Chapter 11. Distributed File Systems

Considering that sharing data is fundamental to distributed systems, it is not surprising that distributed file systems form the basis for many distributed applications. Distributed file systems allow multiple processes to share data over long periods of time in a secure and reliable way. As such, they have been used as the basic layer for distributed systems and applications. In this chapter, we consider distributed file systems as a paradigm for general-purpose distributed systems.

11.1. Architecture

We start our discussion on distributed file systems by looking at how they are generally organized. Most systems are built following a traditional client-server architecture, but fully decentralized solutions exist as well. In the following, we will take a look at both kinds of organizations.

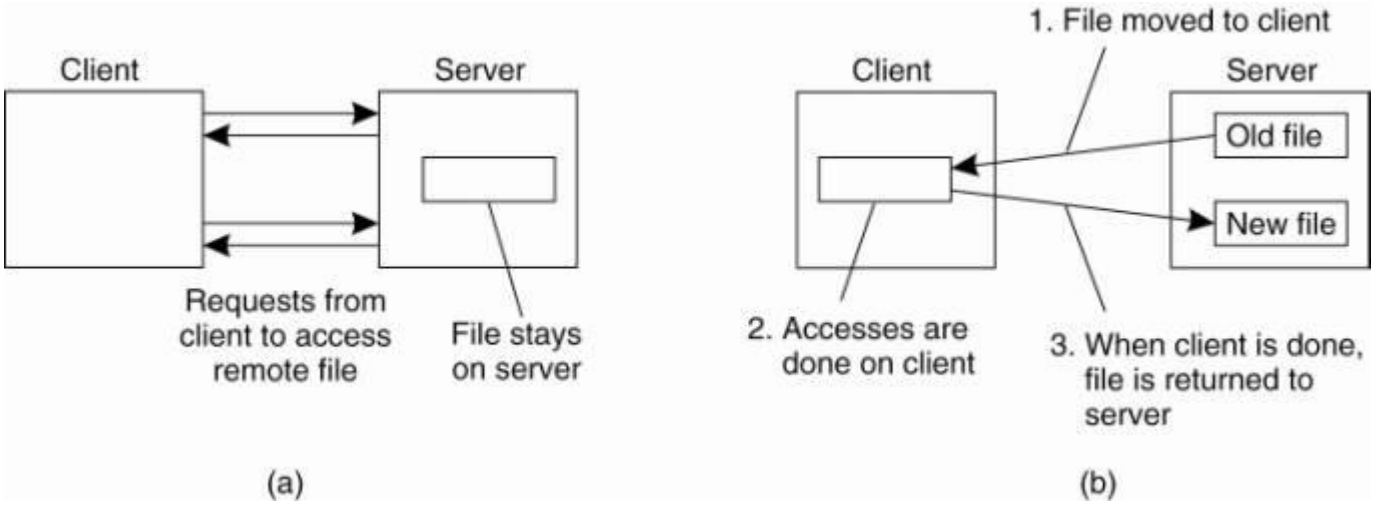
11.1.1. Client-Server Architectures

Many distributed files systems are organized along the lines of client-server architectures, with Sun Microsystem's Network File System (NFS) being one of the most widely-deployed ones for UNIX-based systems. We will take NFS as a canonical example for server-based distributed file systems throughout this chapter. In particular, we concentrate on NFSv3, the widely-used third version of NFS (Callaghan, 2000) and NFSv4, the most recent, fourth version (Shepler et al., 2003). We will discuss the differences between them as well.
[Page 492]

The basic idea behind NFS is that each file server provides a standardized view of its local file system. In other words, it should not matter how that local file system is implemented; each NFS server supports the same model. This approach has been adopted for other distributed files systems as well. NFS comes with a communication protocol that allows clients to access the files stored on a server, thus allowing a heterogeneous collection of processes, possibly running on different operating systems and machines, to share a common file system.

The model underlying NFS and similar systems is that of a remote file service. In this model, clients are offered transparent access to a file system that is managed by a remote server. However, clients are normally unaware of the actual location of files. Instead, they are offered an interface to a file system that is similar to the interface offered by a conventional local file system. In particular, the client is offered only an interface containing various file operations, but the server is responsible for implementing those operations. This model is therefore also referred to as the remote access model. It is shown in Fig. 11-1(a).

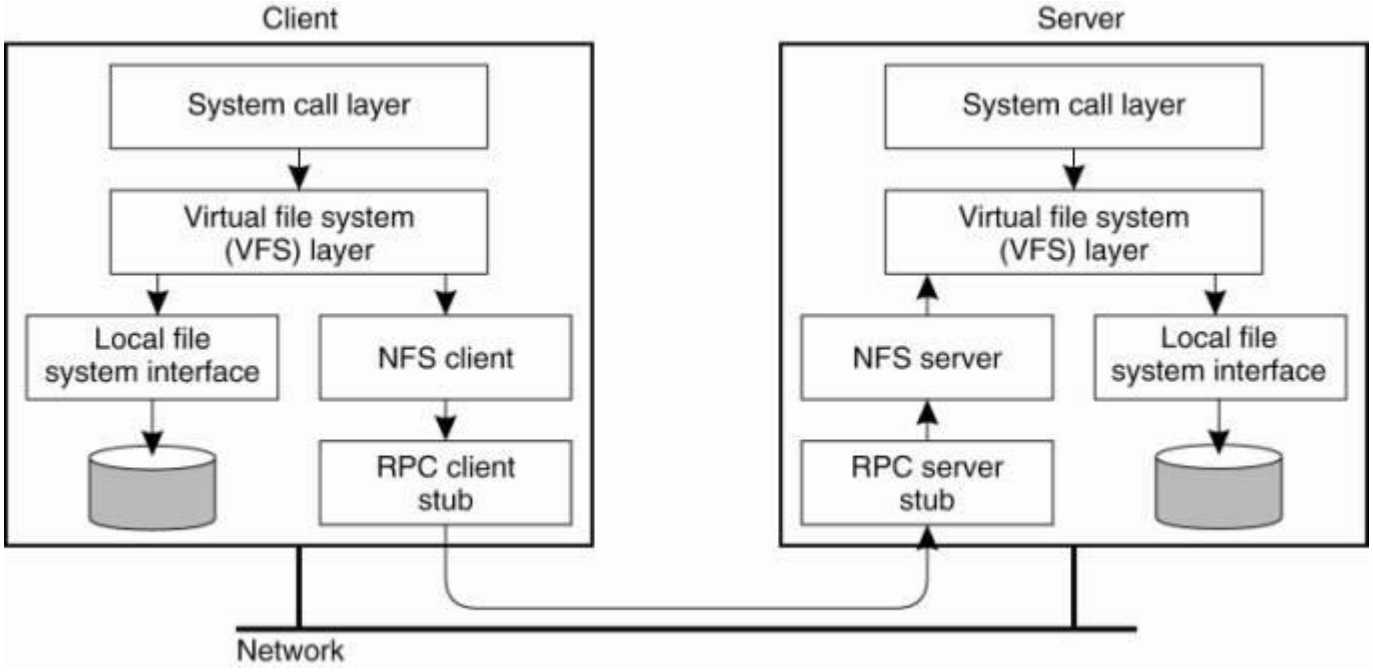
Figure 11-1. (a) The remote access model. (b) The upload/download model.



In contrast, in the upload/download model a client accesses a file locally after having downloaded it from the server, as shown in Fig. 11-1(b). When the client is finished with the file, it is uploaded back to the server again so that it can be used by another client. The Internet's FTP service can be used this way when a client downloads a complete file, modifies it, and then puts it back.

NFS has been implemented for a large number of different operating systems, although the UNIX-based versions are predominant. For virtually all modern UNIX systems, NFS is generally implemented following the layered architecture shown in Fig. 11-2.

Figure 11-2. The basic NFS architecture for UNIX systems.
(This item is displayed on page 493 in the print version)



A client accesses the file system using the system calls provided by its local operating system. However, the local UNIX file system interface is replaced by an interface to the Virtual File System (VFS), which by now is a de facto standard for interfacing to different (distributed) file systems (Kleiman, 1986). Virtually all modern operating systems provide VFS, and not doing so more or less forces developers to largely reimplement huge of an operating system when adopting a new file-system structure. With NFS, operations on the VFS interface are either passed to a local file system, or passed to a separate component known as the NFS client, which takes care of handling access to files stored at a remote server. In NFS, all client-server communication is done through RPCs. The NFS client implements the NFS file system operations as RPCs to the server. Note that the operations offered by the VFS interface can be different from those offered by the NFS client. The whole idea of the VFS is to hide the differences between various file systems.
[Page 493]

On the server side, we see a similar organization. The NFS server is responsible for handling incoming client requests. The RPC stub unmarshals requests and the NFS server converts them to regular VFS file operations that are subsequently passed to the VFS layer. Again, the VFS is responsible for implementing a local file system in which the actual files are stored.

An important advantage of this scheme is that NFS is largely independent of local file systems. In principle, it really does not matter whether the operating system at the client or server implements a UNIX file system, a Windows 2000 file system, or even an old MS-DOS file system. The only important issue is that these file systems are compliant with the file system model offered by NFS. For example, MS-DOS with its short file names cannot be used to implement an NFS server in a fully transparent way.

File System Model

The file system model offered by NFS is almost the same as the one offered by UNIX-based systems. Files are treated as uninterpreted sequences of bytes. They are hierarchically organized into a naming graph in which nodes represent directories and files. NFS also supports hard links as well as symbolic links, like any UNIX file system. Files are named, but are otherwise accessed by means of a UNIX-like file handle, which we discuss in detail below. In other words, to access a file, a client must first look up its name in a naming service and obtain the associated file handle. Furthermore, each file has a number of attributes whose values can be looked up and changed. We return to file naming in detail later in this chapter.

Fig. 11-3 shows the general file operations supported by NFS versions 3 and 4, respectively. The create operation is used to create a file, but has somewhat different meanings in NFSv3 and NFSv4. In version 3, the operation is used for creating regular files. Special files are created using separate operations. The link operation is used to create hard links. Symlink is used to create symbolic links. Mkdir is used to create subdirectories. Special files, such as device files, sockets, and named pipes are created by means of the mknod operation.

Figure 11-3. An incomplete list of file system operations supported by NFS.
(This item is displayed on page 495 in the print version)

Operation	v3	v4	Description
Create	Yes	No	Create a regular file

Create	No	Yes	Create a nonregular file
Link	Yes	Yes	Create a hard link to a file
Symlink	Yes	No	Create a symbolic link to a file
Mkdir	Yes	No	Create a subdirectory in a given directory
Mknod	Yes	No	Create a special file
Rename	Yes	Yes	Change the name of a file
Remove	Yes	Yes	Remove a file from a file system
Rmdir	Yes	No	Remove an empty subdirectory from a directory
Open	No	Yes	Open a file
Close	No	Yes	Close a file
Lookup	Yes	Yes	Look up a file by means of a file name
Readdir	Yes	Yes	Read the entries in a directory
Readlink	Yes	Yes	Read the path name stored in a symbolic link
Getattr	Yes	Yes	Get the attribute values for a file
Setattr	Yes	Yes	Set one or more attribute values for a file
Read	Yes	Yes	Read the data contained in a file
Write	Yes	Yes	Write data to a file

This situation is changed completely in NFSv4, where create is used for creating nonregular files, which include symbolic links, directories, and special files. Hard links are still created using a separate link operation, but regular files are created by means of the open operation, which is new to NFS and is a major deviation from the approach to file handling in older versions. Up until version 4, NFS was designed to allow its file servers to be stateless. For reasons we discuss later in this chapter, this design criterion has been abandoned in NFSv4, in which it is assumed that servers will generally maintain state between operations on the same file.

The operation rename is used to change the name of an existing file the same as in UNIX.

Files are deleted by means of the remove operation. In version 4, this operation is used to remove any kind of file. In previous versions, a separate rmdir operation was needed to remove a subdirectory. A file is removed by its name and has the effect that the number of hard links to it is decreased by one. If the number of links drops to zero, the file may be destroyed.

Version 4 allows clients to open and close (regular) files. Opening a nonexisting file has the side effect that a new file is created. To open a file, a client provides a name, along with various values for attributes. For example, a client may specify that a file should be opened for write access. After a file has been successfully opened, a client can access that file by means of its file handle. That handle is also used to close the file, by which the client tells the server that it will no longer need to have access to the file. The server, in turn, can release any state it maintained to provide that client access to the file.
[Page 495]

The lookup operation is used to look up a file handle for a given path name. In NFSv3, the lookup operation will not resolve a name beyond a mount point. (Recall from Chap. 5 that a mount point is a directory that essentially represents a link to a subdirectory in a foreign name space.) For example, assume that the name /remote/vu refers to a mount point in a naming

graph. When resolving the name /remote/vu/mbox, the lookup operation in NFSv3 will return the file handle for the mount point /remote/vu along with the remainder of the path name (i.e., mbox). The client is then required to explicitly mount the file system that is needed to complete the name lookup. A file system in this context is the collection of files, attributes, directories, and data blocks that are jointly implemented as a logical block device (Tanenbaum and Woodhull, 2006).

In version 4, matters have been simplified. In this case, lookup will attempt to resolve the entire name, even if this means crossing mount points. Note that this approach is possible only if a file system has already been mounted at mount points. The client is able to detect that a mount point has been crossed by inspecting the file system identifier that is later returned when the lookup completes.

There is a separate operation readdir to read the entries in a directory. This operation returns a list of (name, file handle) pairs along with attribute values that the client requested. The client can also specify how many entries should be returned. The operation returns an offset that can be used in a subsequent call to readdir in order to read the next series of entries.
[Page 496]

Operation readlink is used to read the data associated with a symbolic link. Normally, this data corresponds to a path name that can be subsequently looked up. Note that the lookup operation cannot handle symbolic links. Instead, when a symbolic link is reached, name resolution stops and the client is required to first call readlink to find out where name resolution should continue.

Files have various attributes associated with them. Again, there are important differences between NFS version 3 and 4, which we discuss in detail later. Typical attributes include the type of the file (telling whether we are dealing with a directory, a symbolic link, a special file, etc.), the file length, the identifier of the file system that contains the file, and the last time the file was modified. File attributes can be read and set using the operations getattr and setattr, respectively.

Finally, there are operations for reading data from a file, and writing data to a file. Reading data by means of the operation read is completely straightforward. The client specifies the offset and the number of bytes to be read. The client is returned the actual number of bytes that have been read, along with additional status information (e.g., whether the end-of-file has been reached).

Writing data to a file is done using the write operation. The client again specifies the position in the file where writing should start, the number of bytes to be written, and the data. In addition, it can instruct the server to ensure that all data are to be written to stable storage (we discussed stable storage in Chap. 8). NFS servers are required to support storage devices that can survive power supply failures, operating system failures, and hardware failures.

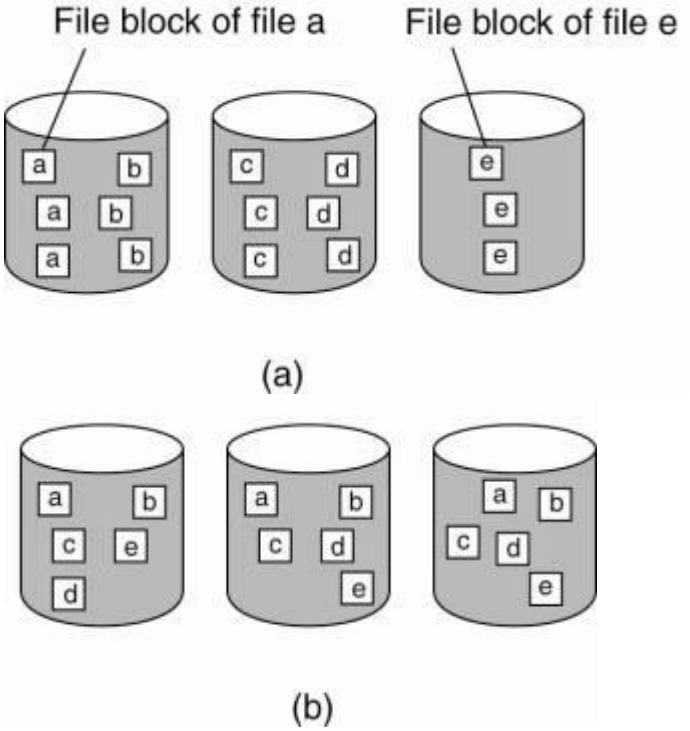
11.1.2. Cluster-Based Distributed File Systems

NFS is a typical example for many distributed file systems, which are generally organized according to a traditional client-server architecture. This architecture is often enhanced for server clusters with a few differences.

Considering that server clusters are often used for parallel applications, it is not surprising that their associated file systems are adjusted accordingly. One well-known technique is to deploy file-striping techniques, by which a single file is distributed across multiple servers. The basic idea is simple: by distributing a large file across multiple servers, it becomes possible to fetch different parts in parallel. Of course, such an organization works well only if the application is organized in such a way that parallel data access makes sense. This generally requires that the data as stored in the file have a very regular structure, for example, a (dense) matrix.

For general-purpose applications, or those with irregular or many different types of data structures, file striping may not be an effective tool. In those cases, it is often more convenient to partition the file system as a whole and simply store different files on different servers, but not to partition a single file across multiple servers. The difference between these two approaches is shown in Fig. 11-4.
[Page 497]

Figure 11-4. The difference between (a) distributing whole files across several servers and (b) striping files for parallel access.

