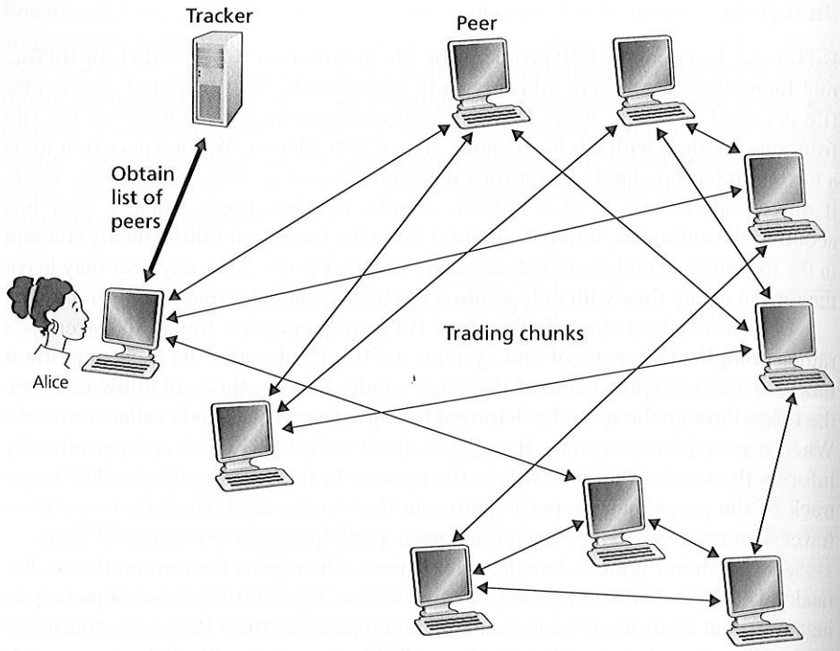

Figure 2.26 ♦ File distribution with BitTorrent

To determine which requests she responds to, BitTorrent uses a clever trading algorithm. The basic idea is that Alice gives priority to the neighbors that are currently supplying her data *at the highest rate*. Specifically, for each of her neighbors, Alice continually measures the rate at which she receives bits and determines the four peers that are feeding her bits at the highest rate. She then reciprocates by sending chunks to these same four peers. Every 10 seconds, she recalculates the rates and possibly modifies the set of four peers. In BitTorrent lingo, these four peers are said to be **unchoked**. Importantly, every 30 seconds, she also picks one additional neighbor at random and sends it chunks. Let's call the randomly chosen peer Bob. In BitTorrent lingo, Bob is said to be **optimistically unchoked**. Because Alice is sending data to Bob, she may become one of Bob's top four uploaders, in which case Bob would start to send data to Alice. If the rate at which Bob sends data to Alice is high enough, Bob could then, in turn, become one of Alice's top four uploaders. In other words, every 30 seconds, Alice will randomly choose a new trading partner and initiate trading with that partner. If the two peers are satisfied with the trading, they will put each other in their top four lists and continue trading with each other until one of the peers finds a better partner. The effect is that peers capable of uploading at compatible rates tend to find each other. The random neighbor selection also allows new peers to get

chunks, so that they can have something to trade. All other neighboring peers besides these five peers (four "top" peers and one probing peer) are "choked," that is, they do not receive any chunks from Alice. BitTorrent has a number of interesting mechanisms that are not discussed here, including pieces (mini-chunks), pipelining, random first selection, endgame mode, and anti-snubbing [Cohen 2003].

The incentive mechanism for trading just described is often referred to as tit-for-tat [Cohen 2003]. It has been shown that this incentive scheme can be circumvented [Liogkas 2006; Locher 2006; Piatek 2007]. Nevertheless, the BitTorrent ecosystem is wildly successful, with millions of simultaneous peers actively sharing files in hundreds of thousands of torrents. If BitTorrent had been designed without tit-for-tat (or a variant), but otherwise exactly the same, BitTorrent would likely not even exist now, as the majority of the users would have been freeriders [Saroiu 2002].

