application-specific protocol. In this model, the front end (the client software) does no processing other than necessary for presenting the application's interface.

Continuing along this line of reasoning, we may also move part of the application to the front end, as shown in Fig. 2-5(c). An example where this makes sense is where the application makes use of a form that needs to be filled in entirely before it can be processed. The front end can then check the correctness and consistency of the form, and where necessary interact with the user. Another example of the organization of Fig. 2-5(c), is that of a word processor in which the basic editing functions execute on the client side where they operate on locally cached, or in-memory data, but where the advanced support tools such as checking the spelling and grammar execute on the server side.

In many client-server environments, the organizations shown in Fig. 2-5(d) and Fig. 2-5(e) are particularly popular. These organizations are used where the client machine is a PC or workstation, connected through a network to a distributed file system or database. Essentially, most of the application is running on the client machine, but all operations on files or database entries go to the server. For example, many banking applications run on an end-user's machine where the user prepares transactions and such. Once finished, the application contacts the database on the bank's server and uploads the transactions for further processing. Fig. 2-5(e) represents the situation where the client's local disk contains part of the data. For example, when browsing the Web, a client can gradually build a huge cache on local disk of most recent inspected Web pages.

We note that for a few years there has been a strong trend to move away from the configurations shown in Fig. 2-5(d) and Fig. 2-5(e) in those case that client software is placed at end-user machines. In these cases, most of the processing and data storage is handled at the server side. The reason for this is simple: although client machines do a lot, they are also more problematic to manage. Having more functionality on the client machine makes client-side software more prone to errors and more dependent on the client's underlying platform (i.e., operating system and resources). From a system's management perspective, having what are called **fat clients** is not optimal. Instead the **thin clients** as represented by the organizations shown in Fig. 2-5(a)–(c) are much easier, perhaps at the cost of less sophisticated user interfaces and client-perceived performance.

Note that this trend does not imply that we no longer need distributed systems. On the contrary, what we are seeing is that server-side solutions are becoming increasingly more distributed as a single server is being replaced by multiple servers running on different machines. In particular, when distinguishing only client and server machines as we have done so far, we miss the point that a server may sometimes need to act as a client, as shown in Fig. 2-6, leading to a **(physically) three-tiered architecture**.

In this architecture, programs that form part of the processing level reside on a separate server, but may additionally be partly distributed across the client and server machines. A typical example of where a three-tiered architecture is used is

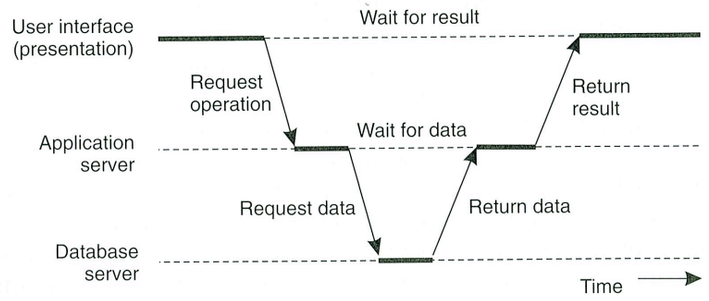

Figure 2-6. An example of a server acting as client.

in transaction processing. As we discussed in Chap. 1, a separate process, called the transaction processing monitor, coordinates all transactions across possibly different data servers.

Another, but very different example where we often see a three-tiered architecture is in the organization of Web sites. In this case, a Web server acts as an entry point to a site, passing requests to an application server where the actual processing takes place. This application server, in turn, interacts with a database server. For example, an application server may be responsible for running the code to inspect the available inventory of some goods as offered by an electronic bookstore. To do so, it may need to interact with a database containing the raw inventory data. We will come back to Web site organization in Chap. 12.

2.2.2 Decentralized Architectures

Multitiered client-server architectures are a direct consequence of dividing applications into a user-interface, processing components, and a data level. The different tiers correspond directly with the logical organization of applications. In many business environments, distributed processing is equivalent to organizing a client-server application as a multitiered architecture. We refer to this type of distribution as **vertical distribution**. The characteristic feature of vertical distribution is that it is achieved by placing *logically* different components on different machines. The term is related to the concept of **vertical fragmentation** as used in distributed relational databases, where it means that tables are split column-wise, and subsequently distributed across multiple machines (Oszu and Valduriez, 1999).

