

connection requests, as done in `TCPServer.java`. If multiple clients access this application, they will all send their packets into this single door, `serverSocket`.

```
String sentence = new String(receivePacket.getData());
InetAddress IPAddress = receivePacket.getAddress();
int port = receivePacket.getPort();
```

The above three lines unravel the packet that arrives from the client. The first of the three lines extracts the data from the packet and places the data in the `String sentence`; it has an analogous line in `UDPClient`. The second line extracts the IP address; the third line extracts the client port number, which is chosen by the client and is different from the server port number 9876. (We will discuss client port numbers in some detail in the next chapter.) It is necessary for the server to obtain the address (IP address and port number) of the client, so that it can send the capitalized sentence back to the client.

That completes our analysis of the UDP program pair. To test the application, you install and compile `UDPClient.java` in one host and `UDPServer.java` in another host. (Be sure to include the proper hostname of the server in `UDPClient.java`.) Then execute the two programs on their respective hosts. Unlike with TCP, you can first execute the client side and then the server side. This is because the client process does not attempt to initiate a connection with the server when you execute the client program. Once you have executed the client and server programs, you may use the application by typing a line at the client.

## 2.9 Building a Simple Web Server

Now that we have studied HTTP in some detail and have learned how to write client/server applications in Java, let us combine this new knowledge and build a simple Web server in Java. We will see that the task is remarkably easy.

### 2.9.1 Web Server Functions

Our goal is to build a server that does the following:

- ◆ Handles only one HTTP request
- ◆ Accepts and parses the HTTP request
- ◆ Gets the requested file from the server's file system
- ◆ Creates an HTTP response message consisting of the requested file preceded by header lines
- ◆ Sends the response directly to the client

Let's try to make the code as simple as possible in order to shed some light on the networking issues. The code that we present will be far from bulletproof! For example, let's not worry about handling exceptions. Let's also assume that the client requests an object that is—for sure—in the server's file system.

### WebServer.java

Here is the code for a simple Web server:

```
import java.io.*;
import java.net.*;
import java.util.*;
class WebServer {
    public static void main(String argv[]) throws Exception {
        String requestMessageLine;
        String fileName;
        ServerSocket listenSocket = new ServerSocket(6789);
        Socket connectionSocket = listenSocket.accept();
        BufferedReader inFromClient =
            new BufferedReader(new InputStreamReader(
                connectionSocket.getInputStream()));
        DataOutputStream outToClient =
            new DataOutputStream(
                connectionSocket.getOutputStream());
        requestMessageLine = inFromClient.readLine();
        StringTokenizer tokenizedLine =
            new StringTokenizer(requestMessageLine);
        if (tokenizedLine.nextToken().equals("GET")){
            fileName = tokenizedLine.nextToken();
            if (fileName.startsWith("/") == true )
                fileName = fileName.substring(1);
            File file = new File(fileName);
            int numBytes = (int) file.length();
            FileInputStream inFile = new FileInputStream (
                fileName);
            byte[] fileInBytes = new byte[numBytes];
            inFile.read(fileInBytes);
            outToClient.writeBytes(
                "HTTP/1.0 200 Document Follows\r\n");
            if (fileName.endsWith(".jpg"))
                outToClient.writeBytes("Content-Type:
                    image/jpeg\r\n");
            if (fileName.endsWith(".gif"))
```

```

        outToClient.writeBytes("Content-Type:
                                image/gif\r\n");
        outToClient.writeBytes("Content-Length: " +
                                numOfBytes + "\r\n");
        outToClient.writeBytes("\r\n");
        outToClient.write(fileInBytes, 0, numOfBytes);
        connectionSocket.close();
    }
    else System.out.println("Bad Request Message");
}
}

```

Let us now take a look at the code. The first half of the program is almost identical to `TCPServer.java`. As with `TCPServer.java`, we import the `java.io` and `java.net` packages. In addition to these two packages we also import the `java.util` package, which contains the `StringTokenizer` class, which is used for parsing HTTP request messages. Looking now at the lines within the class `WebServer`, we define two string objects:

```

String requestMessageLine;
String fileName;

```

The object `requestMessageLine` is a string that will contain the first line in the HTTP request message. The object `fileName` is a string that will contain the file name of the requested file. The next set of commands is identical to the corresponding set of commands in `TCPServer.java`.

```

ServerSocket listenSocket = new ServerSocket(6789);
Socket connectionSocket = listenSocket.accept();
BufferedReader inFromClient =
    new BufferedReader(new InputStreamReader
        (connectionSocket.getInputStream()));
DataOutputStream outToClient =
    new DataOutputStream(connectionSocket.
        getOutputStream());

```

Two socket-like objects are created. The first of these objects is `listenSocket`, which is of type `ServerSocket`. The object `listenSocket` is created by the server program before it receives a request for a TCP connection from a client. It listens at port 6789 and waits for a request from some client to establish a TCP connection. When a request for a connection arrives, the `accept()` method of `listenSocket` creates a new object, `connectionSocket`, of type `Socket`.

