:23 GMT
MIME-version: 1.0
Accept: image/gif
Accept: image/jpeg
Last modified: Mon, 22-Nov-04
```

**b.** A possible response

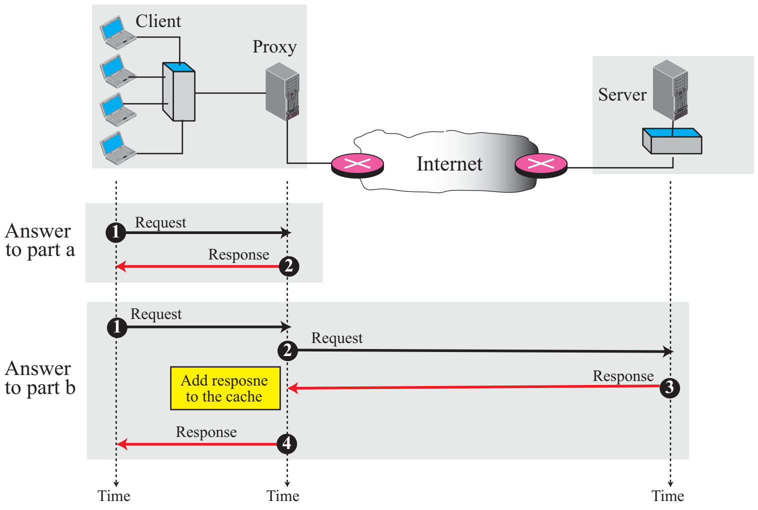
```
HTTP/1.1 200 OK
Date: Fri, 26-Nov-04 16:46:26 GMT
Server: Challenger
MIME-version: 1.0
Content-length: 4623
(Body of document)
```

**P2-3.** The following shows a simple situation. A *portal* is a special site that holds



the often-visited URLs for each client. The cookie stored in the browser under the name of the portal holds the list of the sites the user normally needs to check periodically. When the user clicks on the portal web page, a request is sent with the cookie to the portal site with the list of desired web pages. The portal then fetches the current pages from the corresponding site, compiles a page, and sends it to the browser.

**P2-5.** The following shows a simple example. In part a, the request can be responded to by the proxy server. In part b, the proxy needs to send the request to the true server. When the response is received, the proxy server saves it in the cache for future use, and then sends it to the client.



**P2-7.** The client or server can use one of the general headers called *Connection*, and set the *connection-token* = *close* to define a nonpersistent connection.

**P2-9.**

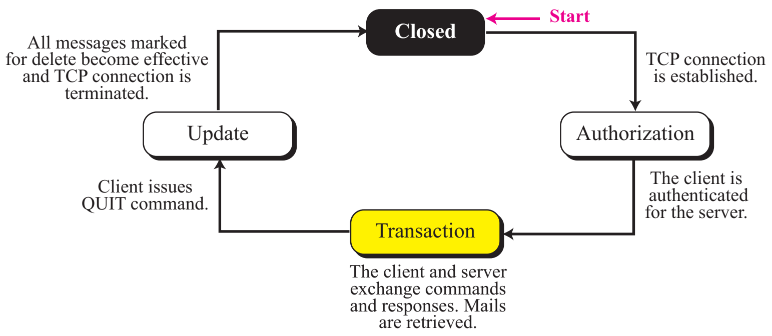
MIME-version: 1.1

Content-Type: Text/Plain
Content-Transfer-Encoding: 7bit

**P2-11.** The result is "Vw/w" in ASCII as shown below:

| | | | | |
|---|---|---|---|---|
| **Original:** | **01010111 00001111 11110000** | | | |
| **Grouped by six:** | 010101 | 110000 | 111111 | 110000 |
| **Base64:** | 21 | 48 | 63 | 48 |
| **ASCII:** | V | w | / | w |

**P2-13.** The following shows the four states and the transitions from one state to another.


Closed
Start
All messages marked for delete become effective and TCP connection is terminated.
TCP connection is established.
Update
Authorization
Client issues QUIT command.
The client is authenticated for the server.
Transaction
The client and server exchange commands and responses. Mails are retrieved.

**P2-15.** The following gives the meaning and usage of each command:

**a.** The UIDL (unique ID listing) is used by the client to provide a unique identifier for the message (if used with an argument) or for all messages (if no argument is used).