Again, from a system management perspective, having a vertical distribution can help: functions are logically and physically split across multiple machines, where each machine is tailored to a specific group of functions. However, vertical distribution is only one way of organizing client-server applications. In modern architectures, it is often the distribution of the clients and the servers that counts,

which we refer to as **horizontal distribution**. In this type of distribution, a client or server may be physically split up into logically equivalent parts, but each part is operating on its own share of the complete data set, thus balancing the load. In this section we will take a look at a class of modern system architectures that support horizontal distribution, known as **peer-to-peer systems**.

From a high-level perspective, the processes that constitute a peer-to-peer system are all equal. This means that the functions that need to be carried out are represented by every process that constitutes the distributed system. As a consequence, much of the interaction between processes is symmetric: each process will act as a client and a server at the same time (which is also referred to as acting as a **servent**).

Given this symmetric behavior, peer-to-peer architectures evolve around the question how to organize the processes in an **overlay network**, that is, a network in which the nodes are formed by the processes and the links represent the possible communication channels (which are usually realized as TCP connections). In general, a process cannot communicate directly with an arbitrary other process, but is required to send messages through the available communication channels. Two types of overlay networks exist: those that are structured and those that are not. These two types are surveyed extensively in Lua et al. (2005) along with numerous examples. Aberer et al. (2005) provide a reference architecture that allows for a more formal comparison of the different types of peer-to-peer systems. A survey taken from the perspective of content distribution is provided by Androutsellis-Theotokis and Spinellis (2004).

Structured Peer-to-Peer Architectures

In a structured peer-to-peer architecture, the overlay network is constructed using a deterministic procedure. By far the most-used procedure is to organize the processes through a **distributed hash table** (**DHT**). In a DHT-based system, data items are assigned a random key from a large identifier space, such as a 128-bit or 160-bit identifier. Likewise, nodes in the system are also assigned a random number from the same identifier space. The crux of every DHT-based system is then to implement an efficient and deterministic scheme that uniquely maps the key of a data item to the identifier of a node based on some distance metric (Balakrishnan, 2003). Most importantly, when looking up a data item, the network address of the node responsible for that data item is returned. Effectively, this is accomplished by *routing* a request for a data item to the responsible node.

For example, in the Chord system (Stoica et al., 2003) the nodes are logically organized in a ring such that a data item with key k is mapped to the node with the smallest identifier $id \geq k$. This node is referred to as the *successor* of key k and denoted as *succ(k)*, as shown in Fig. 2-7. To actually look up the data item, an application running on an arbitrary node would then call the function LOOKUP(k)

which would subsequently return the network address of *succ(k)*. At that point, the application can contact the node to obtain a copy of the data item.

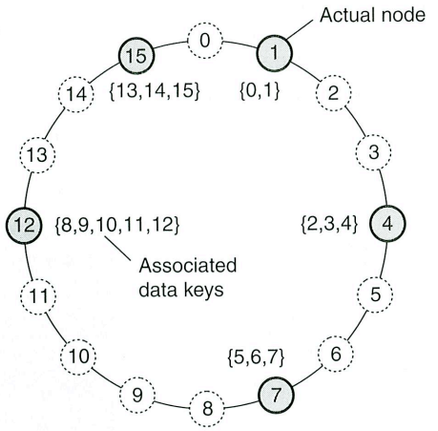

Figure 2-7. The mapping of data items onto nodes in Chord.

We will not go into algorithms for looking up a key now, but defer that discussion until Chap. 5 where we describe details of various naming systems. Instead, let us concentrate on how nodes organize themselves into an overlay network, or, in other words, **membership management**. In the following, it is important to realize that looking up a key does not follow the logical organization of nodes in the ring from Fig. 2-7. Rather, each node will maintain shortcuts to other nodes in such a way that lookups can generally be done in $O(log\,(N))$ number of steps, where N is the number of nodes participating in the overlay.