Interesting variants of the BitTorrent protocol are proposed [Guo 2005; Piatek 2007]. Also, many of the P2P live streaming applications, such as PPLive and ppstream, have been inspired by BitTorrent [Hei 2007].

2.6.2 Distributed Hash Tables (DHTs)

In this section, we will consider how to implement a simple database in a P2P network. Let's begin by describing a centralized version of this simple database, which will simply contain (key, value) pairs. For example, the keys could be social security numbers and the values could be the corresponding human names; in this case, an example key-value pair is (156-45-7081, Johnny Wu). Or the keys could be content names (e.g., names of movies, albums, and software), and the value could be the IP address at which the content is stored; in this case, an example key-value pair is (Led Zeppelin IV, 128.17.123.38). We query the database with a key. If there are one or more key-value pairs in the database that match the query key, the database returns the corresponding values. So, for example, if the database stores social security numbers and their corresponding human names, we can query with a specific social security number, and the database returns the name of the human who has that social security number. Or, if the database stores content names and their corresponding IP addresses, we can query with a specific content name, and the database returns the IP addresses that store the specific content.

Building such a database is straightforward with a client-server architecture that stores all the (key, value) pairs in one central server. So in this section, we'll instead consider how to build a distributed, P2P version of this database that will store the (key, value) pairs over millions of peers. In the P2P system, each peer will only hold a small subset of the totality of the (key, value) pairs. We'll allow any peer to query the distributed database with a particular key. The distributed database will then locate the peers that have the corresponding (key, value) pairs and return the key-value pairs to the querying peer. Any peer will also be allowed to insert new key-value pairs into the database. Such a distributed database is referred to as a **distributed hash table (DHT)**.

VideoNote
Walking through distributed hash tables

Before describing how we can create a DHT, let's first describe a specific example DHT service in the context of P2P file sharing. In this case, a key is the content name and the value is the IP address of a peer that has a copy of the content. So, if Bob and Charlie each have a copy of the latest Linux distribution, then the DHT database will include the following two key-value pairs: (Linux, IP_{Bob}) and (Linux, $IP_{Charlie}$). More specifically, since the DHT database is distributed over the peers, some peer, say Dave, will be responsible for the key "Linux" and will have the corresponding key-value pairs. Now suppose Alice wants to obtain a copy of Linux. Clearly, she first needs to know which peers have a copy of Linux before she can begin to download it. To this end, she queries the DHT with "Linux" as the key. The DHT then determines that the peer Dave is responsible for the key "Linux." The DHT then contacts peer Dave, obtains from Dave the key-value pairs (Linux, IP_{Bob}) and (Linux, $IP_{Charlie}$), and passes them on to Alice. Alice can then download the latest Linux distribution from either IP_{Bob} or $IP_{Charlie}$.

Now let's return to the general problem of designing a DHT for general key-value pairs. One naïve approach to building a DHT is to randomly scatter the (key, value) pairs across all the peers and have each peer maintain a list of the IP addresses of all participating peers. In this design, the querying peer sends its query to all other peers, and the peers containing the (key, value) pairs that match the key can respond with their matching pairs. Such an approach is completely unscalable, of course, as it would require each peer to not only know about all other peers (possibly millions of such peers!) but even worse, have each query sent to *all* peers.

We now describe an elegant approach to designing a DHT. To this end, let's first assign an identifier to each peer, where each identifier is an integer in the range $[0, 2^n - 1]$ for some fixed n. Note that each such identifier can be expressed by an n-bit representation. Let's also require each key to be an integer in the same range. The astute reader may have observed that the example keys described a little earlier (social security numbers and content names) are not integers. To create integers out of such keys, we will use a hash function that maps each key (e.g., social security number) to an integer in the range $[0, 2^n - 1]$. A hash function is a many-to-one function for which two different inputs can have the same output (same integer), but the likelihood of the having the same output is extremely small. (Readers who are unfamiliar with hash functions may want to visit Chapter 7, in which hash functions are discussed in some detail.) The hash function is assumed to be available to all peers in the system. Henceforth, when we refer to the "key," we are referring to the hash of the original key. So, for example, if the original key is "Led Zeppelin IV," the key used in the DHT will be the integer that equals the hash of "Led Zeppelin IV." As you may have guessed, this is why "Hash" is used in the term "Distributed Hash Function."