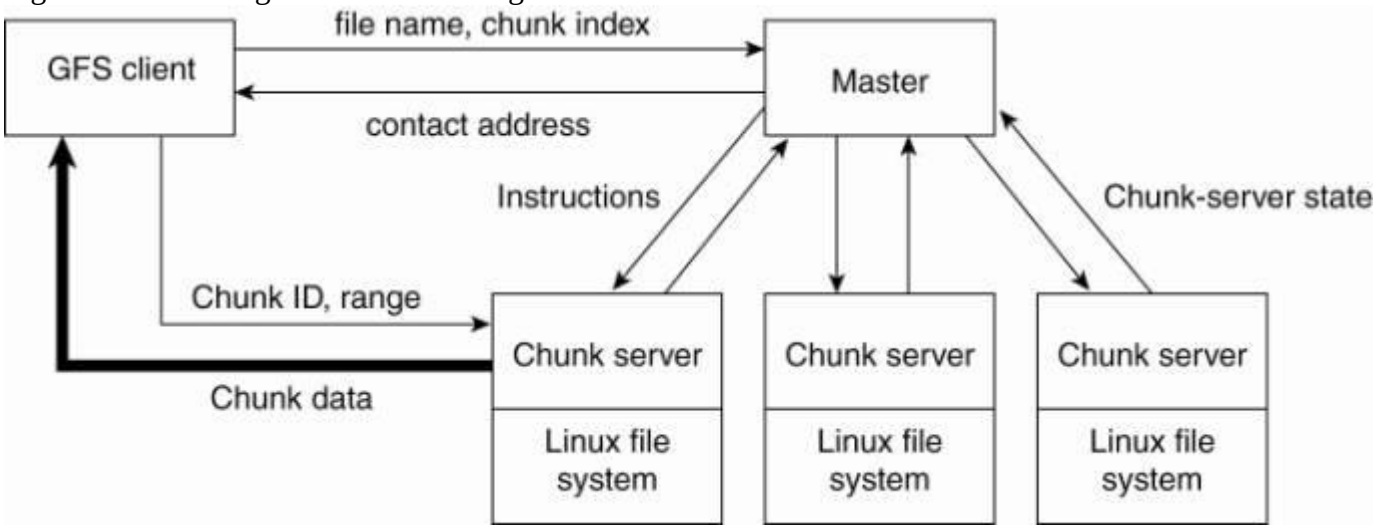


More interesting are the cases of organizing a distributed file system for very large data centers such as those used by companies like Amazon and Google. These companies offer services to Web clients resulting in reads and updates to a massive number of files distributed across literally tens of thousands of computers [see also Barroso et al. (2003)]. In such environments, the traditional assumptions concerning distributed file systems no longer hold. For example, we can expect that at any single moment there will be a computer malfunctioning.

To address these problems, Google, for example, has developed its own Google file system (GFS), of which the design is described in Ghemawat et al. (2003). Google files tend to be very

large, commonly ranging up to multiple gigabytes, where each one contains lots of smaller objects. Moreover, updates to files usually take place by appending data rather than overwriting parts of a file. These observations, along with the fact that server failures are the norm rather than the exception, lead to constructing clusters of servers as shown in Fig. 11-5.

Figure 11-5. The organization of a Google cluster of servers.



Each GFS cluster consists of a single master along with multiple chunk servers. Each GFS file is divided into chunks of 64 Mbyte each, after which these chunks are distributed across what are called chunk servers. An important observation is that a GFS master is contacted only for metadata information. In particular, a GFS client passes a file name and chunk index to the master, expecting a contact address for the chunk. The contact address contains all the information to access the correct chunk server to obtain the required file chunk.
[Page 498]

To this end, the GFS master essentially maintains a name space, along with a mapping from file name to chunks. Each chunk has an associated identifier that will allow a chunk server to lookup it up. In addition, the master keeps track of where a chunk is located. Chunks are replicated to handle failures, but no more than that. An interesting feature is that the GFS master does not attempt to keep an accurate account of chunk locations. Instead, it occasionally contacts the chunk servers to see which chunks they have stored.

The advantage of this scheme is simplicity. Note that the master is in control of allocating chunks to chunk servers. In addition, the chunk servers keep an account of what they have stored. As a consequence, once the master has obtained chunk locations, it has an accurate picture of where data is stored. However, matters would become complicated if this view had to be consistent all the time. For example, every time a chunk server crashes or when a server is added, the master would need to be informed. Instead, it is much simpler to refresh its information from the current set of chunk servers through polling. GFS clients simply get to know which chunk servers the master believes is storing the requested data. Because chunks are replicated anyway, there is a high probability that a chunk is available on at least one of the chunk servers.

Why does this scheme scale? An important design issue is that the master is largely in control, but that it does not form a bottleneck due to all the work it needs to do. Two important types of measures have been taken to accommodate scalability.

First, and by far the most important one, is that the bulk of the actual work is done by chunk servers. When a client needs to access data, it contacts the master to find out which chunk servers hold that data. After that, it communicates only with the chunk servers. Chunks are replicated according to a primary-backup scheme. When the client is performing an update operation, it contacts the nearest chunk server holding that data, and pushes its updates to that server. This server will push the update to the next closest one holding the data, and so on. Once all updates have been propagated, the client will contact the primary chunk server, who will then assign a sequence number to the update operation and pass it on to the backups. Meanwhile, the master is kept out of the loop.

Second, the (hierarchical) name space for files is implemented using a simple single-level table, in which path names are mapped to metadata (such as the equivalent of inodes in traditional file systems). Moreover, this entire table is kept in main memory, along with the mapping of files to chunks. Updates on these data are logged to persistent storage. When the log becomes too large, a checkpoint is made by which the main-memory data is stored in such a way that it can be immediately mapped back into main memory. As a consequence, the intensity of I/O of a GFS master is strongly reduced.
[Page 499]

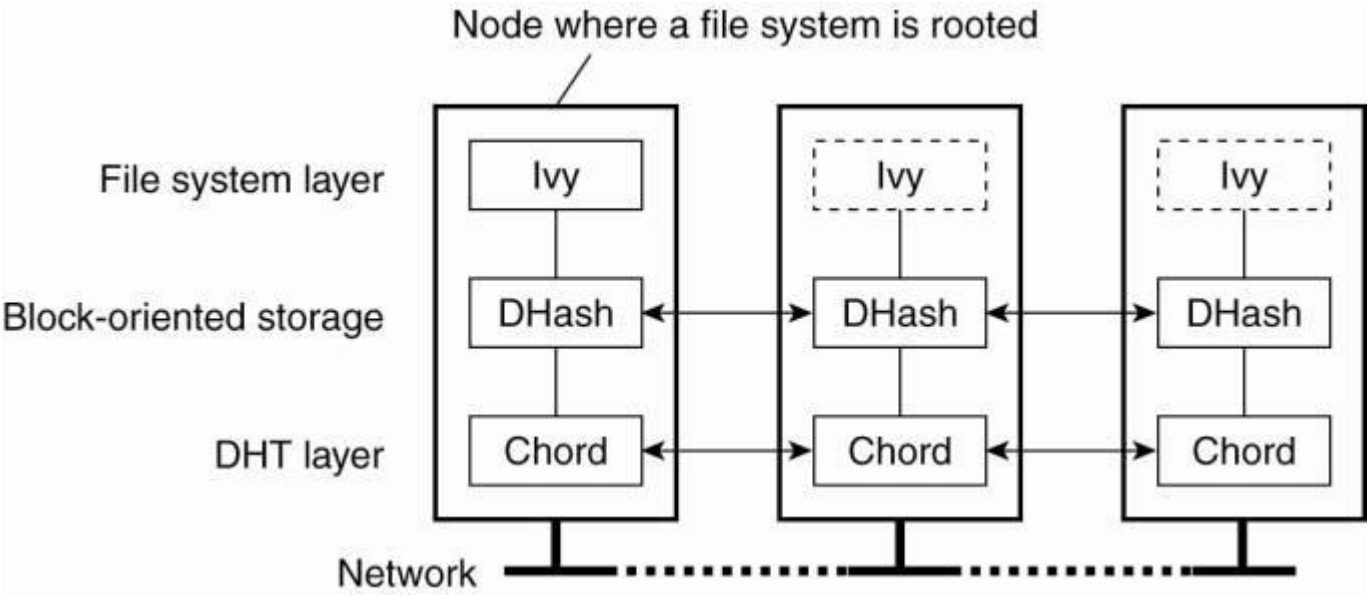
This organization allows a single master to control a few hundred chunk servers, which is a considerable size for a single cluster. By subsequently organizing a service such as Google into smaller services that are mapped onto clusters, it is not hard to imagine that a huge collection of clusters can be made to work together.

11.1.3. Symmetric Architectures

Of course, fully symmetric organizations that are based on peer-to-peer technology also exist. All current proposals use a DHT-based system for distributing data, combined with a key-based lookup mechanism. An important difference is whether they build a file system on top of a distributed storage layer, or whether whole files are stored on the participating nodes.

An example of the first type of file system is Ivy, a distributed file system that is built using a Chord DHT-based system. Ivy is described in Muthitacharoen et al. (2002). Their system essentially consists of three separate layers as shown in Fig. 11-6. The lowest layer is formed by a Chord system providing basic decentralized lookup facilities. In the middle is a fully distributed block-oriented storage layer. Finally, on top there is a layer implementing an NFS-like file system.

Figure 11-6. The organization of the Ivy distributed file system.



Data storage in Ivy is realized by a Chord-based, block-oriented distributed storage system called DHash (Dabek et al., 2001). In essence, DHash is quite simple. It only knows about data blocks, each block typically having a size of 8 KB. Ivy uses two kinds of data blocks. A content-hash block has an associated key, which is computed as the secure hash of the block's content. In this way, whenever a block is looked up, a client can immediately verify whether the correct block has been looked up, or that another or corrupted version is returned. Furthermore, Ivy also makes use of public-key blocks, which are blocks having a public key as lookup key, and whose content has been signed with the associated private key.
[Page 500]

To increase availability, DHash replicates every block B to the k immediate successors of the server responsible for storing B. In addition, looked up blocks are also cached along the route that the lookup request followed.

Files are implemented as a separate data structure on top of DHash. To achieve this goal, each user maintains a log of operations it carries out on files. For simplicity, we assume that there is only a single user per node so that each node will have its own log. A log is a linked list of immutable records, where each record contains all the information related to an operation on the Ivy file system. Each node appends records only to its own, local, log. Only a log's head is mutable, and points to the most recently appended record. Each record is stored in a separate content-hash block, whereas a log's head is kept in a public-key block.

There are different types of records, roughly corresponding to the different operations supported by NFS. For example, when performing an update operation on a file, a write record is created, containing the file's identifier along with the offset for the pile pointer and the data that is being written. Likewise, there are records for creating files (i.e., adding a new inode), manipulating directories, etc.

To create a new file system, a node simply creates a new log along with a new inode that will serve as the root. Ivy deploys what is known as an NFS loopback server which is just a local

user-level server that accepts NFS requests from local clients. In the case of Ivy, this NFS server supports mounting the newly created file system allowing applications to access it as any other NFS file system.

When performing a read operation, the local Ivy NFS server makes a pass over the log, collecting data from those records that represent write operations on the same block of data, allowing it to retrieve the most recently stored values. Note that because each record is stored as a DHash block, multiple lookups across the overlay network may be needed to retrieve the relevant values.

Instead of using a separate block-oriented storage layer, alternative designs propose to distribute whole files instead of data blocks. The developers of Kosha (Butt et al. 2004) propose to distribute files at a specific directory level. In their approach, each node has a mount point named /kosha containing the files that are to be distributed using a DHT-based system. Distributing files at directory level 1 means that all files in a subdirectory /kosha/a will be stored at the same node. Likewise, distribution at level 2 implies that all files stored in subdirectory /kosha/a/aa are stored at the same node. Taking a level-1 distribution as an example, the node responsible for storing files under /kosha/a is found by computing the hash of a and taking that as the key in a lookup.

The potential drawback of this approach is that a node may run out of disk space to store all the files contained in the subdirectory that it is responsible for. Again, a simple solution is found in placing a branch of that subdirectory on another node and creating a symbolic link to where the branch is now stored.

11.2. Processes

When it comes to processes, distributed file systems have no unusual properties. In many cases, there will be different types of cooperating processes: storage servers and file managers, just as we described above for the various organizations.

The most interesting aspect concerning file system processes is whether or not they should be stateless. NFS is a good example illustrating the trade-offs. One of its long-lasting distinguishing features (compared to other distributed file systems), was the fact that servers were stateless. In other words, the NFS protocol did not require that servers maintained any client state. This approach was followed in versions 2 and 3, but has been abandoned for version 4.

The primary advantage of the stateless approach is simplicity. For example, when a stateless server crashes, there is essentially no need to enter a recovery phase to bring the server to a previous state. However, as we explained in Chap. 8, we still need to take into account that the client cannot be given any guarantees whether or not a request has actually been carried out.

The stateless approach in the NFS protocol could not always be fully followed in practical implementations. For example, locking a file cannot easily be done by a stateless server. In the case of NFS, a separate lock manager is used to handle this situation. Likewise, certain authentication protocols require that the server maintains state on its clients. Nevertheless, NFS

servers could generally be designed in such a way that only very little information on clients needed to be maintained. For the most part, the scheme worked adequately.

Starting with version 4, the stateless approach was abandoned, although the new protocol is designed in such a way that a server does not need to maintain much information about its clients. Besides those just mentioned, there are other reasons to choose for a stateful approach. An important reason is that NFS version 4 is expected to also work across wide-area networks. This requires that clients can make effective use of caches, in turn requiring an efficient cache consistency protocol. Such protocols often work best in collaboration with a server that maintains some information on files as used by its clients. For example, a server may associate a lease with each file it hands out to a client, promising to give the client exclusive read and write access until the lease expires or is refreshed. We return to such issues later in this chapter.

The most apparent difference with the previous versions is the support for the open operation. In addition, NFS supports callback procedures by which a server can do an RPC to a client. Clearly, callbacks also require a server to keep track of its clients.

Similar reasoning has affected the design of other distributed file systems. By and large, it turns out that maintaining a fully stateless design can be quite difficult, often leading to building stateful solutions as an enhancement, such as is the case with NFS file locking.

11.3. Communication

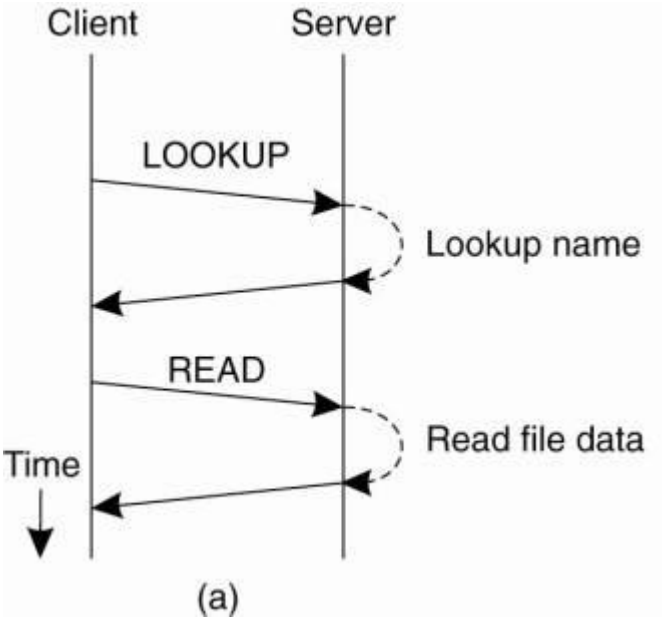
As with processes, there is nothing particularly special or unusual about communication in distributed file systems. Many of them are based on remote procedure calls (RPCs), although some interesting enhancements have been made to support special cases. The main reason for choosing an RPC mechanism is to make the system independent from underlying operating systems, networks, and transport protocols.

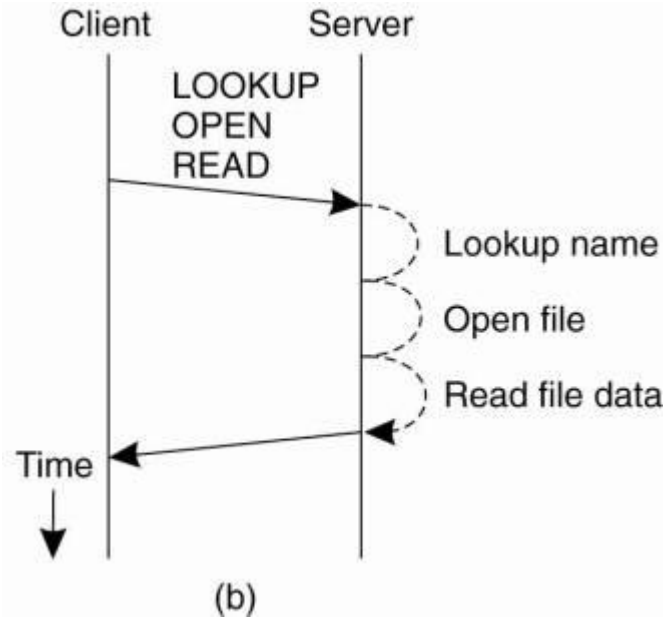
11.3.1. RPCs in NFS

For example, in NFS, all communication between a client and server proceeds along the Open Network Computing RPC (ONC RPC) protocol, which is formally defined in Srinivasan (1995a), along with a standard for representing marshaled data (Srinivasan, 1995b). ONC RPC is similar to other RPC systems as we discussed in Chap. 4.

Every NFS operation can be implemented as a single remote procedure call to a file server. In fact, up until NFSv4, the client was made responsible for making the server's life as easy as possible by keeping requests relatively simple. For example, in order to read data from a file for the first time, a client normally first had to look up the file handle using the lookup operation, after which it could issue a read request, as shown in Fig. 11-7(a).

Figure 11-7. (a) Reading data from a file in NFS version 3. (b) Reading data using a compound procedure in version 4.





This approach required two successive RPCs. The drawback became apparent when considering the use of NFS in a wide-area system. In that case, the extra latency of a second RPC led to performance degradation. To circumvent such problems, NFSv4 supports compound procedures by which several RPCs can be grouped into a single request, as shown in Fig. 11-7(b).
[Page 503]

In our example, the client combines the lookup and read request into a single RPC. In the case of version 4, it is also necessary to open the file before reading can take place. After the file handle has been looked up, it is passed to the open operation, after which the server continues

with the read operation. The overall effect in this example is that only two messages need to be exchanged between the client and server.

There are no transactional semantics associated with compound procedures. The operations grouped together in a compound procedure are simply handled in the order as requested. If there are concurrent operations from other clients, then no measures are taken to avoid conflicts. If an operation fails for whatever reason, then no further operations in the compound procedure are executed, and the results found so far are returned to the client. For example, if lookup fails, a succeeding open is not even attempted.