Now consider Chord again. When a node wants to join the system, it starts with generating a random identifier *id*. Note that if the identifier space is large enough, then provided the random number generator is of good quality, the probability of generating an identifier that is already assigned to an actual node is close to zero. Then, the node can simply do a lookup on *id*, which will return the network address of *succ(id)*. At that point, the joining node can simply contact *succ(id)* and its predecessor and insert itself in the ring. Of course, this scheme requires that each node also stores information on its predecessor. Insertion also yields that each data item whose key is now associated with node *id*, is transferred from *succ(id)*.

Leaving is just as simple: node *id* informs its departure to its predecessor and successor, and transfers its data items to *succ(id)*.

Similar approaches are followed in other DHT-based systems. As an example, consider the **Content Addressable Network** (**CAN**), described in Ratnasamy et al. (2001). CAN deploys a d-dimensional Cartesian coordinate space, which is completely partitioned among all all the nodes that participate in the system. For

purpose of illustration, let us consider only the 2-dimensional case, of which an example is shown in Fig. 2-8.

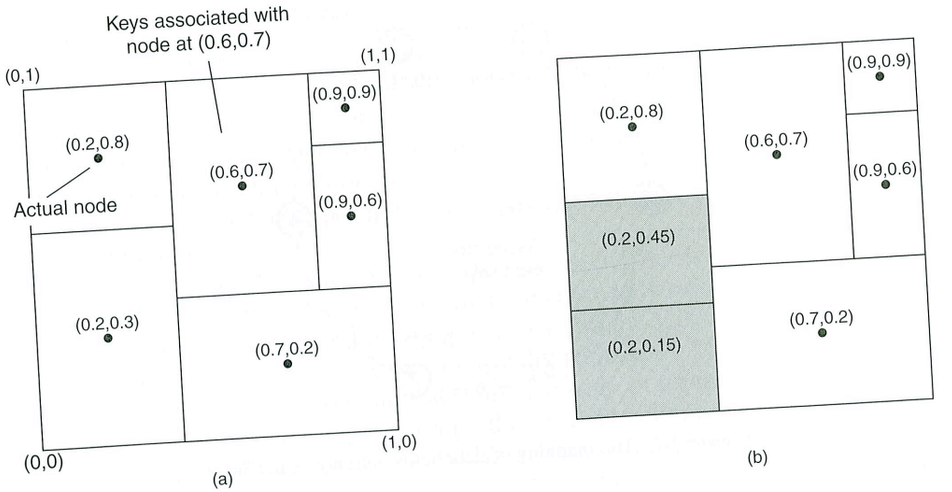

Figure 2-8. (a) The mapping of data items onto nodes in CAN. (b) Splitting a region when a node joins.

Fig. 2-8(a) shows how the two-dimensional space $[0,1]\times[0,1]$ is divided among six nodes. Each node has an associated region. Every data item in CAN will be assigned a unique point in this space, after which it is also clear which node is responsible for that data (ignoring data items that fall on the border of multiple regions, for which a deterministic assignment rule is used).

When a node P wants to join a CAN system, it picks an arbitrary point from the coordinate space and subsequently looks up the node Q in whose region that point falls. This lookup is accomplished through positioned-based routing, of which the details are deferred until later chapters. Node Q then splits its region into two halves, as shown in Fig. 2-8(b), and one half is assigned to the node P. Nodes keep track of their neighbors, that is, nodes responsible for adjacent region. When splitting a region, the joining node P can easily come to know who its new neighbors are by asking node P. As in Chord, the data items for which node P is now responsible are transferred from node Q.

Leaving is a bit more problematic in CAN. Assume that in Fig. 2-8, the node with coordinate (0.6,0.7) leaves. Its region will be assigned to one of its neighbors, say the node at (0.9,0.9), but it is clear that simply merging it and obtaining a rectangle cannot be done. In this case, the node at (0.9,0.9) will simply take care of that region and inform the old neighbors of this fact. Obviously, this may lead to less symmetric partitioning of the coordinate space, for which reason a background process is periodically started to repartition the entire space.