**b.** The TOP command, always with two arguments, is used by the client to ask the server to return the top lines of a particular message. In this case, it means to return the top 15 lines of message 1.

**c.** The USER command is used by the client to define its user identification.

**d.** The PASS command is used by the client to define its password.

**P2-17.** The following shows an example. We have shown only part of the transaction (when the client starts retrieving the message):

```
Client: RETR 1
Server: +OK 230 octets
Server: …                                   // Message contents
Server: .                                   // Dot means end of mail
Client: DELE 1
Server: +OK message 1 deleted
Client: RETR 2
Server: OK 400 octets
Server: …                                   // Message contents
Server: .                                   // Dot means end of mail
Client: DELE 2
Server: +OK message 2 deleted
```

**P2-19.** SMTP does not create a session between the client and sender in which the client can send some e-mails in a single session. The e-mails need to be sent one by one. However, POP3 allows the user to retrieve all e-mails received in the mailbox in one session.

**P2-21.** The following shows the commands and responses. The file mode in this case needs to be C (compressed).

| | |
|---|---|
| **Server:** | **220 (Service ready)** |
| **Client:** | **USER Jane** |
| **Server:** | **331 (User name OK. Password?)** |
| **Client:** | **PASS xxxx** |
| **Server:** | **220 (User login OK)** |
| **Client:** | **PORT 61017** |
| **Server:** | **150 (Data connection will open shortly)** |
| **Client:** | **STRU F** |
| **Server:** | **200 (OK)** |
| **Client:** | **TYPE E** |
| **Server:** | **200 (OK)** |
| **Client:** | **MODE C** |
| **Server:** | **200 (OK)** |
| **Client:** | **RETR /usr/users/report/huge** |
| **Server:** | **125 (Data connection OK)** |
| **Transferring the file from the server to the client** | |
| **Server:** | **226 (Closing data connection)** |
| **Client:** | **Quit** |
| **Server:** | **221 (Server closing)** |

**P2-23.** The following shows the set of commands and responses. Note that there is no data transfer connection in this; we have only a control connection. We need to give the name of the file to be renamed (/usr/users/report/file1) and then the new name (/usr/top/letters/file1).

| | |
|---|---|
| **Server:** | **220 (Service ready)** |
| **Client:** | **USER Maria** |
| **Server:** | **331 (User name OK. Password?)** |
| **Client:** | **PASS xxxx** |
| **Server:** | **220 (User login OK)** |
| **Client:** | **RNFR** */usr/users/report/file1* |
| **Server:** | **200 (Command OK)** |
| **Client:** | **RNTO** */usr/top/letters/file1* |
| **Server:** | **200 (Command OK)** |
| **Client:** | **Quit** |
| **Server:** | **221 (Server closing)** |

**P2-25.** FTP simulates a session by creating a control connection. When the control connection is created between a client and a server, file transfer can continue as long as the control connection still exists.

**P2-27.** For a node to be the predecessor of a key, the key needs to be a member of the half-open interval made of the node and its successor node. In other words, *key* ∈ (12, 17]. Note that the interval is 13, 14, 15, 16, and 17; it does not include 12, but it includes 17.

**a.** k12 ∉ (12, 17], which means that N12 is not the predecessor of k12.

**b.** k15 ∈ (12, 17], which means that N12 is the predecessor of k15.

**c.** k17 ∈ (12, 17], which means that N12 is the predecessor of k17.

**d.** k22 ∉ (12, 17], which means that N12 is not the predecessor of k22.

**P2-29.** A node can quickly find if it is the predecessor of a key when key ∈ (node, successor]. If the key is not the predecessor, the node needs to find the closest predecessor. For this purpose, it needs to check the entries in the finger table (from bottom to top) to find a node, between itself and the key, that is closest to the key as far as the information in the table can tell. The table needs to be searched from bottom to top because many entries can be between the node and the key, but the closest to the key is definitely the one which is found first if we start from the bottom of the table. When the closest predecessor is found, the node can ask it to continue with the search.