Let's now consider the problem of storing the (key, value) pairs in the DHT. The central issue here is defining a rule for assigning keys to peers. Given that each peer has an integer identifier and that each key is also an integer in the same range, a natural approach is to assign each (key, value) pair to the peer whose identifier is the *closest* to the key. To implement such a scheme, we'll need to define what is meant by "closest," for which many conventions are possible. For convenience, let's define the

closest peer as the *closest successor of the key*. To gain some insight here, let's take a look at a specific example. Suppose $n = 4$ so that all the peer and key identifiers are in the range [0, 15]. Further suppose that there are eight peers in the system with identifiers 1, 3, 4, 5, 8, 10, 12, and 15. Finally, suppose we want to store the (key, value) pair (11, Johnny Wu) in one of the eight peers. But in which peer? Using our closest convention, since peer 12 is the closest successor for key 11, we therefore store the pair (11, Johnny Wu) in the peer 12. [To complete our definition of closest, if the key is exactly equal to one of the peer identifiers, we store the (key, value) pair in that matching peer; and if the key is larger than all the peer identifiers, we use a modulo-2^n convention, storing the (key, value) pair in the peer with the smallest identifier.]

Now suppose a peer, Alice, wants to insert a (key, value) pair into the DHT. Conceptually, this is straightforward: She first determines the peer whose identifier is closest to the key; she then sends a message to that peer, instructing it to store the (key, value) pair. But how does Alice determine the peer that is closest to the key? If Alice were to keep track of all the peers in the system (peer IDs and corresponding IP addresses), she could locally determine the closest peer. But such an approach requires e*ach* peer to keep track of *all* other peers in the DHT—which is completely impractical for a large-scale system with millions of peers.

Circular DHT

To address this problem of scale, let's now consider organizing the peers into a circle. In this circular arrangement, each peer only keeps track of its immediate successor and immediate predecessor (modulo 2^n). An example of such a circle is shown in Figure 2.27(a). In this example, n is again 4 and there are the same eight

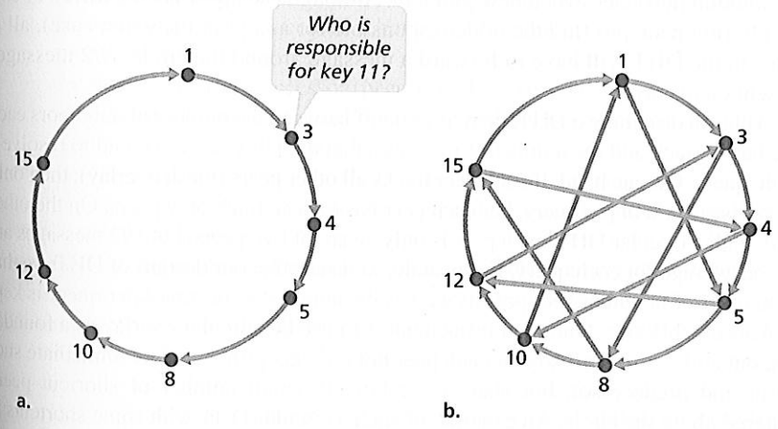

Figure 2.27 ♦ (a) A circular DHT. Peer 3 wants to determine who is responsible for key 11. (b) A circular DHT with shortcuts

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

Using the circular overlay in Figure 2.27(a), now suppose that peer 3 wants to determine which peer in the DHT is responsible for key 11. Using the circular overlay, the origin peer (peer 3) creates a message saying "Who is responsible for key 11?" and sends this message clockwise around the circle. Whenever a peer receives such a message, because it knows the identifier of its successor and predecessor, it can determine whether it is responsible for (that is, closest to) the key in question. If a peer is not responsible for the key, it simply sends the message to its successor. So, for example, when peer 4 receives the message asking about key 11, it determines that it is not responsible for the key (because its successor is closer to the key), so it just passes the message along to peer 5. This process continues until the message arrives at peer 12, who determines that it is the closest peer to key 11. At this point, peer 12 can send a message back to the querying peer, peer 3, indicating that it is responsible for key 11.