Unstructured Peer-to-Peer Architectures

Unstructured peer-to-peer systems largely rely on randomized algorithms for constructing an overlay network. The main idea is that each node maintains a list of neighbors, but that this list is constructed in a more or less random way. Likewise, data items are assumed to be randomly placed on nodes. As a consequence, when a node needs to locate a specific data item, the only thing it can effectively do is flood the network with a search query (Risson and Moors, 2006). We will return to searching in unstructured overlay networks in Chap. 5, and for now concentrate on membership management.

One of the goals of many unstructured peer-to-peer systems is to construct an overlay network that resembles a **random graph**. The basic model is that each node maintains a list of c neighbors, where, ideally, each of these neighbors represents a randomly chosen *live* node from the current set of nodes. The list of neighbors is also referred to as a **partial view**. There are many ways to construct such a partial view. Jelasity et al. (2004, 2005a) have developed a framework that captures many different algorithms for overlay construction to allow for evaluations and comparison. In this framework, it is assumed that nodes regularly exchange entries from their partial view. Each entry identifies another node in the network, and has an associated age that indicates how old the reference to that node is. Two threads are used, as shown in Fig. 2-9.

The active thread takes the initiative to communicate with another node. It selects that node from its current partial view. Assuming that entries need to be *pushed* to the selected peer, it continues by constructing a buffer containing $c/2+1$ entries, including an entry identifying itself. The other entries are taken from the current partial view.

If the node is also in *pull mode* it will wait for a response from the selected peer. That peer, in the meantime, will also have constructed a buffer by means the passive thread shown in Fig. 2-9(b), whose activities strongly resemble that of the active thread.

The crucial point is the construction of a new partial view. This view, for initiating as well as for the contacted peer, will contain exactly c entries, part of which will come from received buffer. In essence, there are two ways to construct the new view. First, the two nodes may decide to discard the entries that they had sent to each other. Effectively, this means that they will *swap* part of their original views. The second approach is to discard as many *old* entries as possible. In general, it turns out that the two approaches are complementary [see Jelasity et al. (2005a) for the details]. It turns out that many membership management protocols for unstructured overlays fit this framework. There are a number of interesting observations to make.

First, let us assume that when a node wants to join it contacts an arbitrary other node, possibly from a list of well-known access points. This access point is just a regular member of the overlay, except that we can assume it to be highly

Actions by active thread (periodically repeated):

```
select a peer P from the current partial view;
if PUSH_MODE {
    mybuffer = [(MyAddress, 0)];
    permute partial view;
    move H oldest entries to the end;
    append first c/2 entries to mybuffer;
    send mybuffer to P;
} else {
    send trigger to P;
}
if PULL_MODE {
    receive P's buffer;
}
construct a new partial view from the current one and P's buffer;
increment the age of every entry in the new partial view;
```

(a)

Actions by passive thread:

```
receive buffer from any process Q;
if PULL_MODE {
    mybuffer = [(MyAddress, 0)];
    permute partial view;
    move H oldest entries to the end;
    append first c/2 entries to mybuffer;
    send mybuffer to P;
}
construct a new partial view from the current one and P's buffer;
increment the age of every entry in the new partial view;
```

(b)

Figure 2-9. (a) The steps taken by the active thread. (b) The steps take by the passive thread.

available. In this case, it turns out that protocols that use only *push mode* or only *pull mode* can fairly easily lead to disconnected overlays. In other words, groups of nodes will become isolated and will never be able to reach every other node in the network. Clearly, this is an undesirable feature, for which reason it makes more sense to let nodes actually *exchange* entries.

Second, leaving the network turns out to be a very simple operation provided the nodes exchange partial views on a regular basis. In this case, a node can simply depart without informing any other node. What will happen is that when a node P selects one of its apparent neighbors, say node Q, and discovers that Q no longer responds, it simply removes the entry from its partial view to select another peer. It turns out that when constructing a new partial view, a node follows the policy to discard as many old entries as possible, departed nodes will rapidly be forgotten. In other words, entries referring to departed nodes will automatically be quickly removed from partial views.