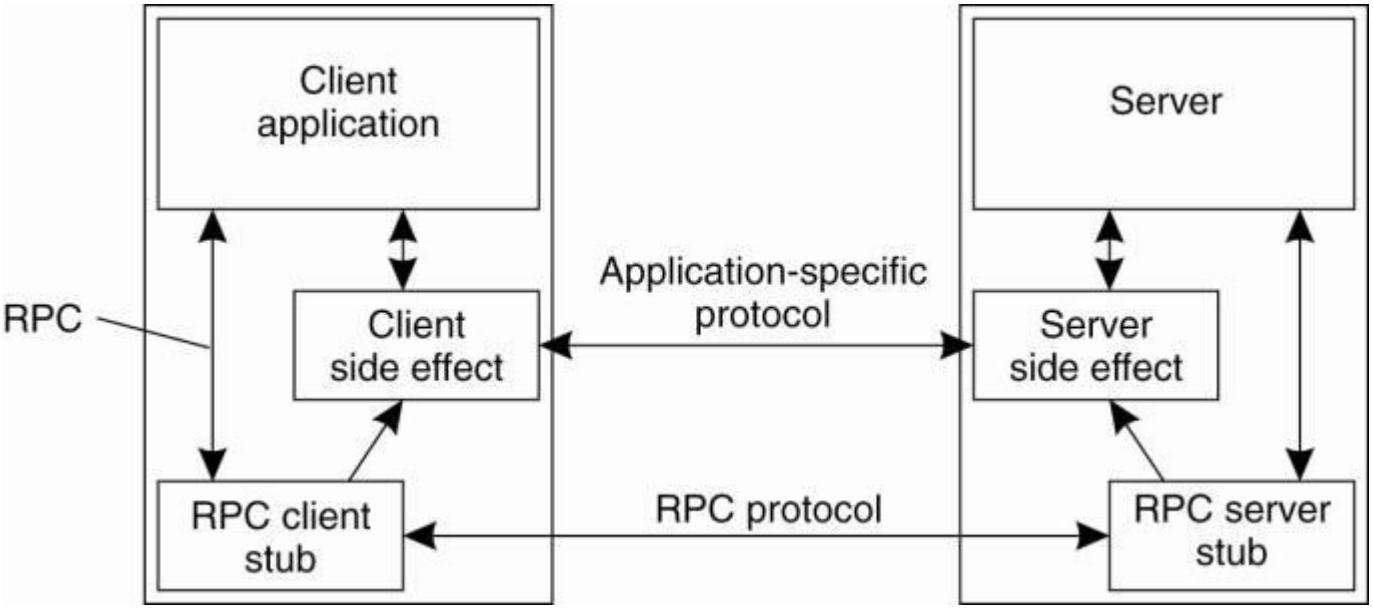
11.3.2. The RPC2 Subsystem

Another interesting enhancement to RPCs has been developed as part of the Coda file system (Kistler and Satyanarayanan, 1992). RPC2 is a package that offers reliable RPCs on top of the (unreliable) UDP protocol. Each time a remote procedure is called, the RPC2 client code starts a new thread that sends an invocation request to the server and subsequently blocks until it receives an answer. As request processing may take an arbitrary time to complete, the server regularly sends back messages to the client to let it know it is still working on the request. If the server dies, sooner or later this thread will notice that the messages have ceased and report back failure to the calling application.

An interesting aspect of RPC2 is its support for side effects. A side effect is a mechanism by which the client and server can communicate using an application-specific protocol. Consider, for example, a client opening a file at a video server. What is needed in this case is that the client and server set up a continuous data stream with an isochronous transmission mode. In other words, data transfer from the server to the client is guaranteed to be within a minimum and maximum end-to-end delay.

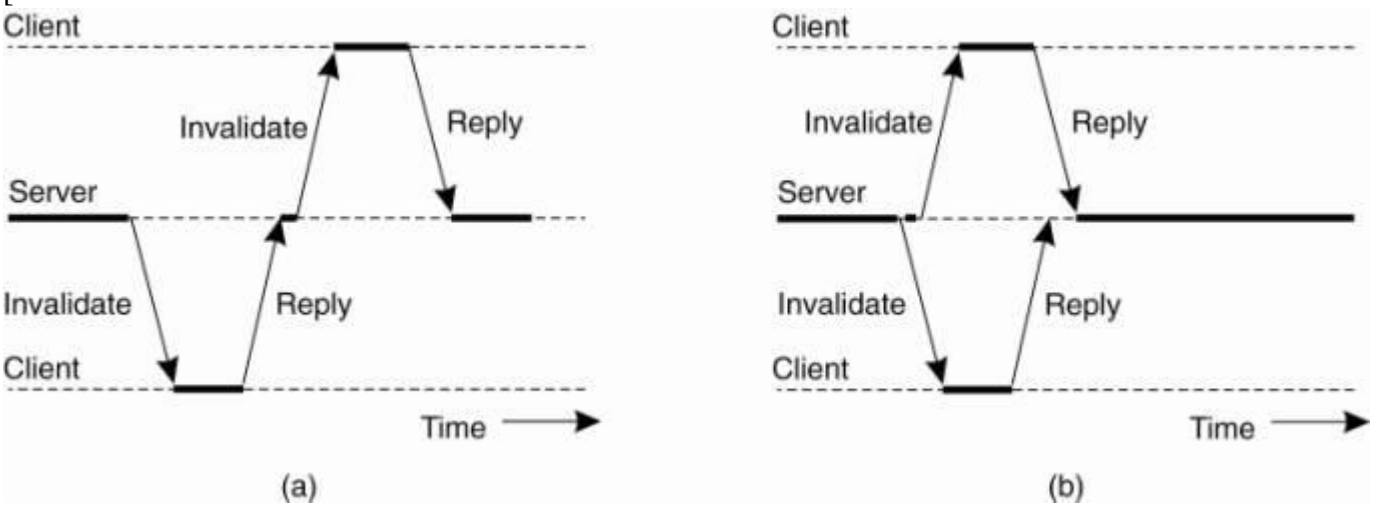
RPC2 allows the client and the server to set up a separate connection for transferring the video data to the client on time. Connection setup is done as a side effect of an RPC call to the server. For this purpose, the RPC2 runtime system provides an interface of side-effect routines that is to be implemented by the application developer. For example, there are routines for setting up a connection and routines for transferring data. These routines are automatically called by the RPC2 runtime system at the client and server, respectively, but their implementation is otherwise completely independent of RPC2. This principle of side effects is shown in Fig. 11-8.

Figure 11-8. Side effects in Coda's RPC2 system.
(This item is displayed on page 504 in the print version)



Another feature of RPC2 that makes it different from other RPC systems is its support for multicasting. An important design issue in Coda is that servers keep track of which clients have a local copy of a file. When a file is modified, a server invalidates local copies by notifying the appropriate clients through an RPC. Clearly, if a server can notify only one client at a time, invalidating all clients may take some time, as illustrated in Fig. 11-9(a).
[Page 504]

Figure 11-9. (a) Sending an invalidation message one at a time. (b) Sending invalidation messages in parallel.

[


The problem is caused by the fact that an RPC may occasionally fail. Invalidating files in a strict sequential order may be delayed considerably because the server cannot reach a possibly crashed client, but will give up on that client only after a relatively long expiration time. Meanwhile, other clients will still be reading from their local copies.

An alternative (and better) solution is shown in Fig. 11-9(b). Here, instead of invalidating each copy one by one, the server sends an invalidation message to all clients at the same time. As a consequence, all nonfailing clients are notified in the same time as it would take to do an immediate RPC. Also, the server notices within the usual expiration time that certain clients are failing to respond to the RPC, and can declare such clients as being crashed.
[Page 505]

Parallel RPCs are implemented by means of the MultiRPC system, which is part of the RPC2 package (Satyanarayanan and Siegel, 1990). An important aspect of MultiRPC is that the parallel invocation of RPCs is fully transparent to the callee. In other words, the receiver of a MultiRPC call cannot distinguish that call from a normal RPC. At the caller's side, parallel execution is also largely transparent. For example, the semantics of MultiRPC in the presence of failures are much the same as that of a normal RPC. Likewise, the side-effect mechanisms can be used in the same way as before.

MultiRPC is implemented by essentially executing multiple RPCs in parallel. This means that the caller explicitly sends an RPC request to each recipient. However, instead of immediately waiting for a response, it defers blocking until all requests have been sent. In other words, the caller invokes a number of one-way RPCs, after which it blocks until all responses have been received from the nonfailing recipients. An alternative approach to parallel execution of RPCs in MultiRPC is provided by setting up a multicast group, and sending an RPC to all group members using IP multicast.

11.3.3. File-Oriented Communication in Plan 9

Finally, it is worth mentioning a completely different approach to handling communication in distributed file systems. Plan 9 (Pike et al., 1995). is not so much a distributed file system, but rather a file-based distributed system. All resources are accessed in the same way, namely with file-like syntax and operations, including even resources such as processes and network interfaces. This idea is inherited from UNIX, which also attempts to offer file-like interfaces to resources, but it has been exploited much further and more consistently in Plan 9. To illustrate, network interfaces are represented by a file system, in this case consisting of a collection of special files. This approach is similar to UNIX, although network interfaces in UNIX are represented by files and not file systems. (Note that a file system in this context is again the logical block device containing all the data and metadata that comprise a collection of files.) In Plan 9, for example, an individual TCP connection is represented by a subdirectory consisting of the files shown in Fig. 11-10.

Figure 11-10. Files associated with a single TCP connection in Plan 9.
(This item is displayed on page 506 in the print version)

File	Description
ctl	Used to write protocol-specific control commands
data	Used to read and write data
listen	Used to accept incoming connection setup requests
local	Provides information on the caller's side of the connection

remote	Provides information on the other side of the connection
status	Provides diagnostic information on the current status of the connection

The file ctl is used to send control commands to the connection. For example, to open a telnet session to a machine with IP address 192.31.231.42 using port 23, requires that the sender writes the text string "connect 192.31.231.42!23" to file ctl. The receiver would previously have written the string "announce 23" to its own ctl file, indicating that it can accept incoming session requests.

The data file is used to exchange data by simply performing read and write operations. These operations follow the usual UNIX semantics for file operations.
[Page 506]

For example, to write data to a connection, a process simply invokes the operation

```
res = write(fd, buf, nbytes);
```

where fd is the file descriptor returned after opening the data file, buf is a pointer to a buffer containing the data to be written, and nbytes is the number of bytes that should be extracted from the buffer. The number of bytes actually written is returned and stored in the variable res.

The file listen is used to wait for connection setup requests. After a process has announced its willingness to accept new connections, it can do a blocking read on file listen. If a request comes in, the call returns a file descriptor to a new ctl file corresponding to a newly-created connection directory. It is thus seen how a completely file-oriented approach toward communication can be realized.

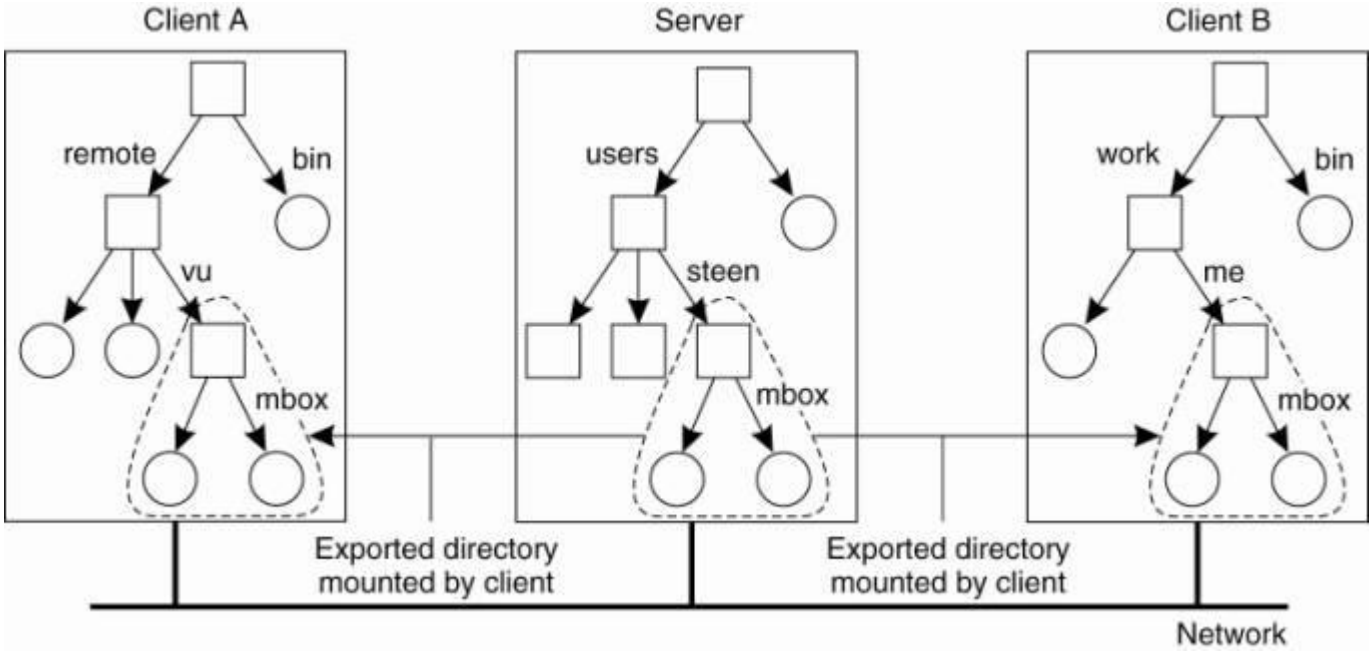
11.4. Naming

Naming arguably plays an important role in distributed file systems. In virtually all cases, names are organized in a hierarchical name space like those we discussed in Chap. 5. In the following we will again consider NFS as a representative for how naming is often handled in distributed file systems.

11.4.1. Naming in NFS

The fundamental idea underlying the NFS naming model is to provide clients complete transparent access to a remote file system as maintained by a server. This transparency is achieved by letting a client be able to mount a remote file system into its own local file system, as shown in Fig. 11-11.

Figure 11-11. Mounting (part of) a remote file system in NFS.
(This item is displayed on page 507 in the print version)



Instead of mounting an entire file system, NFS allows clients to mount only part of a file system, as also shown in Fig. 11-11. A server is said to export a directory when it makes that directory and its entries available to clients. An exported directory can be mounted into a client's local name space.
[Page 507]

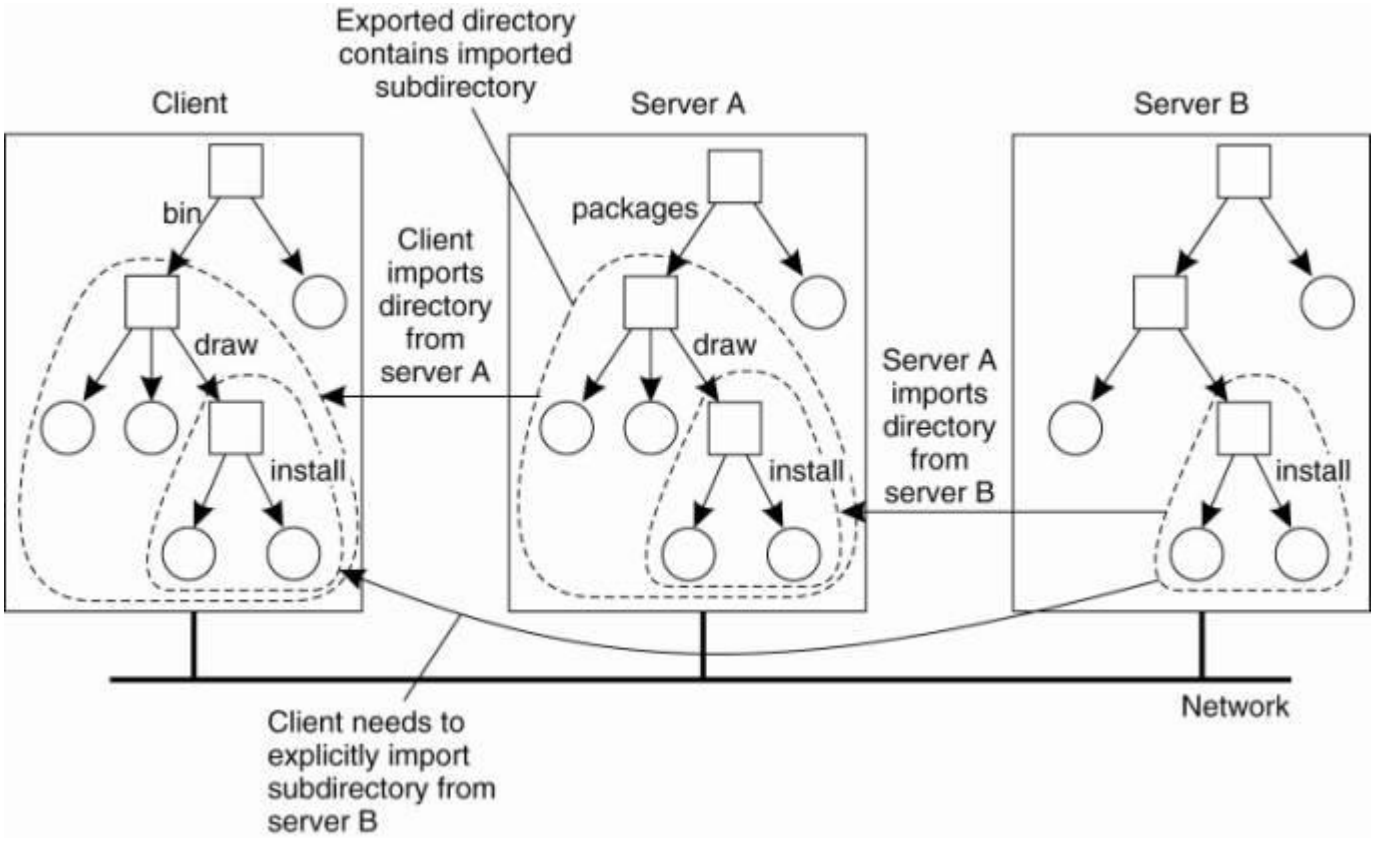
This design approach has a serious implication: in principle, users do not share name spaces. As shown in Fig. 11-11, the file named /remote/vu/mbox at client A is named /work/me/mbox at client B. A file's name therefore depends on how clients organize their own local name space, and where exported directories are mounted. The drawback of this approach in a distributed file system is that sharing files becomes much harder. For example, Alice cannot tell Bob about a file using the name she assigned to that file, for that name may have a completely different meaning in Bob's name space of files.