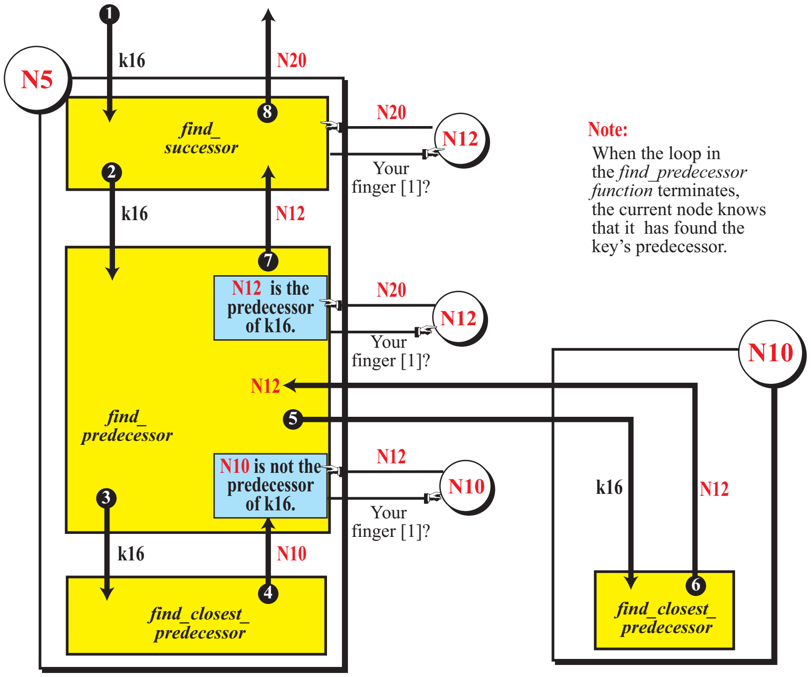
**a.** Since k1 ∉ (N2, N4], N2 is not the predecessor of k1. Node N2 needs to find the closest predecessor of k1. The table is searched from the bottom. N12 ∈ (N2, k1) because 1 here means 17. So, as far as N2 knows, N12 is the closest predecessor for k1. N2 will ask N12 to search further.

**b.** Since k6 ∉ (N2, N4], N2 is not the predecessor of k6. Node N2 needs to find the closest predecessor of k6. The table is searched from the bottom. N12 ∉ (N2, k6), N10 ∉ (N2, k6), N7 ∉ (N2, k6), but N4 ∈ (N2, k6). So, as far as N2 knows, N4 is the closest predecessor for k6. N2 will ask N4 to search further.

**c.** Since k9 ∉ (N2, N4], N2 is not the predecessor of k9. Node N2 needs to find the closest predecessor of k9. The table is searched from the bottom. N12 ∉ (N2, k9), N10 ∉ (N2, k9), but N7 ∈ (N2, k9). So, as far as N2 knows, N7 is the closest predecessor for k9. N2 will ask N7 to search further.

**d.** Since k13 ∉ (N2, N4], N2 is not the predecessor of k13. Node N2 needs to find the closest predecessor of k13. The table is searched from the bottom. N12 ∈ (N2, k13), so, as far as N2 knows, N12 is the closest predecessor for k13. N2 will ask N12 to search further.

**P2-31.** Let us go through the event as we did in Example 2.16 in the text. The following shows how N5 with some help from N10 and N12 goes through the sequence of events to find the successor of k16:


N5
k16
N20
find_ successor
N20
N12
Your finger [1]?
k16
N12
N12 is the predecessor of k16.
N20
N12
Your finger [1]?
N10
N12
find_ predecessor
N10 is not the predecessor of k16.
N12
N10
Your finger [1]?
k16
N12
k16
N10
find_closest_ predecessor
find_closest_ predecessor
Note:
When the loop in the find_predecessor function terminates, the current node knows that it has found the key's predecessor.

a. Event 1: Since N5 is not the responsible node for k16, it calls its *find_successor* function.

b. Event 2: N5 calls the *find_predecessor* function.

c. Event 3: Since N5 is not the predecessor node of k16, N5 calls the *find_closest_predecessor* function. This function checks the finger table of N5 from bottom to top. It finds that N10 is the closest predecessor because $10 \in (5, 16)$.

d. Event 4: The *find_closest_predecessor* function returns N10 to the *find_predecessor* function as the closest predecessor of k16. N5 finds that N10 is not the predecessor of k16 (by remotely calling the N10.finger[1]).

e. Event 5: N5 remotely asks N10 to find the closest predecessor of k16.