The circular DHT provides a very elegant solution for reducing the amount of overlay information each peer must manage. In particular, each peer needs only to be aware of two peers, its immediate successor and its immediate predecessor. But this solution introduces yet a new problem. Although each peer is only aware of two neighboring peers, to find the node responsible for a key (in the worst case), all N nodes in the DHT will have to forward a message around the circle; $N/2$ messages are sent on average.

Thus, in designing a DHT, there is tradeoff between the number of neighbors each peer has to track and the number of messages that the DHT needs to send to resolve a single query. On one hand, if each peer tracks all other peers (mesh overlay), then only one message is sent per query, but each peer has to keep track of N peers. On the other hand, with a circular DHT, each peer is only aware of two peers, but $N/2$ messages are sent on average for each query. Fortunately, we can refine our designs of DHTs so that the number of neighbors per peer as well as the number of messages per query is kept to an acceptable size. One such refinement is to use the circular overlay as a foundation, but add "shortcuts" so that each peer not only keeps track of its immediate successor and predecessor, but also of a relatively small number of shortcut peers scattered about the circle. An example of such a circular DHT with some shortcuts is shown in Figure 2.27(b). Shortcuts are used to expedite the routing of query messages. Specifically, when a peer receives a message that is querying for a key, it forwards the message to the neighbor (successor neighbor or one of the shortcut neighbors) which is the closet to the key. Thus, in Figure 2.27(b), when peer 4 receives the message asking about key 11, it determines that the closet peer to the key (among its neighbors) is its shortcut neighbor 10 and then forwards the message directly to peer 10. Clearly, shortcuts can significantly reduce the number of messages used to process a query.

The next natural question is "How many shortcut neighbors should a peer have, and which peers should be these shortcut neighbors? This question has received significant attention in the research community [Balakrishnan 2003; Androutsellis-Theotokis 2004]. Importantly, it has been shown that the DHT can be designed so that both the number of neighbors per peer as well as the number of messages per query is $O(\log N)$, where N is the number of peers. Such designs strike a satisfactory compromise between the extreme solutions of using mesh and circular overlay topologies.

Peer Churn

In P2P systems, a peer can come or go without warning. Thus, when designing a DHT, we also must be concerned about maintaining the DHT overlay in the presence of such peer churn. To get a big-picture understanding of how this could be accomplished, let's once again consider the circular DHT in Figure 2.27(a). To handle peer churn, we will now require each peer to track (that is, know the IP address of) its first and second successors; for example, peer 4 now tracks both peer 5 and peer 8. We also require each peer to periodically verify that its two successors are alive (for example, by periodically sending ping messages to them and asking for responses). Let's now consider how the DHT is maintained when a peer abruptly leaves. For example, suppose peer 5 in Figure 2.27(a) abruptly leaves. In this case, the two peers preceding the departed peer (4 and 3) learn that 5 has departed, since it no longer responds to ping messages. Peers 4 and 3 thus need to update their successor state information. Let's consider how peer 4 updates its state:

1. Peer 4 replaces its first successor (peer 5) with its second successor (peer 8).
2. Peer 4 then asks its new first successor (peer 8) for the identifier and IP address of its immediate successor (peer 10). Peer 4 then makes peer 10 its second successor.

In the homework problems, you will be asked to determine how peer 3 updates its overlay routing information.

Having briefly addressed what has to be done when a peer leaves, let's now consider what happens when a peer wants to join the DHT. Let's say a peer with identifier 13 wants to join the DHT, and at the time of joining, it only knows about peer 1's existence in the DHT. Peer 13 would first send peer 1 a message, saying "what will be 13's predecessor and successor?" This message gets forwarded through the DHT until it reaches peer 12, who realizes that it will be 13's predecessor and that its current successor, peer 15, will become 13's successor. Next, peer 12 sends this predecessor and successor information to peer 13. Peer 13 can now join

the DHT by making peer 15 its successor and by notifying peer 12 that it should change its immediate successor to 13.