There are several ways to solve this problem, but the most common one is to provide each client with a name space that is partly standardized. For example, each client may be using the local directory /usr/bin to mount a file system containing a standard collection of programs that are available to everyone. Likewise, the directory /local may be used as a standard to mount a local file system that is located on the client's host.

An NFS server can itself mount directories that are exported by other servers. However, it is not allowed to export those directories to its own clients. Instead, a client will have to explicitly mount such a directory from the server that maintains it, as shown in Fig. 11-12. This restriction comes partly from simplicity. If a server could export a directory that it mounted from another server, it would have to return special file handles that include an identifier for a server. NFS does not support such file handles.

Figure 11-12. Mounting nested directories from multiple servers in NFS.

(This item is displayed on page 508 in the print version)



To explain this point in more detail, assume that server A hosts a file system FSA from which it exports the directory /packages. This directory contains a subdirectory /draw that acts as a mount point for a file system FSB that is exported by server B and mounted by A. Let A also export /packages/draw to its own clients, and assume that a client has mounted /packages into its local directory /bin as shown in Fig. 11-12.
[Page 508]

If name resolution is iterative (as is the case in NFSv3), then to resolve the name /bin/draw/install, the client contacts server A when it has locally resolved /bin and requests A to return a file handle for directory /draw. In that case, server A should return a file handle that includes an identifier for server B, for only B can resolve the rest of the path name, in this case /install. As we have said, this kind of name resolution is not supported by NFS.

Name resolution in NFSv3 (and earlier versions) is strictly iterative in the sense that only a single file name at a time can be looked up. In other words, resolving a name such as /bin/draw/install requires three separate calls to the NFS server. Moreover, the client is fully responsible for implementing the resolution of a path name. NFSv4 also supports recursive name lookups. In this case, a client can pass a complete path name to a server and request that server to resolve it.

There is another peculiarity with NFS name lookups that has been solved with version 4. Consider a file server hosting several file systems. With the strict iterative name resolution in version 3, whenever a lookup was done for a directory on which another file system was mounted, the lookup would return the file handle of the directory. Subsequently reading that directory would return its original content, not that of the root directory of the mounted file system.
[Page 509]

To explain, assume that in our previous example that both file systems FSA and FSB are hosted by a single server. If the client has mounted /packages into its local directory /bin, then looking up the file name draw at the server would return the file handle for draw. A subsequent call to the server for listing the directory entries of draw by means of readdir would then return the list of directory entries that were originally stored in FSA in subdirectory /packages/draw. Only if the client had also mounted file system FSB, would it be possible to properly resolve the path name draw/install relative to /bin.

NFSv4 solves this problem by allowing lookups to cross mount points at a server. In particular, lookup returns the file handle of the mounted directory instead of that of the original directory. The client can detect that the lookup has crossed a mount point by inspecting the file system identifier of the looked up file. If required, the client can locally mount that file system as well.

File Handles

A file handle is a reference to a file within a file system. It is independent of the name of the file it refers to. A file handle is created by the server that is hosting the file system and is unique with respect to all file systems exported by the server. It is created when the file is created. The client is kept ignorant of the actual content of a file handle; it is completely opaque. File handles were 32 bytes in NFS version 2, but were variable up to 64 bytes in version 3 and 128 bytes in version 4. Of course, the length of a file handle is not opaque.

Ideally, a file handle is implemented as a true identifier for a file relative to a file system. For one thing, this means that as long as the file exists, it should have one and the same file handle. This persistence requirement allows a client to store a file handle locally once the associated file has been looked up by means of its name. One benefit is performance: as most file operations require a file handle instead of a name, the client can avoid having to look up a name repeatedly before every file operation. Another benefit of this approach is that the client can now access the file independent of its (current) names.

Because a file handle can be locally stored by a client, it is also important that a server does not reuse a file handle after deleting a file. Otherwise, a client may mistakenly access the wrong file when it uses its locally stored file handle.

Note that the combination of iterative name lookups and not letting a lookup operation allow crossing a mount point introduces a problem with getting an initial file handle. In order to access files in a remote file system, a client will need to provide the server with a file handle of the directory where the lookup should take place, along with the name of the file or directory that is to be resolved. NFSv3 solves this problem through a separate mount protocol, by which a client actually mounts a remote file system. After mounting, the client is passed back the root file handle of the mounted file system, which it can subsequently use as a starting point for looking up names.

[Page 510]

In NFSv4, this problem is solved by providing a separate operation putrootfh that tells the server to solve all file names relative to the root file handle of the file system it manages. The root file handle can be used to look up any other file handle in the server's file system. This approach has the additional benefit that there is no need for a separate mount protocol. Instead, mounting can be integrated into the regular protocol for looking up files. A client can simply mount a remote file system by requesting the server to resolve names relative to the file system's root file handle using putrootfh.

Automounting

As we mentioned, the NFS naming model essentially provides users with their own name space. Sharing in this model may become difficult if users name the same file differently. One solution to this problem is to provide each user with a local name space that is partly standardized, and subsequently mounting remote file systems the same for each user.

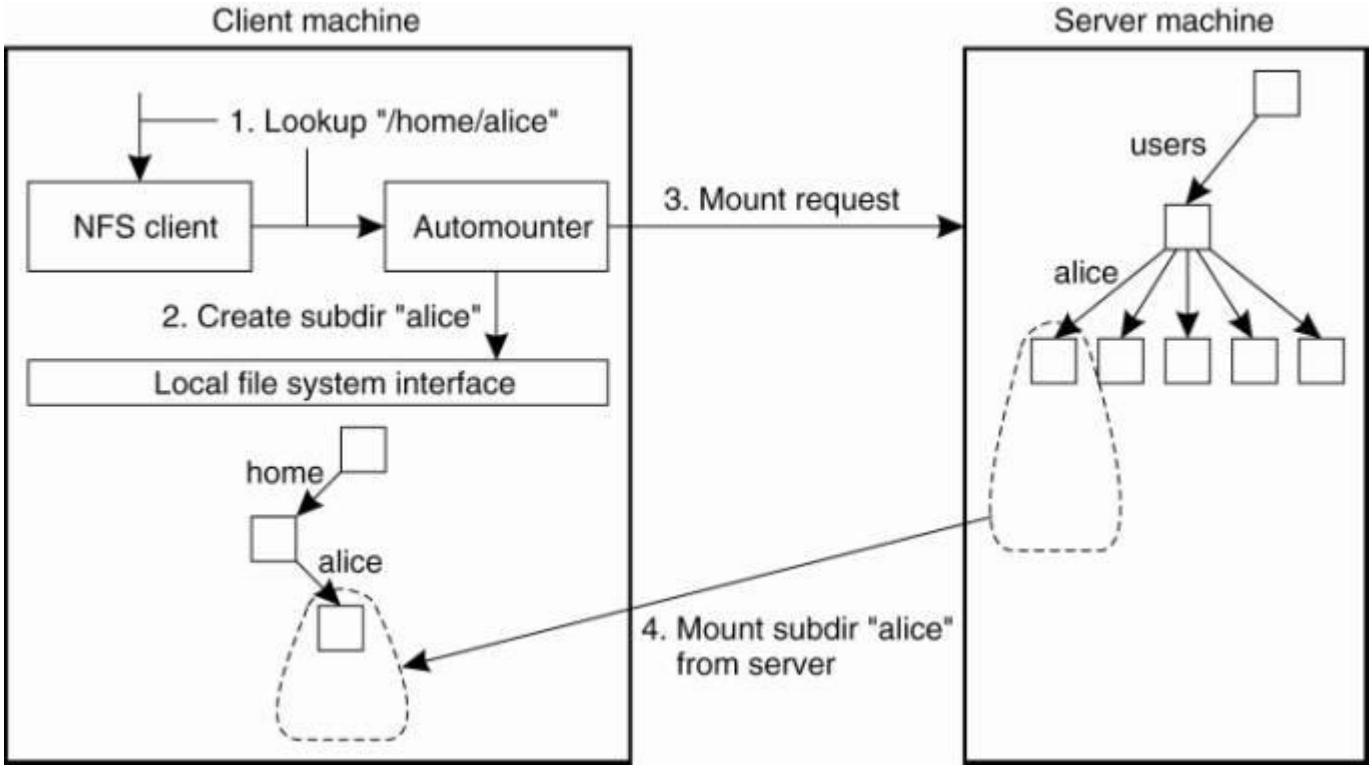
Another problem with the NFS naming model has to do with deciding when a remote file system should be mounted. Consider a large system with thousands of users. Assume that each user has a local directory /home that is used to mount the home directories of other users. For example, Alice's home directory may be locally available to her as /home/alice, although the actual files are stored on a remote server. This directory can be automatically mounted when Alice logs into her workstation. In addition, she may have access to Bob's public files by accessing Bob's directory through /home/bob.

The question, however, is whether Bob's home directory should also be mounted automatically when Alice logs in. The benefit of this approach would be that the whole business of mounting file systems would be transparent to Alice. However, if this policy were followed for every user, logging in could incur a lot of communication and administrative overhead. In addition, it would require that all users are known in advance. A much better approach is to transparently mount another user's home directory on demand, that is, when it is first needed.

On-demand mounting of a remote file system (or actually an exported directory) is handled in NFS by an automounter, which runs as a separate process on the client's machine. The principle underlying an automounter is relatively simple. Consider a simple automounter implemented as a user-level NFS server on a UNIX operating system. For alternative implementations, see Callaghan (2000).

Assume that for each user, the home directories of all users are available through the local directory /home, as described above. When a client machine boots, the automounter starts with mounting this directory. The effect of this local mount is that whenever a program attempts to access /home, the UNIX kernel will forward a lookup operation to the NFS client, which in this case, will forward the request to the automounter in its role as NFS server, as shown in Fig. 11-13.

Figure 11-13. A simple automounter for NFS.
(This item is displayed on page 511 in the print version)



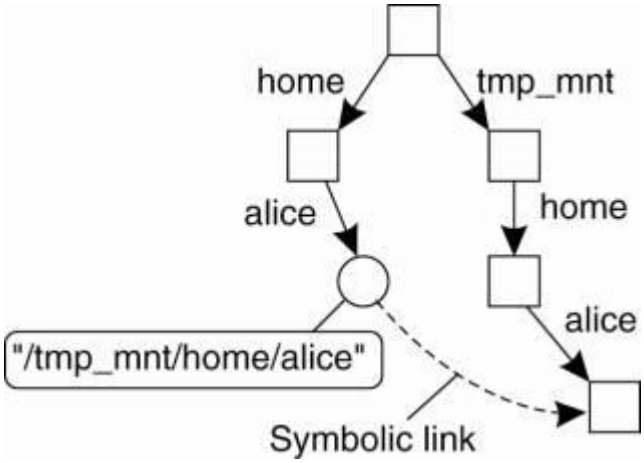
[Page 511]

For example, suppose that Alice logs in. The login program will attempt to read the directory /home/alice to find information such as login scripts. The automounter will thus receive the request to look up subdirectory /home/alice, for which reason it first creates a subdirectory /alice in /home. It then looks up the NFS server that exports Alice's home directory to subsequently mount that directory in /home/alice. At that point, the login program can proceed.

The problem with this approach is that the automounter will have to be involved in all file operations to guarantee transparency. If a referenced file is not locally available because the corresponding file system has not yet been mounted, the automounter will have to know. In particular, it will need to handle all read and write requests, even for file systems that have already been mounted. This approach may incur a large performance problem. It would be better to have the auto mounter only mount/unmount directories, but otherwise stay out of the loop.

A simple solution is to let the automounter mount directories in a special subdirectory, and install a symbolic link to each mounted directory. This approach is shown in Fig. 11-14.

Figure 11-14. Using symbolic links with automounting.
(This item is displayed on page 512 in the print version)



In our example, the user home directories are mounted as subdirectories of /tmp_mnt. When Alice logs in, the automounter mounts her home directory in /tmp_mnt/home/alice and creates a symbolic link /home/alice that refers to that subdirectory. In this case, whenever Alice executes a command such as

ls –l /home/alice

the NFS server that exports Alice's home directory is contacted directly without further involvement of the automounter.
[Page 512]

11.4.2. Constructing a Global Name Space

Large distributed systems are commonly constructed by gluing together various legacy systems into one whole. When it comes to offering shared access to files, having a global name space is about the minimal glue that one would like to have. At present, file systems are mostly opened for sharing by using primitive means such as access through FTP. This approach, for example, is generally used in Grid computing.

More sophisticated approaches are followed by truly wide-area distributed file systems, but these often require modifications to operating system kernels in order to be adopted. Therefore, researchers have been looking for approaches to integrate existing file systems into a single, global name space but using only user-level solutions. One such system, simply called Global Name Space Service (GNS) is proposed by Anderson et al. (2004).

GNS does not provide interfaces to access files. Instead, it merely provides the means to set up a global name space in which several existing name spaces have been merged. To this end, a GNS client maintains a virtual tree in which each node is either a directory or a junction. A junction is a special node that indicates that name resolution is to be taken over by another process, and as such bears some resemblance with a mount point in traditional file system. There are five different types of junctions, as shown in Fig. 11-15.

Figure 11-15. Junctions in GNS.

(This item is displayed on page 513 in the print version)

Junction	Description
GNS junction	Refers to another GNS instance
Logical file-system name	Reference to subtree to be looked up in a location service
Logical file name	Reference to a file to be looked up in a location service
Physical file-system name	Reference to directly remote-accessible subtree
Physical file name	Reference to directly remote-accessible file

A GNS junction simply refers to another GNS instance, which is just another virtual tree hosted at possibly another process. The two logical junctions contain information that is needed to contact a location service. The latter will provide the contact address for accessing a file system and a file, respectively. A physical file-system name refers to a file system at another server, and corresponds largely to a contact address that a logical junction would need. For example, a URL such as ftp://ftp.cs.vu.nl/pub would contain all the information to access files at the indicated FTP server. Analogously, a URL such as http://www.cs.vu.nl/index.htm is a typical example of a physical file name.
[Page 513]

Obviously, a junction should contain all the information needed to continue name resolution. There are many ways of doing this, but considering that there are so many different file systems, each specific junction will require its own implementation. Fortunately, there are also many common ways of accessing remote files, including protocols for communicating with NFS servers, FTP servers, and Windows-based machines (notably CIFS).

GNS has the advantage of decoupling the naming of files from their actual location. In no way does a virtual tree relate to where files and directories are physically placed. In addition, by using a location service it is also possible to move files around without rendering their names unresolvable. In that case, the new physical location needs to be registered at the location service. Note that this is completely the same as what we have discussed in Chap. 5.

11.5. Synchronization

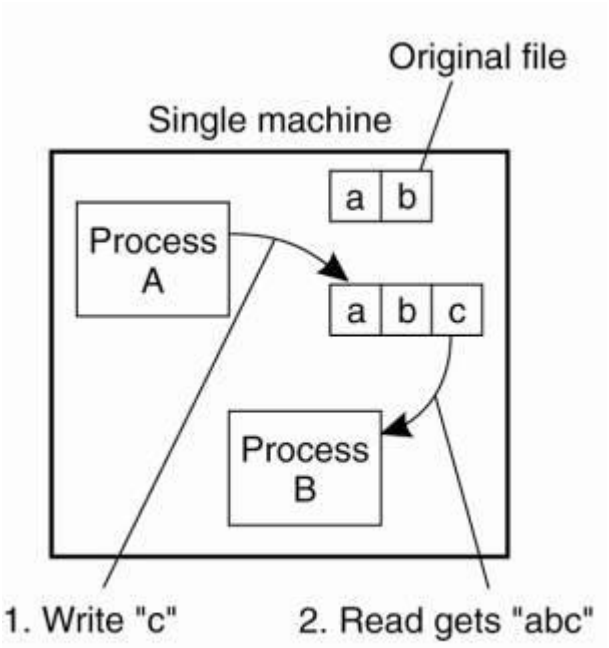
Let us now continue our discussion by focusing on synchronization issues in distributed file systems. There are various issues that require our attention. In the first place, synchronization for file systems would not be an issue if files were not shared. However, in a distributed system, the semantics of file sharing becomes a bit tricky when performance issues are at stake. To this end, different solutions have been proposed of which we discuss the most important ones next.

11.5.1. Semantics of File Sharing

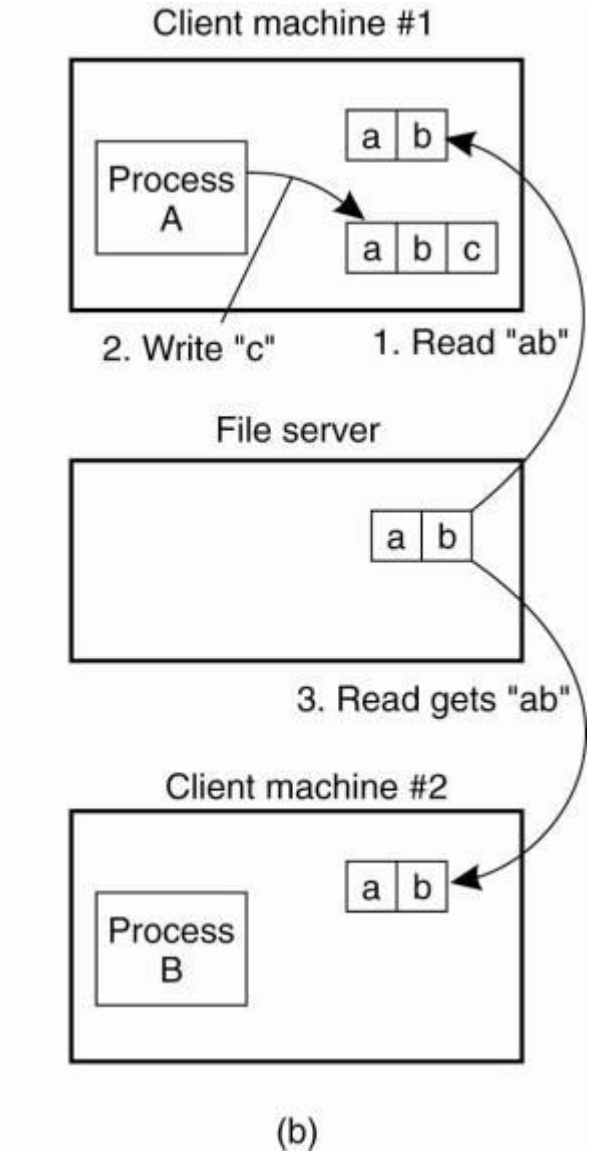
When two or more users share the same file at the same time, it is necessary to define the semantics of reading and writing precisely to avoid problems. In single-processor systems that permit processes to share files, such as UNIX, the semantics normally state that when a read operation follows a write operation, the read returns the value just written, as shown in Fig. 11-16(a). Similarly, when two writes happen in quick succession, followed by a read, the value read is the value stored by the last write. In effect, the system enforces an absolute time ordering

on all operations and always returns the most recent value. We will refer to this model as UNIX semantics. This model is easy to understand and straightforward to implement.
[Page 514]

Figure 11-16. (a) On a single processor, when a read follows a write, the value returned by the read is the value just written. (b) In a distributed system with caching, obsolete values may be returned.