However, there is a price to pay when this strategy is followed. To explain, consider for a node P the set of nodes that have an entry in their partial view that refers to P. Technically, this is known as the **indegree** of a node. The higher node P's indegree is, the higher the probability that some other node will decide to contact P. In other words, there is a danger that P will become a popular node, which could easily bring it into an imbalanced position regarding workload. Systematically discarding old entries turns out to promote nodes to ones having a high indegree. There are other trade-offs in addition, for which we refer to Jelasity et al. (2005a).

Topology Management of Overlay Networks

Although it would seem that structured and unstructured peer-to-peer systems form strict independent classes, this need actually not be case [see also Castro et al. (2005)]. One key observation is that by carefully exchanging and selecting entries from partial views, it is possible to construct and maintain specific topologies of overlay networks. This topology management is achieved by adopting a two-layered approach, as shown in Fig. 2-10.

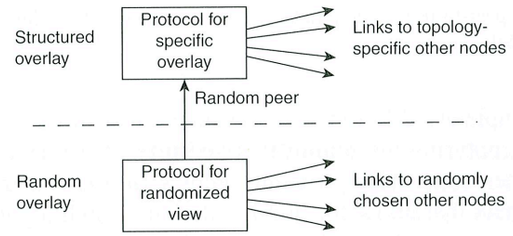

Figure 2-10. A two-layered approach for constructing and maintaining specific overlay topologies using techniques from unstructured peer-to-peer systems.

The lowest layer constitutes an unstructured peer-to-peer system in which nodes periodically exchange entries of their partial views with the aim to maintain an accurate random graph. Accuracy in this case refers to the fact that the partial view should be filled with entries referring to randomly selected *live* nodes.

The lowest layer passes its partial view to the higher layer, where an additional selection of entries takes place. This then leads to a second list of neighbors corresponding to the desired topology. Jelasity and Babaoglu (2005) propose to use a *ranking function* by which nodes are ordered according to some criterion relative to a given node. A simple ranking function is to order a set of nodes by increasing distance from a given node P. In that case, node P will gradually build

up a list of its nearest neighbors, provided the lowest layer continues to pass randomly selected nodes.

As an illustration, consider a logical grid of size $N \times N$ with a node placed on each point of the grid. Every node is required to maintain a list of c nearest neighbors, where the distance between a node at (a_1, a_2) and (b_1, b_2) is defined as d_1+d_2, with $d_i=min\,(N-|a_i-b_i|, |a_i-b_i|)$. If the lowest layer periodically executes the protocol as outlined in Fig. 2-9, the topology that will evolve is a torus, shown in Fig. 2-11.

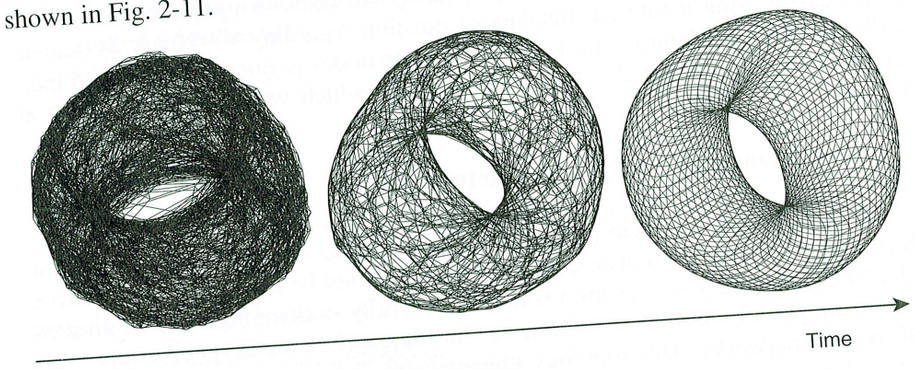

Figure 2-11. Generating a specific overlay network using a two-layered unstructured peer-to-peer system [adapted with permission from Jelasity and Babaoglu (2005)].