Next, two streams are created: the `BufferedReader inFromClient` and the `DataOutputStream outToClient`. The HTTP request message comes from the network, through `connectionSocket` and into `inFromClient`; the HTTP response message goes into `outToClient`, through `connectionSocket` and into the network. The remaining portion of the code differs significantly from `TCPServer.java`.

```
requestMessageLine = inFromClient.readLine();
```

The above command reads the first line of the HTTP request message. This line is supposed to be of the form

```
GET file_name HTTP/1.0
```

Our server must now parse the line to extract the file name.

```
StringTokenizer tokenizedLine =  
    new StringTokenizer(requestMessageLine);  
if (tokenizedLine.nextToken().equals("GET")){  
    fileName = tokenizedLine.nextToken();  
    if (fileName.startsWith("/") == true )  
        fileName = fileName.substring(1);
```

The above commands parse the first line of the request message to obtain the requested file name. The object `tokenizedLine` can be thought of as the original request line with each of the “words” `GET`, `file_name`, and `HTTP/1.0` placed in a separate placeholder called a token. The server knows from the HTTP RFC that the file name for the requested file is contained in the token that follows the token containing “GET.” This file name is put in a string called `fileName`. The purpose of the last `if` statement in the above code is to remove the slash that may precede the file name.

```
FileInputStream inFile = new FileInputStream (fileName);
```

The above command attaches a stream, `inFile`, to the file `fileName`.

```
byte[] fileInBytes = new byte[numOfBytes];  
inFile.read(fileInBytes);
```

These commands determine the size of the file and construct an array of bytes of that size. The name of the array is `fileInBytes`. The last command reads from the stream `inFile` to the byte array `fileInBytes`. The program must convert to bytes because the output stream `outToClient` may only be fed with bytes.

Now we are ready to construct the HTTP response message. To this end we must first send the HTTP response header lines into the `DataOutputStream` `outToClient`:

```
outToClient.writeBytes("HTTP/1.0 200 Document
    Follows\r\n");
if (fileName.endsWith(".jpg"))
    outToClient.writeBytes("Content-Type:
        image/jpeg\r\n");
if (fileName.endsWith(".gif"))
    outToClient.writeBytes("Content-Type:
        image/gif\r\n");
outToClient.writeBytes("Content-Length: " + numBytes +
    "\r\n");
outToClient.writeBytes("\r\n")
```

The above set of commands is particularly interesting. These commands prepare the header lines for the HTTP response message and send the header lines to the TCP send buffer. The first command sends the mandatory status line `HTTP/1.0 200 Document Follows`, followed by a carriage return and a line feed. The next two command lines prepare a single content-type header line. If the server is to transfer a GIF image, then the server prepares the header line `Content-Type: image/gif`. If, on the other hand, the server is to transfer a JPEG image, then the server prepares the header line `Content-Type: image/jpeg`. (In this simple Web server, no content line is sent if the object is neither a GIF nor a JPEG image.) The server then prepares and sends a content-length header line and a mandatory blank line to precede the object itself that is to be sent. We now must send the file `FileName` into the `DataOutputStream` `outToClient`.

We can now send the requested file:

```
outToClient.write(fileInBytes, 0, numBytes);
```

The above command sends the requested file, `fileInBytes`, to the TCP send buffer. TCP will concatenate the file, `fileInBytes`, to the header lines just created, segment the concatenation if necessary, and send the TCP segments to the client. After serving one request for one file, the server performs some housekeeping by closing the socket `connectionSocket`:

```
connectionSocket.close();
```

To test this Web server, install it on a host. Also put some files in the host. Then use a browser running on any machine to request a file from the server. When you

request a file, you will need to use the port number that you include in the server code (for example, 6789). So if your server is located at `somehost.somewhere.edu`, the file is `somefile.html`, and the port number is 6789, then the browser should request the following:

```
http://somehost.somewhere.edu:6789/somefile.html
```

## 2.10 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 Internet applications and seen its use in the HTTP, FTP, SMTP, POP3, and DNS protocols. We've studied these important application-level protocols, their associated applications (the Web, file transfer, e-mail, and DNS) in some detail. We've also learned about the increasingly prevalent P2P architecture and seen its use in P2P file sharing. 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, and also built a simple Web server using sockets. The first step in our journey down the layered network architecture is 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, FTP, SMTP, POP3, and DNS protocols, has now added considerable substance to this definition. Protocols are a key concept in networking; our study of applications 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 Sections 2.7 through 2.9. However, we have said little about how TCP and UDP provide these service models. For example, we have said little about how TCP provides a reliable data transfer service to its applications. 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.