Original file
Single machine
a b
Process A
a b c
Process B
1. Write "c"
2. Read gets "abc"

(a)



(b)

In a distributed system, UNIX semantics can be achieved easily as long as there is only one file server and clients do not cache files. All reads and writes go directly to the file server, which processes them strictly sequentially. This approach gives UNIX semantics (except for the minor problem that network delays may cause a read that occurred a microsecond after a write to arrive at the server first and thus gets the old value).

In practice, however, the performance of a distributed system in which all file requests must go to a single server is frequently poor. This problem is often solved by allowing clients to maintain local copies of heavily-used files in their private (local) caches. Although we will discuss the details of file caching below, for the moment it is sufficient to point out that if a client locally modifies a cached file and shortly thereafter another client reads the file from the server, the second client will get an obsolete file, as illustrated in Fig. 11-16(b).
[Page 515]

One way out of this difficulty is to propagate all changes to cached files back to the server immediately. Although conceptually simple, this approach is inefficient. An alternative solution is to relax the semantics of file sharing. Instead of requiring a read to see the effects of all previous writes, one can have a new rule that says: "Changes to an open file are initially visible only to the process (or possibly machine) that modified the file. Only when the file is closed are the changes made visible to other processes (or machines)." The adoption of such a rule does not change what happens in Fig. 11-16(b), but it does redefine the actual behavior (B getting the original value of the file) as being the correct one. When A closes the file, it sends a copy to the server, so that subsequent reads get the new value, as required.

This rule is widely-implemented and is known as session semantics. Most distributed file systems implement session semantics. This means that although in theory they follow the remote access model of Fig. 11-1(a), most implementations make use of local caches, effectively implementing the upload/download model of Fig. 11-1(b).

Using session semantics raises the question of what happens if two or more clients are simultaneously caching and modifying the same file. One solution is to say that as each file is closed in turn, its value is sent back to the server, so the final result depends on whose close request is most recently processed by the server. A less pleasant, but easier to implement alternative is to say that the final result is one of the candidates, but leave the choice of which one unspecified.

A completely different approach to the semantics of file sharing in a distributed system is to make all files immutable. There is thus no way to open a file for writing. In effect, the only operations on files are create and read.

What is possible is to create an entirely new file and enter it into the directory system under the name of a previous existing file, which now becomes inaccessible (at least under that name). Thus although it becomes impossible to modify the file x, it remains possible to replace x by a new file atomically. In other words, although files cannot be updated, directories can be. Once we have decided that files cannot be changed at all, the problem of how to deal with two processes, one of which is writing on a file and the other of which is reading it, just disappears, greatly simplifying the design.

What does remain is the problem of what happens when two processes try to replace the same file at the same time. As with session semantics, the best solution here seems to be to allow one of the new files to replace the old one, either the last one or nondeterministically.

A somewhat stickier problem is what to do if a file is replaced while another process is busy reading it. One solution is to somehow arrange for the reader to continue using the old file, even if it is no longer in any directory, analogous to the way UNIX allows a process that has a file open to continue using it, even after it has been deleted from all directories. Another solution is to detect that the file has changed and make subsequent attempts to read from it fail. [Page 516]

A fourth way to deal with shared files in a distributed system is to use atom\%ic transactions. To summarize briefly, to access a file or a group of files, a process first executes some type of BEGIN_TRANSACTION primitive to signal that what follows must be executed indivisibly. Then come system calls to read and write one or more files. When the requested work has been completed, an END_TRANSACTION primitive is executed. The key property of this method

is that the system guarantees that all the calls contained within the transaction will be carried out in order, without any interference from other, concurrent transactions. If two or more transactions start up at the same time, the system ensures that the final result is the same as if they were all run in some (undefined) sequential order.

In Fig. 11-17 we summarize the four approaches we have discussed for dealing with shared files in a distributed system.

Figure 11-17. Four ways of dealing with the shared files in a distributed system.

Method	Comment
UNIX semantics	Every operation on a file is instantly visible to all processes
Session semantics	No changes are visible to other processes until the file is closed
Immutable files	No updates are possible; simplifies sharing and replication
Transactions	All changes occur atomically

11.5.2. File Locking

Notably in client-server architectures with stateless servers, we need additional facilities for synchronizing access to shared files. The traditional way of doing this is to make use of a lock manager. Without exception, a lock manager follows the centralized locking scheme as we discussed in Chap. 6.

However, matters are not as simple as we just sketched. Although a central lock manager is generally deployed, the complexity in locking comes from the need to allow concurrent access to the same file. For this reason, a great number of different locks exist, and moreover, the granularity of locks may also differ. Let us consider NFSv4 again.

Conceptually, file locking in NFSv4 is simple. There are essentially only four operations related to locking, as shown in Fig. 11-18. NFSv4 distinguishes read locks from write locks. Multiple clients can simultaneously access the same part of a file provided they only read data. A write lock is needed to obtain exclusive access to modify part of a file.

Figure 11-18. NFSv4 operations related to file locking.
(This item is displayed on page 517 in the print version)

Operation	Description
Lock	Create a lock for a range of bytes
Lockt	Test whether a conflicting lock has been granted
Locku	Remove a lock from a range of bytes
Renew	Renew the lease on a specified lock

Operation lock is used to request a read or write lock on a consecutive range of bytes in a file. It is a nonblocking operation; if the lock cannot be granted due to another conflicting lock, the client gets back an error message and has to poll the server at a later time. There is no automatic retry. Alternatively, the client can request to be put on a FIFO-ordered list maintained by the server. As soon as the conflicting lock has been removed, the server will grant the next lock to the client at the top of the list, provided it polls the server before a certain time expires. This

approach prevents the server from having to notify clients, while still being fair to clients whose lock request could not be granted because grants are made in FIFO order.
[Page 517]

The lockt operation is used to test whether a conflicting lock exists. For example, a client can test whether there are any read locks granted on a specific range of bytes in a file, before requesting a write lock for those bytes. In the case of a conflict, the requesting client is informed exactly who is causing the conflict and on which range of bytes. It can be implemented more efficiently than lock, because there is no need to attempt to open a file.

Removing a lock from a file is done by means of the locku operation.

Locks are granted for a specific time (determined by the server). In other words, they have an associated lease. Unless a client renews the lease on a lock it has been granted, the server will automatically remove it. This approach is followed for other server-provided resources as well and helps in recovery after failures. Using the renew operation, a client requests the server to renew the lease on its lock (and, in fact, other resources as well).

In addition to these operations, there is also an implicit way to lock a file, referred to as share reservation. Share reservation is completely independent from locking, and can be used to implement NFS for Windows-based systems. When a client opens a file, it specifies the type of access it requires (namely READ, WRITE, or BOTH), and which type of access the server should deny other clients (NONE, READ, WRITE, or BOTH). If the server cannot meet the client's requirements, the open operation will fail for that client. In Fig. 11-19 we show exactly what happens when a new client opens a file that has already been successfully opened by another client. For an already opened file, we distinguish two different state variables. The access state specifies how the file is currently being accessed by the current client. The denial state specifies what accesses by new clients are not permitted.

Figure 11-19. The result of an open operation with share reservations in NFS. (a) When the client requests shared access given the current denial state. (b) When the client requests a denial state given the current file access state.
(This item is displayed on page 518 in the print version)

		Current file denial state			
		NONE	**READ**	**WRITE**	**BOTH**
Request access	**READ**	Succeed	Fail	Succeed	Fail
	WRITE	Succeed	Succeed	Fail	Fail
	BOTH	Succeed	Fail	Fail	Fail

(a)

		Requested file denial state			
		NONE	READ	WRITE	BOTH
Current access state	READ	Succeed	Fail	Succeed	Fail
	WRITE	Succeed	Succeed	Fail	Fail
	BOTH	Succeed	Fail	Fail	Fail

(b)

In Fig. 11-19(a), we show what happens when a client tries to open a file requesting a specific type of access, given the current denial state of that file. Likewise, Fig. 11-19(b) shows the result of opening a file that is currently being accessed by another client, but now requesting certain access types to be disallowed.
[Page 518]

NFSv4 is by no means an exception when it comes to offering synchronization mechanisms for shared files. In fact, it is by now accepted that any simple set of primitives such as only complete-file locking, reflects poor design. Complexity in locking schemes comes mostly from the fact that a fine granularity of locking is required to allow for concurrent access to shared files. Some attempts to reduce complexity while keeping performance have been taken [see, e.g., Burns et al. (2001)], but the situation remains somewhat unsatisfactory. In the end, we may be looking at completely redesigning our applications for scalability rather than trying to patch situations that come from wanting to share data the way we did in nondistributed systems.

11.5.3. Sharing Files in Coda

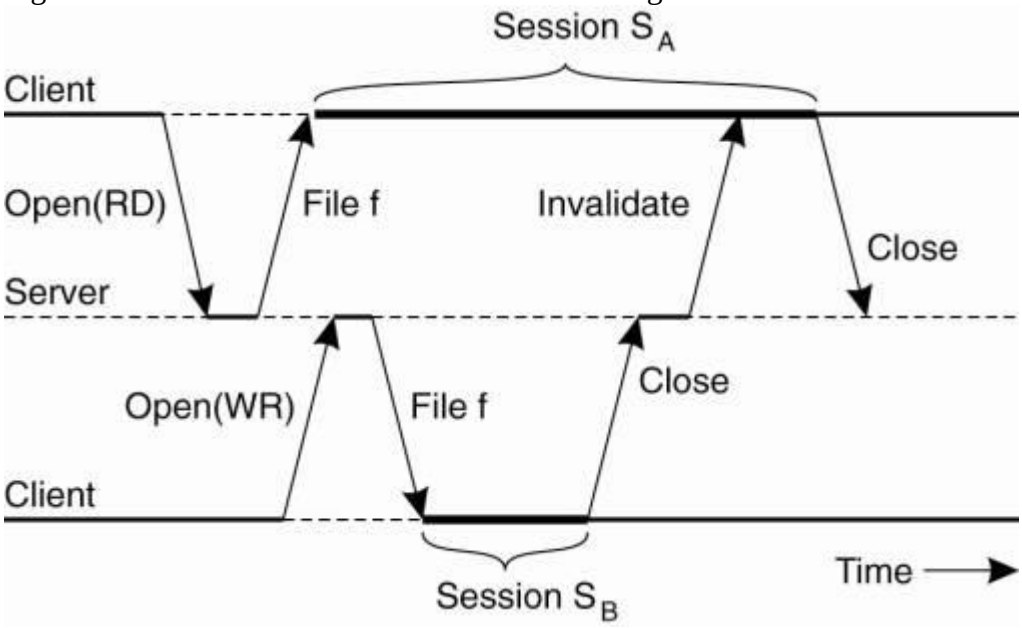
The session semantics in NFS dictate that the last process that closes a file will have its changes propagated to the server; any updates in concurrent, but earlier sessions will be lost. A somewhat more subtle approach can also be taken. To accommodate file sharing, the Coda file system (Kistler and Satyanaryanan, 1992) uses a special allocation scheme that bears some similarities to share reservations in NFS. To understand how the scheme works, the following is important. When a client successfully opens a file f, an entire copy of f is transferred to the client's machine. The server records that the client has a copy of f. So far, this approach is similar to open delegation in NFS.
[Page 519]

Now suppose client A has opened file f for writing. When another client B wants to open f as well, it will fail. This failure is caused by the fact that the server has recorded that client A might have already modified f. On the other hand, had client A opened f for reading, an attempt by client B to get a copy from the server for reading would succeed. An attempt by B to open for writing would succeed as well.

Now consider what happens when several copies of f have been stored locally at various clients. Given what we have just said, only one client will be able to modify f. If this client modifies f and subsequently closes the file, the file will be transferred back to the server. However, every other client may proceed to read its local copy despite the fact that the copy is actually outdated.

The reason for this apparently inconsistent behavior is that a session is treated as a transaction in Coda. Consider Fig. 11-20, which shows the time line for two processes, A and B. Assume A has opened f for reading, leading to session SA. Client B has opened f for writing, shown as session SB.

Figure 11-20. The transactional behavior in sharing files in Coda.



When B closes session SB, it transfers the updated version of f to the server, which will then send an invalidation message to A. A will now know that it is reading from an older version of f. However, from a transactional point of view, this really does not matter because session SA could be considered to have been scheduled before session SB.

11.6. Consistency and Replication

Caching and replication play an important role in distributed file systems, most notably when they are designed to operate over wide-area networks. In what follows, we will take a look at various aspects related to client-side caching of file data, as well as the replication of file servers. Also, we consider the role of replication in peer-to-peer file-sharing systems.
[Page 520]

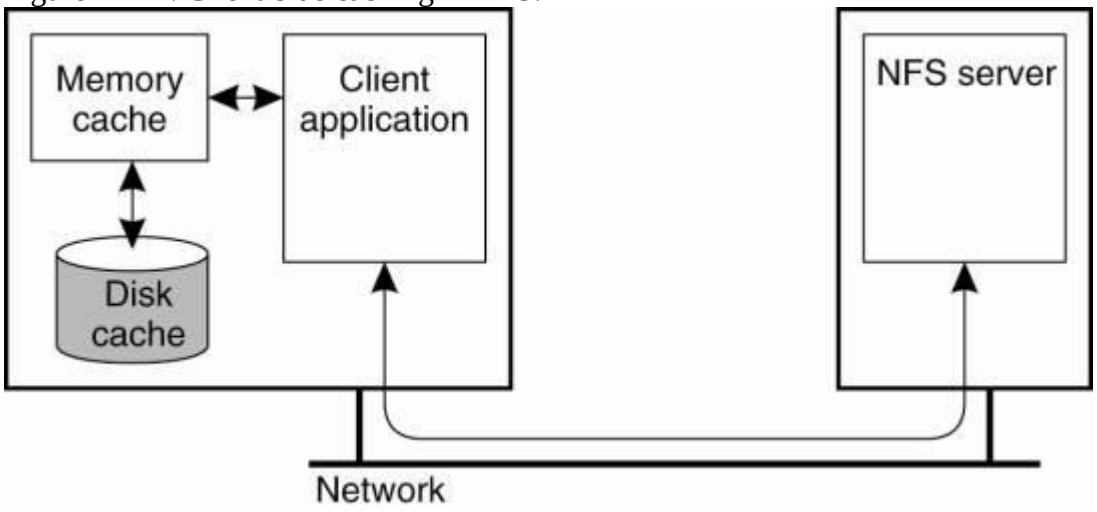
11.6.1. Client-Side Caching

To see how client-side caching is deployed in practice, we return to our example systems NFS and Coda.
Caching in NFS

Caching in NFSv3 has been mainly left outside of the protocol. This approach has led to the implementation of different caching policies, most of which never guaranteed consistency. At best, cached data could be stale for a few seconds compared to the data stored at a server. However, implementations also exist that allowed cached data to be stale for 30 seconds without the client knowing. This state of affairs is less than desirable.

NFSv4 solves some of these consistency problems, but essentially still leaves cache consistency to be handled in an implementation-dependent way. The general caching model that is assumed by NFS is shown in Fig. 11-21. Each client can have a memory cache that contains data previously read from the server. In addition, there may also be a disk cache that is added as an extension to the memory cache, using the same consistency parameters.

Figure 11-21. Client-side caching in NFS.



Typically, clients cache file data, attributes, file handles, and directories. Different strategies exist to handle consistency of the cached data, cached attributes, and so on. Let us first take a look at caching file data.

NFSv4 supports two different approaches for caching file data. The simplest approach is when a client opens a file and caches the data it obtains from the server as the result of various read operations. In addition, write operations can be carried out in the cache as well. When the client closes the file, NFS requires that if modifications have taken place, the cached data must be flushed back to the server. This approach corresponds to implementing session semantics as discussed earlier.
[Page 521]

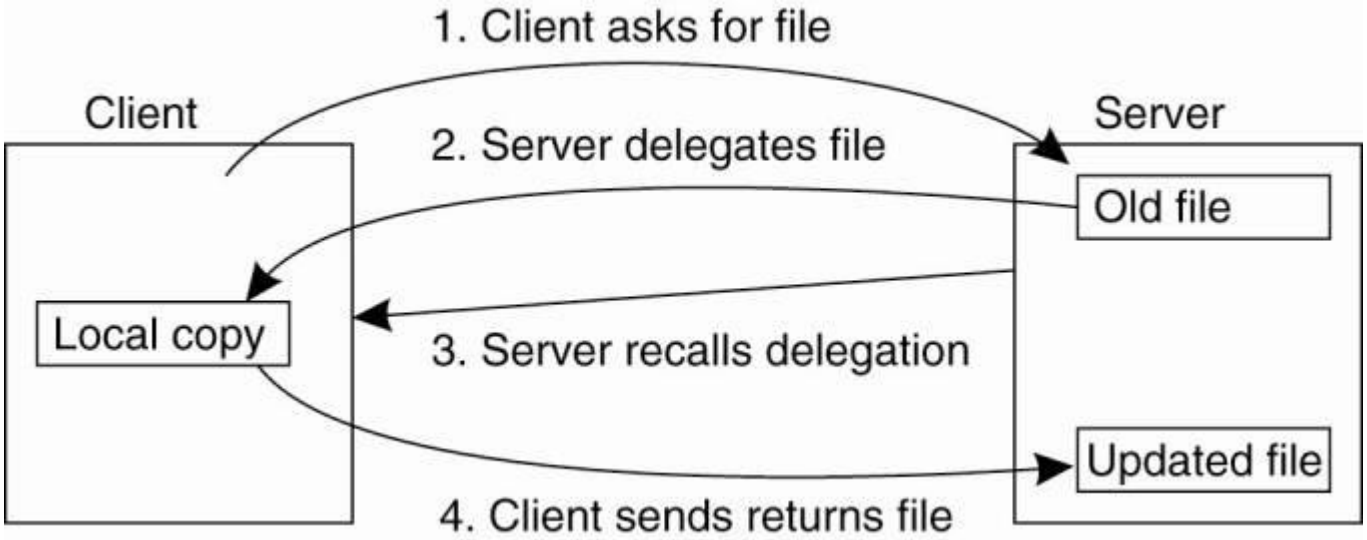
Once (part of) a file has been cached, a client can keep its data in the cache even after closing the file. Also, several clients on the same machine can share a single cache. NFS requires that whenever a client opens a previously closed file that has been (partly) cached, the client must immediately revalidate the cached data. Revalidation takes place by checking when the file was last modified and invalidating the cache in case it contains stale data.