Of course, completely different ranking functions can be used. Notably those that are related to capturing the **semantic proximity** of the data items as stored at a peer node are interesting. This proximity allows for the construction of **semantic overlay networks** that allow for highly efficient search algorithms in unstructured peer-to-peer systems. We will return to these systems in Chap. 5 when we discuss attribute-based naming.

Superpeers

Notably in unstructured peer-to-peer systems, locating relevant data items can become problematic as the network grows. The reason for this scalability problem is simple: as there is no deterministic way of routing a lookup request to a specific data item, essentially the only technique a node can resort to is flooding the request. There are various ways in which flooding can be dammed, as we will discuss in Chap. 5, but as an alternative many peer-to-peer systems have proposed to make use of special nodes that maintain an index of data items.

There are other situations in which abandoning the symmetric nature of peer-to-peer systems is sensible. Consider a collaboration of nodes that offer resources

to each other. For example, in a collaborative **content delivery network** (**CDN**), nodes may offer storage for hosting copies of Web pages allowing Web clients to access pages nearby, and thus to access them quickly. In this case a node P may need to seek for resources in a specific part of the network. In that case, making use of a broker that collects resource usage for a number of nodes that are in each other's proximity will allow to quickly select a node with sufficient resources.

Nodes such as those maintaining an index or acting as a broker are generally referred to as **superpeers**. As their name suggests, superpeers are often also organized in a peer-to-peer network, leading to a hierarchical organization as explained in Yang and Garcia-Molina (2003). A simple example of such an organization is shown in Fig. 2-12. In this organization, every regular peer is connected as a client to a superpeer. All communication from and to a regular peer proceeds through that peer's associated superpeer.

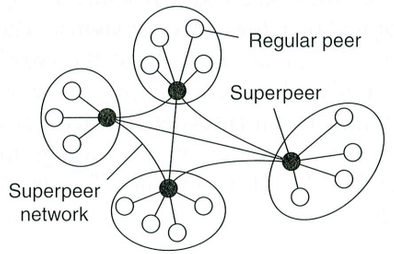

Figure 2-12. A hierarchical organization of nodes into a superpeer network.

In many cases, the client-superpeer relation is fixed: whenever a regular peer joins the network, it attaches to one of the superpeers and remains attached until it leaves the network. Obviously, it is expected that superpeers are long-lived processes with a high availability. To compensate for potential unstable behavior of a superpeer, backup schemes can be deployed, such as pairing every superpeer with another one and requiring clients to attach to both.

Having a fixed association with a superpeer may not always be the best solution. For example, in the case of file-sharing networks, it may be better for a client to attach to a superpeer that maintains an index of files that the client is generally interested in. In that case, chances are bigger that when a client is looking for a specific file, its superpeer will know where to find it. Garbacki et al. (2005) describe a relatively simple scheme in which the client-superpeer relation can change as clients discover better superpeers to associate with. In particular, a superpeer returning the result of a lookup operation is given preference over other superpeers.

As we have seen, peer-to-peer networks offer a flexible means for nodes to join and leave the network. However, with superpeer networks a new problem is introduced, namely how to select the nodes that are eligible to become superpeer.

This problem is closely related to the **leader-election problem**, which we discuss in Chap. 6, when we return to electing superpeers in a peer-to-peer network.

2.2.3 Hybrid Architectures

So far, we have focused on client-server architectures and a number of peer-to-peer architectures. Many distributed systems combine architectural features, as we already came across in superpeer networks. In this section we take a look at some specific classes of distributed systems in which client-server solutions are combined with decentralized architectures.

Edge-Server Systems

An important class of distributed systems that is organized according to a hybrid architecture is formed by **edge-server systems**. These systems are deployed on the Internet where servers are placed "at the edge" of the network. This edge is formed by the boundary between enterprise networks and the actual Internet, for example, as provided by an **Internet Service Provider** (**ISP**). Likewise, where end users at home connect to the Internet through their ISP, the ISP can be considered as residing at the edge of the Internet. This leads to a general organization as shown in Fig. 2-13.