DHTs have been finding widespread use in practice. For example, BitTorrent uses the Kademlia DHT to create a distributed tracker. In the BitTorrent, the key is the torrent identifier and the value is the IP addresses of all the peers currently participating in the torrent [Falkner 2007, Neglia 2007]. In this manner, by querying the DHT with a torrent identifier, a newly arriving BitTorrent peer can determine the peer that is responsible for the identifier (that is, for tracking the peers in the torrent). After having found that peer, the arriving peer can query it for a list of other peers in the torrent.

2.7 Socket Programming: Creating Network Applications

Now that we've looked at a number of important network applications, let's explore how network application programs are actually created. Recall from Section 2.1 that a typical network application consists of a pair of programs—a client program and a server program—residing in two different end systems. When these two programs are executed, a client process and a server process are created, and these processes communicate with each other by reading from, and writing to, sockets. When creating a network application, the developer's main task is therefore to write the code for both the client and server programs.

There are two types of network applications. One type is an implementation whose operation is specified in a protocol standard, such as an RFC or some other standards document; such an application is sometimes referred to as "open," since the rules specifying its operation are known to all. For such an implementation, the client and server programs must conform to the rules dictated by the RFC. For example, the client program could be an implementation of the client side of the FTP protocol, described in Section 2.3 and explicitly defined in RFC 959; similarly, the server program could be an implementation of the FTP server protocol, also explicitly defined in RFC 959. If one developer writes code for the client program and another developer writes code for the server program, and both developers carefully follow the rules of the RFC, then the two programs will be able to interoperate. Indeed, many of today's network applications involve communication between client and server programs that have been created by independent developers—for example, a Firefox browser communicating with an Apache Web server, or a BitTorrent client communicating with BitTorrent tracker.

The other type of network application is a proprietary network application. In this case the client and server programs employ an application-layer protocol that has *not* been openly published in an RFC or elsewhere. A single developer (or development team) creates both the client and server programs, and the developer has complete control over what goes in the code. But because the code does not implement an open protocol, other independent developers will not be able to develop code that interoperates with the application.

In this section, we'll examine the key issues in developing a client-server application, and we'll "get our hands dirty" by looking at code that implements a very simple client-server application. During the development phase, one of the first decisions the developer must make is whether the application is to run over TCP or over UDP. Recall that TCP is connection oriented and provides a reliable byte-stream channel through which data flows between two end systems. UDP is connectionless and sends independent packets of data from one end system to the other, without any guarantees about delivery. Recall also that when a client or server program implements a protocol defined by an RFC, it should use the well-known port number associated with the protocol; conversely, when developing a proprietary application, the developer must be careful to avoid using such well-known port numbers. (Port numbers were briefly discussed in Section 2.1. They are covered in more detail in Chapter 3.)

We introduce UDP and TCP socket programming by way of a simple UDP application and a simple TCP application. We present the simple UDP and TCP applications in Python. We could have written the code in Java, C, or C++, but we chose Python mostly because Python clearly exposes the key socket concepts. With Python there are fewer lines of code, and each line can be explained to the novice programmer without difficulty. But there's no need to be frightened if you are not familiar with Python. You should be able to easily follow the code if you have experience programming in Java, C, or C++.

If you are interested in client-server programming with Java, you are encouraged to see the companion Web site for this textbook; in fact, you can find there all the examples in this section (and associated labs) in Java. For readers who are interested in client-server programming in C, there are several good references available [Donahoo 2001; Stevens 1997; Frost 1994; Kurose 1996]; our Python examples below have a similar look and feel to C.

2.7.1 Socket Programming with UDP

In this subsection, we'll write simple client-server programs that use UDP; in the following section, we'll write similar programs that use TCP.

Recall from Section 2.1 that processes running on different machines communicate with each other by sending messages into sockets. We said that each process is analogous to a house and the process's socket is analogous to a door. The application resides on one side of the door in the house; the transport-layer protocol resides on the other side of the door in the outside world. The application developer has control of everything on the application-layer side of the socket; however, it has little control of the transport-layer side.