In NFSv4 a server may delegate some of its rights to a client when a file is opened. Open delegation takes place when the client machine is allowed to locally handle open and close operations from other clients on the same machine. Normally, the server is in charge of checking whether opening a file should succeed or not, for example, because share reservations need to be taken into account. With open delegation, the client machine is sometimes allowed to make such decisions, avoiding the need to contact the server.

For example, if a server has delegated the opening of a file to a client that requested write permissions, file locking requests from other clients on the same machine can also be handled locally. The server will still handle locking requests from clients on other machines, by simply denying those clients access to the file. Note that this scheme does not work in the case of delegating a file to a client that requested only read permissions. In that case, whenever another local client wants to have write permissions, it will have to contact the server; it is not possible to handle the request locally.

An important consequence of delegating a file to a client is that the server needs to be able to recall the delegation, for example, when another client on a different machine needs to obtain access rights to the file. Recalling a delegation requires that the server can do a callback to the client, as illustrated in Fig. 11-22.

Figure 11-22. Using the NFSv4 callback mechanism to recall file delegation.



A callback is implemented in NFS using its underlying RPC mechanisms. Note, however, that callbacks require that the server keeps track of clients to which it has delegated a file. Here, we see another example where an NFS server can no longer be implemented in a stateless manner. Note, however, that the combination of delegation and stateful servers may lead to various problems in the presence of client and server failures. For example, what should a server do when it had delegated a file to a now unresponsive client? As we discuss shortly, leases will generally form an adequate practical solution.
[Page 522]

Clients can also cache attribute values, but are largely left on their own when it comes to keeping cached values consistent. In particular, attribute values of the same file cached by two different clients may be different unless the clients keep these attributes mutually consistent. Modifications to an attribute value should be immediately forwarded to the server, thus following a write-through cache coherence policy.

A similar approach is followed for caching file handles (or rather, the name-to-file handle mapping) and directories. To mitigate the effects of inconsistencies, NFS uses leases on cached attributes, file handles, and directories. After some time has elapsed, cache entries are thus automatically invalidated and revalidation is needed before they are used again.

Client-Side Caching in Coda

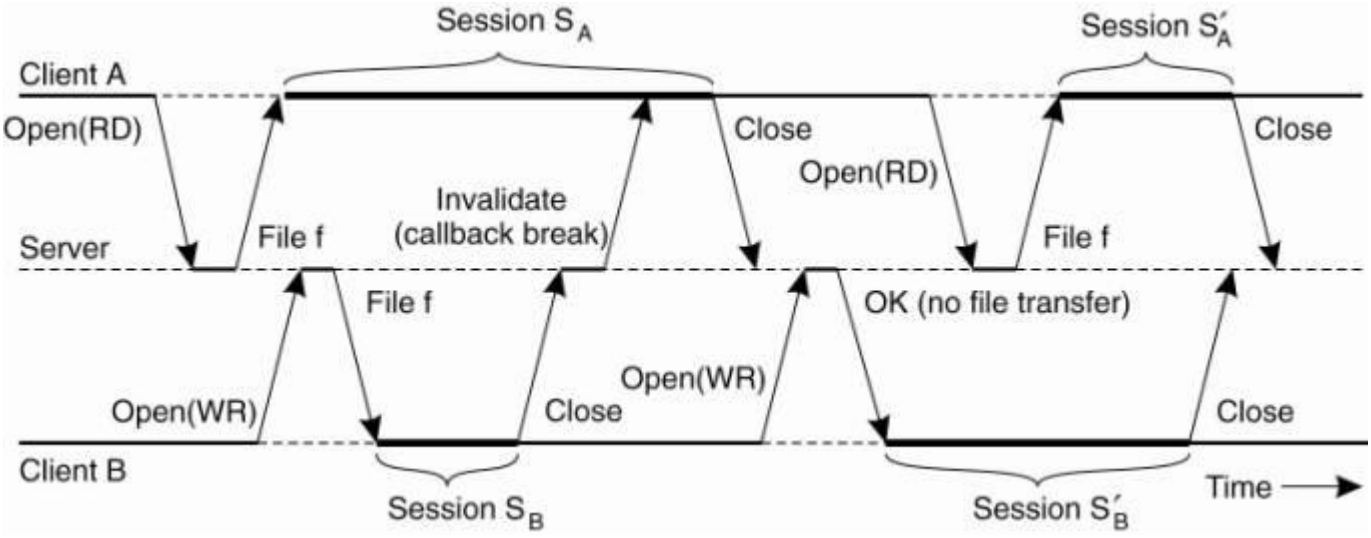
Client-side caching is crucial to the operation of Coda for two reasons. First, caching is done to achieve scalability. Second, caching provides a higher degree of fault tolerance as the client becomes less dependent on the availability of the server. For these two reasons, clients in Coda always cache entire files. In other words, when a file is opened for either reading or writing, an entire copy of the file is transferred to the client, where it is subsequently cached.

Unlike many other distributed file systems, cache coherence in Coda is maintained by means of callbacks. We already came across this phenomenon when discussing file-sharing semantics. For each file, the server from which a client had fetched the file keeps track of which clients have a copy of that file cached locally. A server is said to record a callback promise for a client. When a client updates its local copy of the file for the first time, it notifies the server, which, in turn, sends an invalidation message to the other clients. Such an invalidation message is called a callback break, because the server will then discard the callback promise it held for the client it just sent an invalidation.

The interesting aspect of this scheme is that as long as a client knows it has an outstanding callback promise at the server, it can safely access the file locally. In particular, suppose a client opens a file and finds it is still in its cache. It can then use that file provided the server still has a callback promise on the file for that client. The client will have to check with the server if that promise still holds. If so, there is no need to transfer the file from the server to the client again.

This approach is illustrated in Fig. 11-23, which is an extension of Fig. 11-20. When client A starts session SA, the server records a callback promise. The same happens when B starts session SB. However, when B closes SB, the server breaks its promise to callback client A by sending A a callback break. Note that due to the transactional semantics of Coda, when client A closes session SA, nothing special happens; the closing is simply accepted as one would expect.
[Page 523]

Figure 11-23. The use of local copies when opening a session in Coda.



The consequence is that when A later wants to open session , it will find its local copy of f to be invalid, so that it will have to fetch the latest version from the server. On the other hand, when B opens session , it will notice that the server still has an outstanding callback promise implying that B can simply re-use the local copy it still has from session SB.

Client-Side Caching for Portable Devices

One important development for many distributed systems is that many storage devices can no longer be assumed to be permanently connected to the system through a network. Instead, users have various types of storage devices that are semi-permanently connected, for example, through cradles or docking stations. Typical examples include PDAs, laptop devices, but also portable multimedia devices such as movie and audio players.

In most cases, an explicit upload/download model is used for maintaining files on portable storage devices. Matters can be simplified if the storage device is viewed as part of a distributed file system. In that case, whenever a file needs to be accessed, it may be fetched from the local device or over the connection to the rest of the system. These two cases need to be distinguished.

Tolia et al. (2004) propose to take a very simple approach by storing locally a cryptographic hash of the data contained in files. These hashes are stored on the portable device and used to redirect requests for associated content. For example, when a directory listing is stored locally, instead of storing the data of each listed file, only the computed has is stored. Then, when a file is fetched, the system will first check whether the file is locally available and up-to-date. Note that a stale file will have a different hash than the one stored in the directory listing. If the file is locally available, it can be returned to the client, otherwise a data transfer will need to take place.
[Page 524]

Obviously, when a device is disconnected it will be impossible to transfer any data. Various techniques exist to ensure with high probability that likely to-be-used files are indeed stored locally on the device. Compared to the on-demand data transfer approach inherent to most

caching schemes, in these cases we would need to deploy file-prefetching techniques. However, for many portable storage devices, we can expect that the user will use special programs to pre-install files on the device.

11.6.2. Server-Side Replication

In contrast to client-side caching, server-side replication in distributed file systems is less common. Of course, replication is applied when availability is at stake, but from a performance perspective it makes more sense to deploy caches in which a whole file, or otherwise large parts of it, are made locally available to a client. An important reason why client-side caching is so popular is that practice shows that file sharing is relatively rare. When sharing takes place, it is often only for reading data, in which case caching is an excellent solution.

Another problem with server-side replication for performance is that a combination of a high degree of replication and a low read/write ratio may actually degrade performance. This is easy to understand when realizing that every update operation needs to be carried out at every replica. In other words, for an N-fold replicated file, a single update request will lead to an N-fold increase of update operations. Moreover, concurrent updates need to be synchronized, leading to more communication and further performance reduction.

For these reasons, file servers are generally replicated only for fault tolerance. In the following, we illustrate this type of replication for the Coda file system.

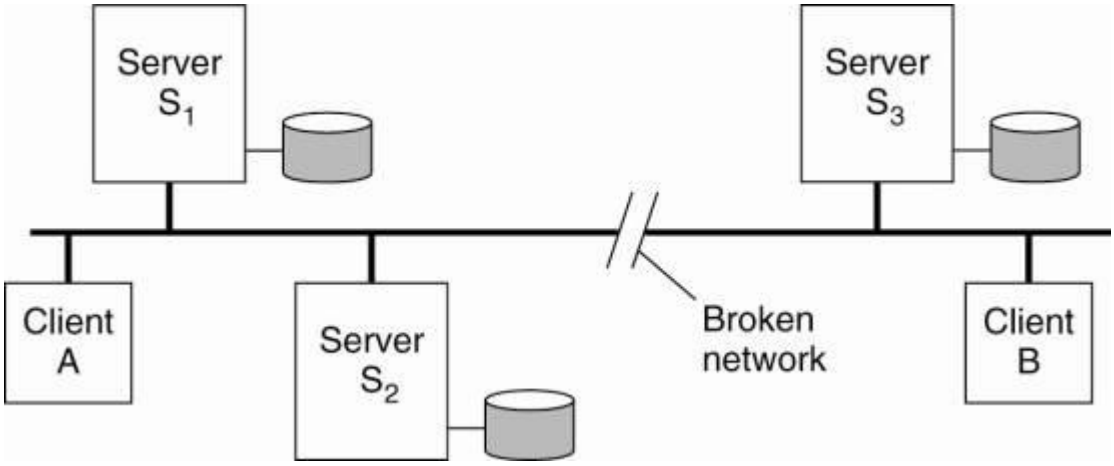
Server Replication in Coda

Coda allows file servers to be replicated. As we mentioned, the unit of replication is a collection of files called a volume. In essence, a volume corresponds to a UNIX disk partition, that is, a traditional file system like the ones directly supported by operating systems, although volumes are generally much smaller. The collection of Coda servers that have a copy of a volume, are known as that volume's Volume Storage Group, or simply VSG. In the presence of failures, a client may not have access to all servers in a volume's VSG. A client's Accessible Volume Storage Group (AVSG) for a volume consists of those servers in that volume's VSG that the client can contact at the moment. If the AVSG is empty, the client is said to be disconnected.
[Page 525]

Coda uses a replicated-write protocol to maintain consistency of a replicated volume. In particular, it uses a variant of Read-One, Write-All (ROWA), which was explained in Chap. 7. When a client needs to read a file, it contacts one of the members in its AVSG of the volume to which that file belongs. However, when closing a session on an updated file, the client transfers it in parallel to each member in the AVSG. This parallel transfer is accomplished by means of MultiRPC as explained before.

This scheme works fine as long as there are no failures, that is, for each client, that client's AVSG of a volume is the same as its VSG. However, in the presence of failures, things may go wrong. Consider a volume that is replicated across three servers S1, S2, and S3. For client A, assume its AVSG covers servers S1 and S2 whereas client B has access only to server S3, as shown in Fig. 11-24.

Figure 11-24. Two clients with a different AVSG for the same replicated file.



Coda uses an optimistic strategy for file replication. In particular, both A and B will be allowed to open a file, f, for writing, update their respective copies, and transfer their copy back to the members in their AVSG. Obviously, there will be different versions of f stored in the VSG. The question is how this inconsistency can be detected and resolved.

The solution adopted by Coda is deploying a versioning scheme. In particular, a server Si in a VSG maintains a Coda version vector CVVi(f) for each file f contained in that VSG. If CVVi(f)[j]=k, thenserver Si knows that server Sj has seen at least version k of file f. CVVi(f)[i] is the number of the current version of f stored at server Si. An update of f at server Si will lead to an increment of CVVi(f)[i]. Note that version vectors are completely analogous to the vector timestamps discussed in Chap. 6.

Returning to our three-server example, CVVi(f) is initially equal to [1,1,1] for each server Si. When client A reads f from one of the servers in its AVSG, say S1, it also receives CVV1(f). After updating f, client A multicasts f to each server in its AVSG, that is, S1 and S2. Both servers will then record that their respective copy has been updated, but not that of S3. In other words,

CVV1(f) = CVV2(f) = [2,2,1]
[Page 526]

Meanwhile, client B will be allowed to open a session in which it receives a copy of f from server S3, and subsequently update f as well. When closing its session and transferring the update to S3, server S3 will update its version vector to CVV3(f)=[1,1,2].

When the partition is healed, the three servers will need to reintegrate their copies of f. By comparing their version vectors, they will notice that a conflict has occurred that needs to be repaired. In many cases, conflict resolution can be automated in an application-dependent way, as discussed in Kumar and Satyanarayanan (1995). However, there are also many cases in which users will have to assist in resolving a conflict manually, especially when different users have changed the same part of the same file in different ways.
11.6.3. Replication in Peer-to-Peer File Systems

Let us now examine replication in peer-to-peer file-sharing systems. Here, replication also plays an important role, notably for speeding up search and look-up requests, but also to balance load between nodes. An important property in these systems is that virtually all files are read only. Updates consist only in the form of adding files to the system. A distinction should be made between unstructured and structured peer-to-peer systems.

Unstructured Peer-to-Peer Systems

Fundamental to unstructured peer-to-peer systems is that looking up data boils down to searching that data in the network. Effectively, this means that a node will simply have to, for example, broadcast a search query to its neighbors, from where the query may be forwarded, and so on. Obviously, searching through broadcasting is generally not a good idea, and special measures need to be taken to avoid performance problems. Searching in peer-to-peer systems is discussed extensively in Risson and Moors (2006).

Independent of the way broadcasting is limited, it should be clear that if files are replicated, searching becomes easier and faster. One extreme is to replicate a file at all nodes, which would imply that searching for any file can be done entirely local. However, given that nodes have a limited capacity, full replication is out of the question. The problem is then to find an optimal replication strategy, where optimality is defined by the number of different nodes that need to process a specific query before a file is found.

Cohen and Shenker (2002) have examined this problem, assuming that file replication can be controlled. In other words, assuming that nodes in an unstructured peer-to-peer system can be instructed to hold copies of files, what is then the best allocation of file copies to nodes?

Let us consider two extremes. One policy is to uniformly distribute n copies of each file across the entire network. This policy ignores that different files may have different request rates, that is, that some files are more popular than others. As an alternative, another policy is to replicate files according to how often they are searched for: the more popular a file is, the more replicas we create and distribute across the overlay.
[Page 527]

As a side remark, note that this last policy may make it very expensive to locate unpopular files. Strange as this may seem, such searches may prove to be increasingly important from an economic point of view. The reasoning is simple: with the Internet allowing fast and easy access to tons of information, exploiting niche markets suddenly becomes attractive. So, if you are interested in getting the right equipment for, let's say a recumbent bicycle, the Internet is the place to go to provided its search facilities will allow you to efficiently discover the appropriate seller.

Quite surprisingly, it turns out that the uniform and the popular policy perform just as good when looking at the average number of nodes that need to be queried. The distribution of queries is the same in both cases, and such that the distribution of documents in the popular policy follows the distribution of queries. Moreover, it turns out that any allocation "between" these two is better. Obtaining such an allocation is doable, but not trivial.

Replication in unstructured peer-to-peer systems happens naturally when users download files from others and subsequently make them available to the community. Controlling these networks is very difficult in practice, except when parts are controlled by a single organization.

Moreover, as indicated by studies conducted on BitTorrent, there is also an important social factor when it comes to replicating files and making them available (Pouwelse et al., 2005). For example, some people show altruistic behavior, or simply continue to make files no longer available than strictly necessary after they have completed their download. The question comes to mind whether systems can be devised that exploit this behavior.