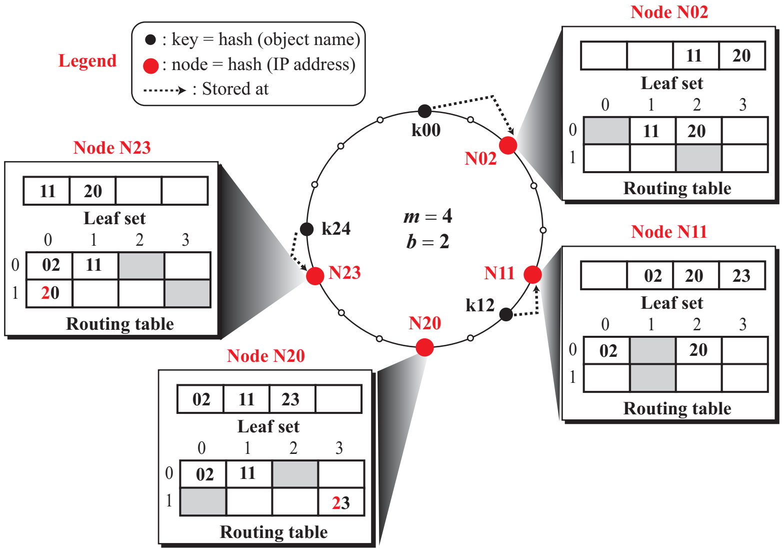
f. Event 6: Node 10 finds that N12 is the closest predecessor of k16 according to its knowledge. It returns N12 to N5. N5 finds that N12 is actually the predecessor of k16 (by asking N12.finger[1].

g. Event 7: The *find_predecessor* function returns N12, as the predecessor of k16, to the *find_successor* function.

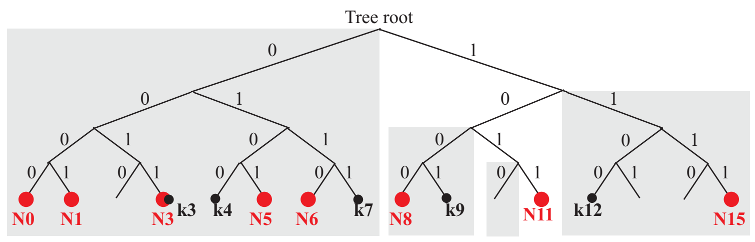
**h.** Event 8: Node N5 finds that the successor node of k16 is N20 by asking N12 to send its finger [1]).

**P2-33.** We have $n = m/b = 32/4 = 8$. The routing table is of size $n \times 2^b$ or $8 \times 16$ (8 rows and 16 columns). The leaf set is a table of one row and 16 columns.

**P2-35.** The following shows the ring, nodes, leaf sets, and routing tables.



**P2-37.** The following shows the four subtrees.



**P2-39.** The node identities in binary are N2(0010), N3(0011), N7(0111), N10(1010), and N12(1100). The following shows the routing tables. The explanation follows:

| | N2 | N3 | N7 | N10 | N12 |
|---|---|---|---|---|---|
| 0 | N10 | N10 | N12 | N2 | N7 |
| 1 | N3 | N2 | N3 | N12 | N10 |
| 2 | N3 | N2 | | | |
| 3 | N3 | N2 | | | |

- **a.** Node N2 has no common prefix with node N10 and node N12, but it is closest to N10 (using XOR criteria). It has one common prefix with N3 and N7, but it is closest to N3. It has two common prefixes with node N3. Finally, it has three common prefixes with N3.
- **b.** Node N3 has no common prefix with node N10 and node N12, but it is closest to N10. It has one common prefix with N2 and N7, but it is closest to N2. It has two common prefixes with node N2. Finally, it has three common prefixes with N2.
- **c.** Node N7 has no common prefix with node N10 and node N12, but it is closest to N12. It has one common prefix with N2 and N3, but it is closest to N3. It has two common prefixes with no nodes. It has three common prefixes with no nodes.
- **d.** Node N10 has no common prefix with nodes N2, N3, and N7, but it is closest to N2. It has one common prefix with N12. It has two common prefixes with no nodes. It has three common prefixes with no nodes.
- **e.** Node N12 has no common prefix with nodes N2, N3, and N7, but it is closest to N7. It has one common prefix with N10. It has two common prefixes with no nodes. It has three common prefixes with no nodes.

**P2-41.** The *accept*() function is a blocking procedure; it is blocked until a connection is established between a client and the server. It then gets the information from the packet sent by the client to create a new socket to communicate with the client.