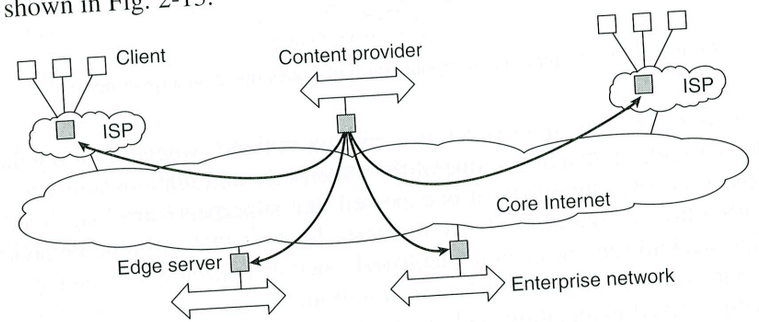

Figure 2-13. Viewing the Internet as consisting of a collection of edge servers.

End users, or clients in general, connect to the Internet by means of an edge server. The edge server's main purpose is to serve content, possibly after applying filtering and transcoding functions. More interesting is the fact that a collection of edge servers can be used to optimize content and application distribution. The basic model is that for a specific organization, one edge server acts as an origin server from which all content originates. That server can use other edge servers for replicating Web pages and such (Leff et al., 2004; Nayate et al., 2004; and Rabinovich and Spatscheck, 2002). We will return to edge-server systems in Chap. 12 when we discuss Web-based solutions.

Collaborative Distributed Systems

Hybrid structures are notably deployed in collaborative distributed systems. The main issue in many of these systems to first get started, for which often a traditional client-server scheme is deployed. Once a node has joined the system, it can use a fully decentralized scheme for collaboration.

To make matters concrete, let us first consider the BitTorrent file-sharing system (Cohen, 2003). BitTorrent is a peer-to-peer file downloading system. Its principal working is shown in Fig. 2-14 The basic idea is that when an end user is looking for a file, he downloads chunks of the file from other users until the downloaded chunks can be assembled together yielding the complete file. An important design goal was to ensure collaboration. In most file-sharing systems, a significant fraction of participants merely download files but otherwise contribute close to nothing (Adar and Huberman, 2000; Saroiu et al., 2003; and Yang et al., 2005). To this end, a file can be downloaded only when the downloading client is providing content to someone else. We will return to this "tit-for-tat" behavior shortly.

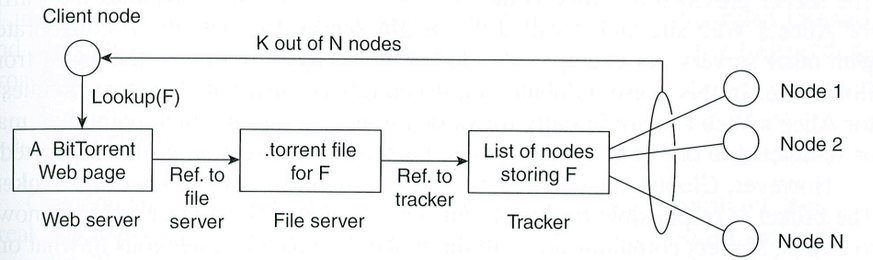

Figure 2-14. The principal working of BitTorrent [adapted with permission from Pouwelse et al. (2004)].

To download a file, a user needs to access a global directory, which is just one of a few well-known Web sites. Such a directory contains references to what are called *.torrent* files. A *.torrent* file contains the information that is needed to download a specific file. In particular, it refers to what is known as a **tracker**, which is a server that is keeping an accurate account of *active* nodes that have (chunks) of the requested file. An active node is one that is currently downloading another file. Obviously, there will be many different trackers, although there will generally be only a single tracker per file (or collection of files).

Once the nodes have been identified from where chunks can be downloaded, the downloading node effectively becomes active. At that point, it will be forced to help others, for example by providing chunks of the file it is downloading that others do not yet have. This enforcement comes from a very simple rule: if node P notices that node Q is downloading more than it is uploading, P can decide to