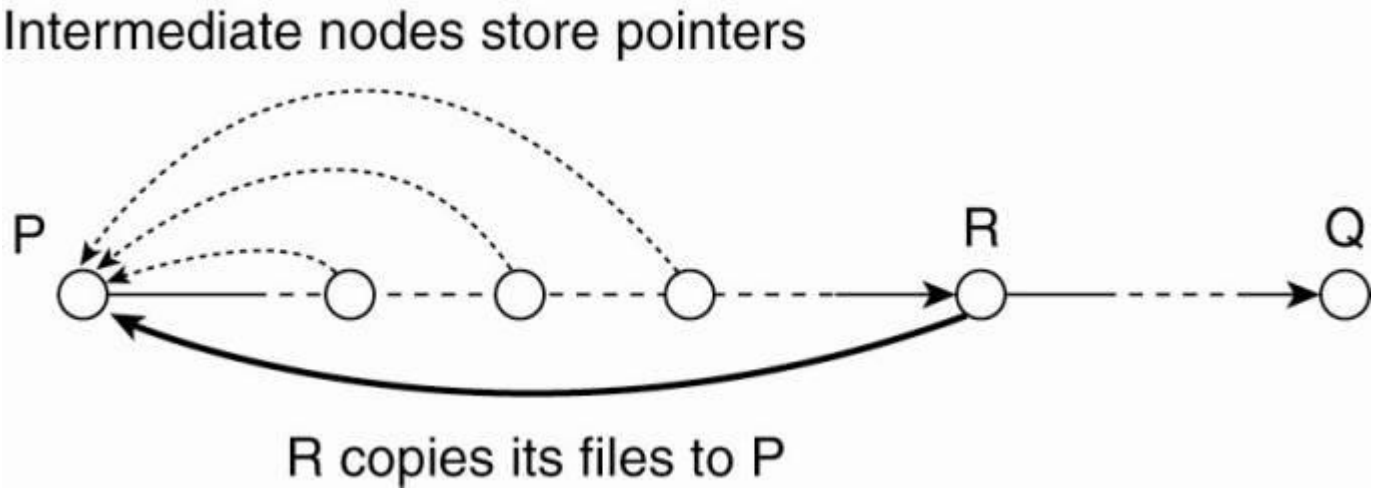
Structured Peer-to-Peer Systems

Considering the efficiency of lookup operations in structured peer-to-peer systems, replication is primarily deployed to balance the load between the nodes. We already encountered in Chap. 5 how a "structured" form of replication, as exploited by Ramasubramanian and Sirer (2004b) could even reduce the average lookup steps to O(1). However, when it comes to load balancing, different approaches need to be explored.

One commonly applied method is to simply replicate a file along the path that a query has followed from source to destination. This replication policy will have the effect that most replicas will be placed close to the node responsible for storing the file, and will thus indeed offload that node when there is a high request rate. However, such a replication policy does not take the load of other nodes into account, and may thus easily lead to an imbalanced system.
[Page 528]

To address these problems, Gopalakrishnan et al. (2004) propose a different scheme that takes the current load of nodes along the query route into account. The principal idea is to store replicas at the source node of a query, and to cache pointers to such replicas in nodes along the query route from source to destination. More specifically, when a query from node P to Q is routed through node R, R will check whether any of its files should be offloaded to P. It does so by simply looking at its own query load. If R is serving too many lookup requests for files it is currently storing in comparison to the load imposed on P, it can ask P to install copies of R's most requested files. This principle is sketched in Fig. 11-25.

Figure 11-25. Balancing load in a peer-to-peer system by replication.



If P can accept file f from R, each node visited on the route from P to R will install a pointer for f to P, indicating that a replica of f can be found at P.

Clearly, disseminating information on where replicas are stored is important for this scheme to work. Therefore, when routing a query through the overlay, a node may also pass on information concerning the replicas it is hosting. This information may then lead to further installment of pointers, allowing nodes to take informed decisions of redirecting requests to nodes that hold a replica of a requested file. These pointers are placed in a limited-size cache and are replaced following a simple least-recently used policy (i.e., cached pointers referring to files that are never asked for, will be removed quickly).

11.6.4. File Replication in Grid Systems

As our last subject concerning replication of files, let us consider what happens in Grid computing. Naturally, performance plays a crucial role in this area as many Grid applications are highly compute-intensive. In addition, we see that applications often also need to process vast amounts of data. As a result, much effort has been put into replicating files to where applications are being executed. The means to do so, however, are (somewhat) surprisingly simple.

A key observation is that in many Grid applications data are read only. Data are often produced from sensors, or from other applications, but rarely updated or otherwise modified after they are produced and stored. As a result, data replication can be applied in abundance, and this is exactly what happens.
[Page 529]

Unfortunately, the size of the data sets are sometimes so enormous that special measures need to be taken to avoid that data providers (i.e., those machine storing data sets) become overloaded due to the amount of data they need to transfer over the network. On the other hand, because much of the data is heavily replicated, balancing the load for retrieving copies is less of an issue.

Replication in Grid systems mainly evolves around the problem of locating the best sources to copy data from. This problem can be solved by special replica location services, very similar to the location services we discussed for naming systems. One obvious approach that has been developed for the Globus toolkit is to use a DHT-based system such as Chord for decentralized lookup of replicas (Cai et al., 2004). In this case, a client passes a file name to any node of the service, where it is converted to a key and subsequently looked up. The information returned to the client contains contact addresses for the requested files.

To keep matters simple, located files are subsequently downloaded from various sites using an FTP-like protocol, after which the client can register its own replicas with the replication location service. This architecture is described in more detail in Chervenak et al. (2005), but the approach is fairly straightforward.

11.7. Fault Tolerance

Fault tolerance in distributed file systems is handled according to the principles we discussed in Chap. 8. As we already mentioned, in many cases, replication is deployed to create fault-

tolerant server groups. In this section, we will therefore concentrate on some special issues in fault tolerance for distributed file systems.

11.7.1. Handling Byzantine Failures

One of the problems that is often ignored when dealing with fault tolerance is that servers may exhibit arbitrary failures. In other words, most systems do not consider the Byzantine failures we discussed in Chap. 8. Besides complexity, the reason for ignoring these type of failures has to do with the strong assumptions that need to be made regarding the execution environment. Notably, it must be assumed that communication delays are bounded.

In practical settings, such an assumption is not realistic. For this reason, Castro and Liskov (2002) have devised a solution for handling Byzantine failures that can also operate in networks such as the Internet. We discuss this protocol here, as it can (and has been) directly applied to distributed file systems, notably an NFS-based system. Of course, there are other applications as well. The basic idea is to deploy active replication by constructing a collection of finite state machines and to have the nonfaulty processes in this collection execute operations in the same order. Assuming that at most k processes fail at once, a client sends an operation to the entire group and accepts an answer that is returned by at least k + 1 different processes.
[Page 530]

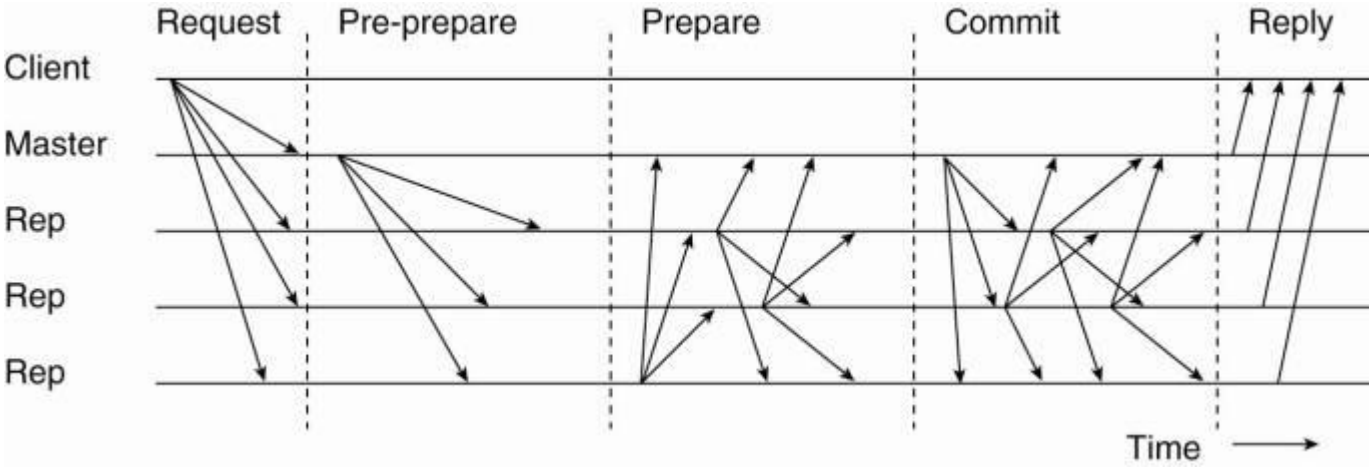
To achieve protection against Byzantine failures, the server group must consist of at least 3k + 1 processes. The difficult part in achieving this protection is to ensure that nonfaulty processes execute all operations in the same order. A simple means to achieve this goal is to assign a coordinator that simply serializes all operations by attaching a sequence number to each request. The problem, of course, is that the coordinator may fail.

It is with failing coordinators that the problems start. Very much like with virtual synchrony, processes go through a series of views, where in each view the members agree on the nonfaulty processes, and initiate a view change when the current master appears to be failing. This latter can be detected if we assume that sequence numbers are handed out one after the other, so that a gap, or a timeout for an operation may indicate that something is wrong. Note that processes may falsely conclude that a new view needs to be installed. However, this will not affect the correctness of the system.

An important part of the protocol relies on the fact that requests can be correctly ordered. To this end, a quorum mechanism is used: whenever a process receives a request to execute operation o with number n in view v, it sends this to all other processes, and waits until it has received a confirmation from at least 2k others that have seen the same request. In this way, we obtain a quorum of size 2k + 1 for the request. Such a confirmation is called a quorum certificate. In essence, it tells us that a sufficiently large number of processes have stored the same request and that it is thus safe to proceed.

The whole protocol consists of five phases, shown in Fig. 11-26.

Figure 11-26. The different phases in Byzantine fault tolerance.



During the first phase, a client sends a request to the entire server group. Once the master has received the request, it multicasts a sequence number in a pre-prepare phase so that the associated operation will be properly ordered. At that point, the slave replicas need to ensure that the master's sequence number is accepted by a quorum, provided that each of them accepts the master's proposal. Therefore, if a slave accepts the proposed sequence number, it multicasts this acceptance to the others. During the commit phase, agreement has been reached and all processes inform each other and execute the operation, after which the client can finally see the result.
[Page 531]

When considering the various phases, it may seem that after the prepare phase, all processes should have agreed on the same ordering of requests. However, this is true only within the same view: if there was a need to change to a new view, different processes may have the same sequence number for different operations, but which were assigned in different views. For this reason, we need the commit phase as well, in which each process now tells the others that it has stored the request in its local log, and for the current view. As a consequence, even if there is a need to recover from a crash, a process will know exactly which sequence number had been assigned, and during which view.

Again, a committed operation can be executed as soon as a nonfaulty process has seen the same 2k commit messages (and they should match its own intentions). Again, we now have a quorum of $2k + 1$ for executing the operation. Of course, pending operations with lower sequence numbers should be executed first.

Changing to a new view essentially follows the view changes for virtual synchrony as described in Chap. 8. In this case, a process needs to send information on the pre-prepared messages that it knows of, as well as the received prepared messages from the previous view. We will skip further details here.

The protocol has been implemented for an NFS-based file system, along with various important optimizations and carefully crafted data structures, of which the details can be found in Castro and Liskov (2002). A description of a wrapper that will allow the incorporation of Byzantine fault tolerance with legacy applications can be found in Castro et al. (2003).

11.7.2. High Availability in Peer-to-Peer Systems

An issue that has received special attention is ensuring availability in peer-to-peer systems. On the one hand, it would seem that by simply replicating files, availability is easy to guarantee. The problem, however, is that the unavailability of nodes is so high that this simple reasoning no longer holds. As we explained in Chap. 8, the key solution to high availability is redundancy. When it comes to files, there are essentially two different methods to realize redundancy: replication and erasure coding.

Erasure coding is a well-known technique by which a file is partitioned into m fragments which are subsequently recoded into n > m fragments. The crucial issue in this coding scheme is that any set of m encoded fragments is sufficient to reconstruct the original file. In this case, the redundancy factor is equal to rec=n/m. Assuming an average node availability of a, and a required file unavailability ofε, we need to guarantee that at least m fragments are available, that is:

$$1 - \varepsilon = \sum_{i=m}^{n} \binom{n}{i} a^i (1-a)^{n-i}$$

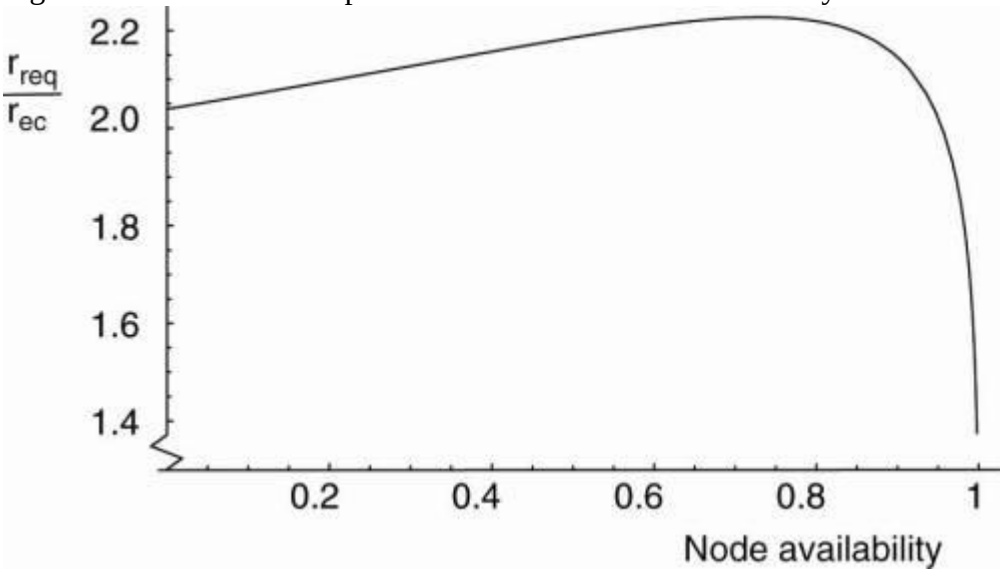
[Page 532]

If we compare this to replicating files, we see that file unavailability is completely dictated by the probability that all its rrep replicas are unavailable. If we assume that node departures are independent and identically distributed, we have

1 - ε = 1 = (1-a)rrep

Applying some algebraic manipulations and approximations, we can express the difference between replication and erasure coding by considering the ratio rrep/rec in its relation to the availability a of nodes. This relation is shown in Fig. 11-27, for which we have set m = 5 [see also Bhagwan et al. (2004) and Rodrigues and Liskov (2005)].

Figure 11-27. The ratio rrep/rec as a function of node availability a.



What we see from this figure is that under all circumstances, erasure coding requires less redundancy than simply replicating files. In other words, replicating files for increasing availability in peer-to-peer networks in which nodes regularly come and go is less efficient from a storage perspective than using erasure coding techniques.

One may argue that these savings in storage are not really an issue anymore as disk capacity is often overwhelming. However, when realizing that maintaining redundancy will impose communication, then lower redundancy is going to save bandwidth usage. This performance gain is notably important when the nodes correspond to user machines connected to the Internet through asymmetric DSL or cable lines, where outgoing links often have a capacity of only a few hundred Kbps.

11.8. Security

Many of the security principles that we discussed in Chap. 9 are directly applied to distributed file systems. Security in distributed file systems organized along a client-server architecture is to have the servers handle authentication and access control. This is a straightforward way of dealing with security, an approach that has been adopted, for example, in systems such as NFS. [Page 533]

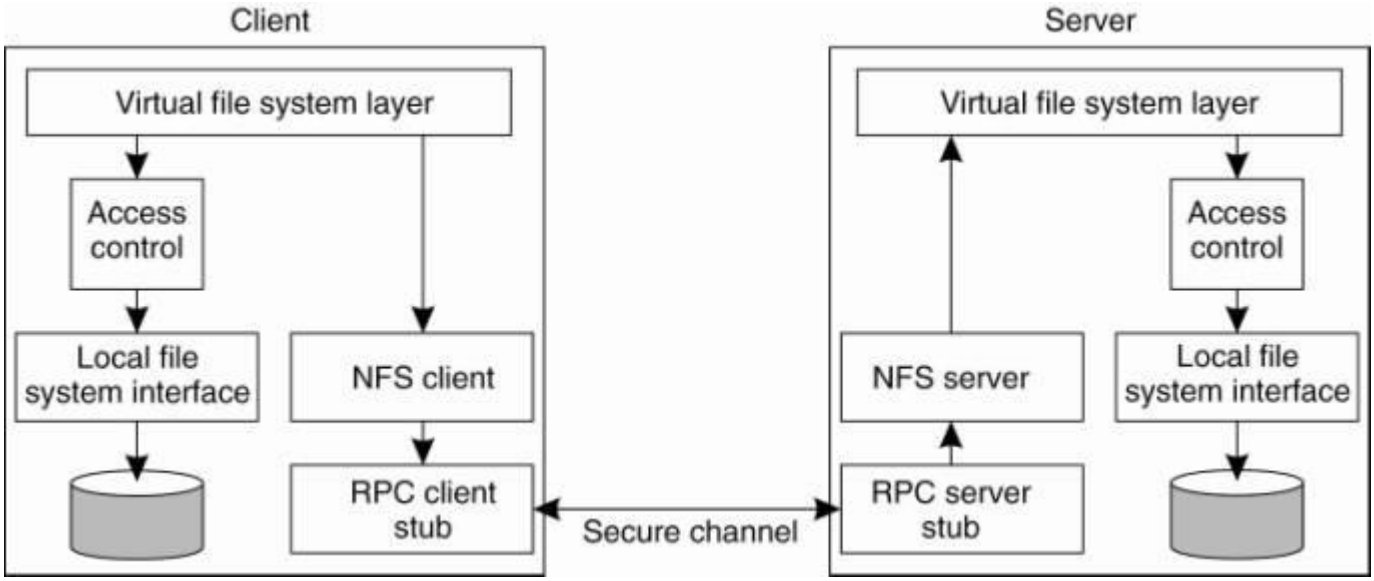
In such cases, it is common to have a separate authentication service, such as Kerberos, while the file servers simply handle authorization. A major drawback of this scheme is that it requires centralized administration of users, which may severely hinder scalability. In the following, we first briefly discuss security in NFS as an example of the traditional approach, after which we pay attention to alternative approaches.

11.8.1. Security in NFS

As we mentioned before, the basic idea behind NFS is that a remote file system should be presented to clients as if it were a local file system. In this light, it should come as no surprise that security in NFS mainly focuses on the communication between a client and a server. Secure communication means that a secure channel between the two should be set up as we discussed in Chap. 9.

In addition to secure RPCs, it is necessary to control file accesses. which are handled by means of access control file attributes in NFS. A file server is in charge of verifying the access rights of its clients, as we will explain below. Combined with secure RPCs, the NFS security architecture is shown in Fig. 11-28.

Figure 11-28. The NFS security architecture.



Secure RPCs

Because NFS is layered on top of an RPC system, setting up a secure channel in NFS boils down to establishing secure RPCs. Up until NFSv4, a secure RPC meant that only authentication was taken care of. There were three ways for doing authentication. We will now examine each one in turn.
[Page 534]

The most widely-used method, one that actually hardly does any authentication, is known as system authentication. In this UNIX-based method, a client simply passes its effective user ID and group ID to the server, along with a list of groups it claims to be a member of. This information is sent to the server as unsigned plaintext. In other words, the server has no way at all of verifying whether the claimed user and group identifiers are actually associated with the sender. In essence, the server assumes that the client has passed a proper login procedure, and that it can trust the client's machine.

The second authentication method in older NFS versions uses Diffie-Hellman key exchange to establish a session key, leading to what is called secure NFS. We explained how Diffie-Hellman key exchange works in Chap. 9. This approach is much better than system authentication, but is more complex, for which reason it is implemented less frequently. Diffie-Hellman can be viewed as a public-key cryptosystem. Initially, there was no way to securely distribute a server's public key, but this was later corrected with the introduction of a secure name service. A point of criticism has always been the use of relatively small public keys, which are only 192 bits in NFS. It has been shown that breaking a Diffie-Hellman system with such short keys is nearly trivial (Lamacchia and Odlyzko, 1991).

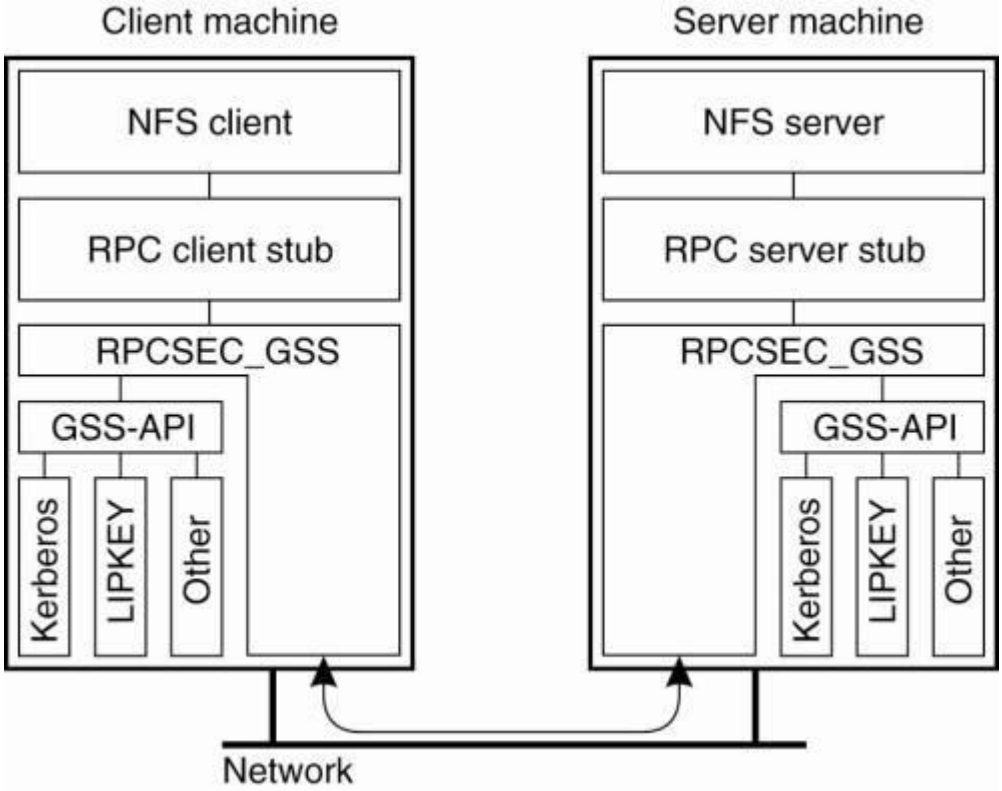
The third authentication protocol is Kerberos, which we also described in Chap. 9.

With the introduction of NFSv4, security is enhanced by the support for RPCSEC_GSS. RPCSEC_GSS is a general security framework that can support a myriad of security mechanism for setting up secure channels (Eisler et al., 1997). In particular, it not only provides

the hooks for different authentication systems, but also supports message integrity and confidentiality, two features that were not supported in older versions of NFS.

RPCSEC_GSS is based on a standard interface for security services, namely GSS-API, which is fully described in Linn (1997). The RPCSEC_GSS is layered on top of this interface, leading to the organization shown in Fig. 11-29.

Figure 11-29. Secure RPC in NFSv4.
(This item is displayed on page 535 in the print version)



For NFSv4, RPCSEC_GSS should be configured with support for Kerberos V5. In addition, the system must also support a method known as LIPKEY, described in Eisler (2000). LIPKEY is a public-key system that allows clients to be authenticated using a password while servers can be authenticated using a public key.

The important aspect of secure RPC in NFS is that the designers have chosen not to provide their own security mechanisms, but only to provide a standard way for handling security. As a consequence, proven security mechanisms, such Kerberos, can be incorporated into an NFS implementation without affecting other parts of the system. Also, if an existing security mechanisms turns out to be flawed (such as in the case of Diffie-Hellman when using small keys), it can easily be replaced.

It should be noted that because RPCSEC_GSS is implemented as part of the RPC layer that underlies the NFS protocols, it can also be used for older versions of NFS. However, this adaptation to the RPC layer became available only with the introduction of NFSv4.

Access Control

Authorization in NFS is analogous to secure RPC: it provides the mechanisms but does not specify any particular policy. Access control is supported by means of the ACL file attribute. This attribute is a list of access control entries, where each entry specifies the access rights for a specific user or group. Many of the operations that NFS distinguishes with respect to access control are relatively straightforward and include those for reading, writing, and executing files, manipulating file attributes, listing directories, and so on.

Noteworthy is also the synchronize operation that essentially tells whether a process that is colocated with a server can directly access a file, bypassing the NFS protocol for improved performance. The NFS model for access control has much richer semantics than most UNIX models. This difference comes from the requirements that NFS should be able to interoperate with Windows systems. The underlying thought is that it is much easier to fit the UNIX model of access control to that of Windows, then the other way around.

Another aspect that makes access control different from file systems such as in UNIX, is that access can be specified for different users and different groups. Traditionally, access to a file is specified for a single user (the owner of the file), a single group of users (e.g., members of a project team), and for everyone else. NFS has many different kinds of users and processes, as shown in Fig. 11-30.

Figure 11-30. The various kinds of users and processes distinguished by NFS with respect to access control.
(This item is displayed on page 536 in the print version)

Type of user	Description
Owner	The owner of a file
Group	The group of users associated with a file
Everyone	Any user or process
Interactive	Any process accessing the file from an interactive terminal
Network	Any process accessing the file via the network
Dialup	Any process accessing the file through a dialup connection to the server
Batch	Any process accessing the file as part of batch job
Anonymous	Anyone accessing the file without authentication
Authenticated	Any authenticated user or process
Service	Any system-defined service process

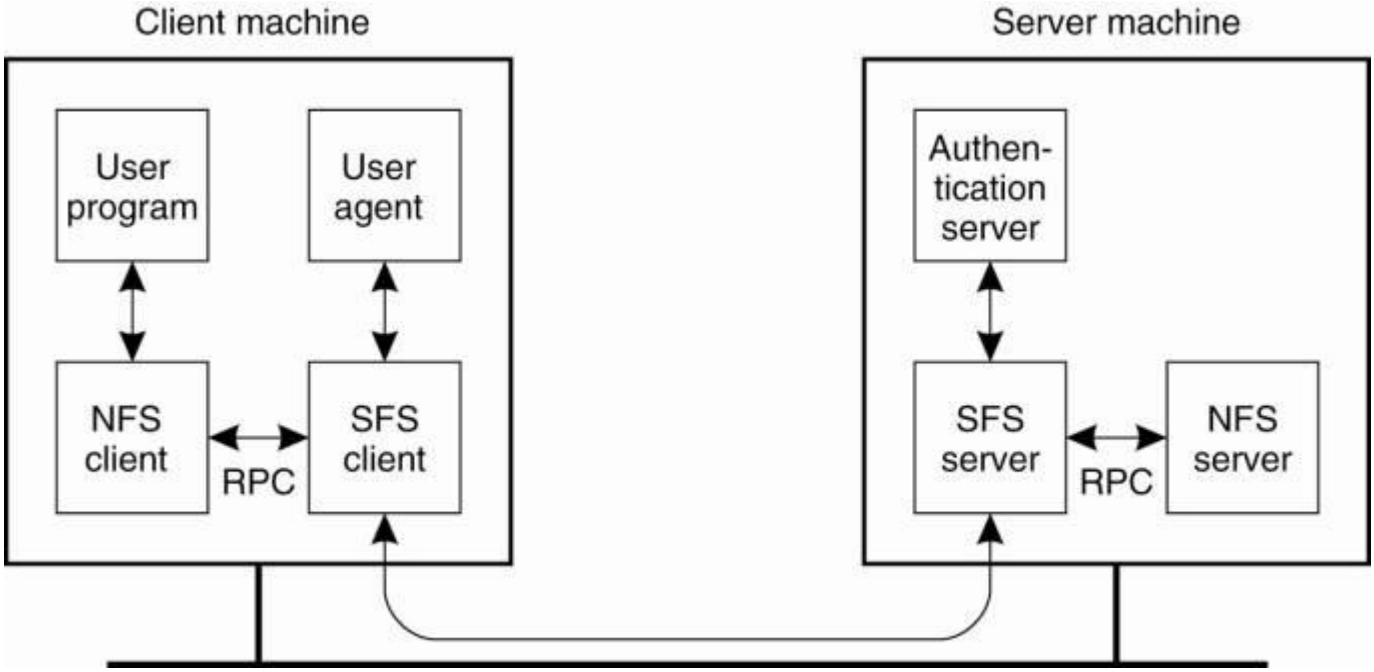
[Page 536]
11.8.2. Decentralized Authentication

One of the main problems with systems such as NFS is that in order to properly handle authentication, it is necessary that users are registered through a central system administration. A solution to this problem is provided by using the Secure File Systems (SFS) in combination with decentralized authentication servers. The basic idea, described in full detail in Kaminsky et al. (2003) is quite simple. What other systems lack is the possibility for a user to specify that a remote user has certain privileges on his files. In virtually all cases, users must be globally known to all authentication servers. A simpler approach would be to let Alice specify that "Bob,

whose details can be found at X," has certain privileges. The authentication server that handles Alice's credentials could then contact server X to get information on Bob.

An important problem to solve is to have Alice's server know for sure it is dealing with Bob's authentication server. This problem can be solved using self-certifying names, a concept introduced in SFS (Mazières et al., 1999) aimed at separating key management from file-system security. The overall organization of SFS is shown in Fig. 11-31. To ensure portability across a wide range of machines, SFS has been integrated with various NFSv3 components. On the client machine, there are three different components, not counting the user's program. The NFS client is used as an interface to user programs, and exchanges information with an SFS client. The latter appears to the NFS client as being just another NFS server.

Figure 11-31. The organization of SFS.
(This item is displayed on page 537 in the print version)

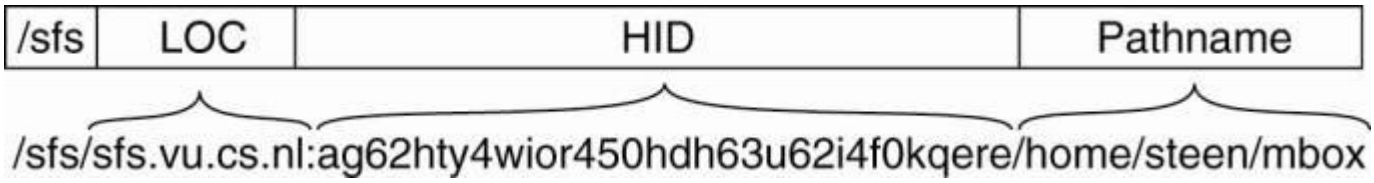


The SFS client is responsible for setting up a secure channel with an SFS server. It is also responsible for communicating with a locally-available SFS user agent, which is a program that automatically handles user authentication. SFS does not prescribe how user authentication should take place. In correspondence with its design goals, SFS separates such matters and uses different agents for different user-authentication protocols.
[Page 537]

On the server side there are also three components. The NFS server is again used for portability reasons. This server communicates with the SFS server which operates as an NFS client to the NFS server. The SFS server forms the core process of SFS. This process is responsible for handling file requests from SFS clients. Analogous to the SFS agent, an SFS server communicates with a separate authentication server to handle user authentication.

What makes SFS unique in comparison to other distributed file systems is its organization of its name space. SFS provides a global name space that is rooted in a directory called /sfs. An SFS client allows its users to create symbolic links within this name space. More importantly, SFS uses self-certifying pathnames to name its files. Such a pathname essentially carries all the information to authenticate the SFS server that is providing the file named by the pathname. A self-certifying pathname consists of three parts, as shown in Fig. 11-32.

Figure 11-32. A self-certifying pathname in SFS.



The first part of the name consists of a location LOC, which is either a DNS domain name identifying the SFS server, or its corresponding IP address. SFS assumes that each server S has a public key . The second part of a self-certifying pathname is a host identifier HID that is computed by taking a cryptographic hash H over the server's location and its public key:
[Page 538]

HID is represented by a 32-digit number in base 32. The third part is formed by the local pathname on the SFS server under which the file is actually stored.

Whenever a client accesses an SFS server, it can authenticate that server by simply asking it for its public key. Using the well-known hash function H, the client can then compute HID and verify it against the value found in the pathname. If the two match, the client knows it is talking to the server bearing the name as found in the location.

How does this approach separate key management from file system security? The problem that SFS solves is that obtaining a server's public key can be completely separated from file system security issues. One approach to getting the server's key is letting a client contact the server and requesting the key as described above. However, it is also possible to locally store a collection of keys, for example by system administrators. In this case, there is no need to contact a server. Instead, when resolving a pathname, the server's key is looked up locally after which the host ID can be verified using the location part of the path name.

To simplify matters, naming transparency can be achieved by using symbolic links. For example, assume a client wants to access a file named

/sfs/sfs.cs.vu.nl:ag62hty4wior450hdh63u62i4f0kqere/home/steen/mbox

To hide the host ID, a user can create a symbolic link

/sfs/vucs ⟶ /sfs/sfs.cs.vu.nl:ag62hty4wior450hdh63u62i4f0kqere

and subsequently use only the pathname /sfs/vucs/home/steen/mbox. Resolution of that name will automatically expand to the full SFS pathname, and using the public key found locally, authenticate the SFS server named sfs.vu.cs.nl.

In a similar fashion, SFS can be supported by certification authorities. Typically, such an authority would maintain links to the SFS servers for which it is acting. As an example, consider an SFS certification authority CA that runs the SFS server named

/sfs/sfs.certsfs.com:kty83pad72qmbna9uefdppioq7053jux

Assuming the client has already installed a symbolic link

/certsfs ⟶ /sfs/sfs.certsfs.com:kty83pad72qmbna9uefdppioq7053jux,

the certification authority could use another symbolic link

/vucs ⟶ /sfs/sfs.vu.cs.nl:ag62hty4wior450hdh63u62i4f0kqere

that points to the SFS server sfs.vu.cs.nl. In this case, a client can simply refer to /certsfs/vucs/home/steen/mbox knowing that it is accessing a file server whose public key has been certified by the certification authority CA.
[Page 539]

Returning to our problem of decentralized authentication, it should now be clear that we have all the mechanisms in place to avoid requiring Bob to be registered at Alice's authentication server. Instead, the latter can simply contact Bob's server provided it is given a name. That name already contains a public key so that Alice's server can verify the identity of Bob's server. After that, Alice's server can accept Bob's privileges as indicated by Alice. As said, the details of this scheme can be found in Kaminsky et al. (2003).

11.8.3. Secure Peer-to-Peer File-Sharing Systems

So far, we have discussed distributed file systems that were relatively easy to secure. Traditional systems either use straightforward authentication and access control mechanisms extended with secure communication, or we can leverage traditional authentication to a completely decentralized scheme. However, matters become complicated when dealing with fully decentralized systems that rely on collaboration, such as in peer-to-peer file-sharing systems.

Secure Lookups in DHT-Based Systems

There are various issues to deal with (Castro et al., 2002a; and Wallach, 2002). Let us consider DHT-based systems. In this case, we need to rely on secure lookup operations, which essentially boil down to a need for secure routing. This means that when a nonfaulty node looks up a key k, its request is indeed forwarded to the node responsible for the data associated with k, or a node storing a copy of that data. Secure routing requires that three issues are dealt with:

1. Nodes are assigned identifiers in a secure way.

2. Routing tables are securely maintained.

3. Lookup requests are securely forwarded between nodes.

When nodes are not securely assigned their identifier, we may face the problem that a malicious node can assign itself an ID so that all lookups for specific keys will be directed to itself, or forwarded along the route that it is part of. This situation becomes more serious when nodes can team up, effectively allowing a group to form a huge "sink" for many lookup requests. Likewise, without secure identifier assignment, a single node may also be able to assign itself many identifiers, also known as a Sybil attack, creating the same effect (Douceur, 2002).

More general than the Sybil attack is an attack by which a malicious node controls so many of a nonfaulty node's neighbors, that it becomes virtually impossible for correct nodes to operate properly. This phenomenon is also known as an eclipse attack, and is analyzed in Singh et al. (2006). Defending yourself against such an attack is difficult. One reasonable solution is to constrain the number of incoming edges for each node. In this way, an attacker can have only a limited number of correct nodes pointing to it. To also prevent an attacker from taking over all incoming links to correct nodes, the number of outgoing links should also be constrained [see also Singh et al. (2004)]. Problematic in all these cases is that a centralized authority is needed for handing out node identifiers. Obviously, such an authority goes against the decentralized nature of peer-to-peer systems.
[Page 540]

When routing tables can be filled in with alternative nodes, as is often the case when optimizing for network proximity, an attacker can easily convince a node to point to malicious nodes. Note that this problem does not occur when there are strong constraints on filling routing table entries, such as in the case of Chord. The solution, therefore, is to mix choosing alternatives with a more constrained filling of tables [of which details are described in Castro et al. (2002a)].

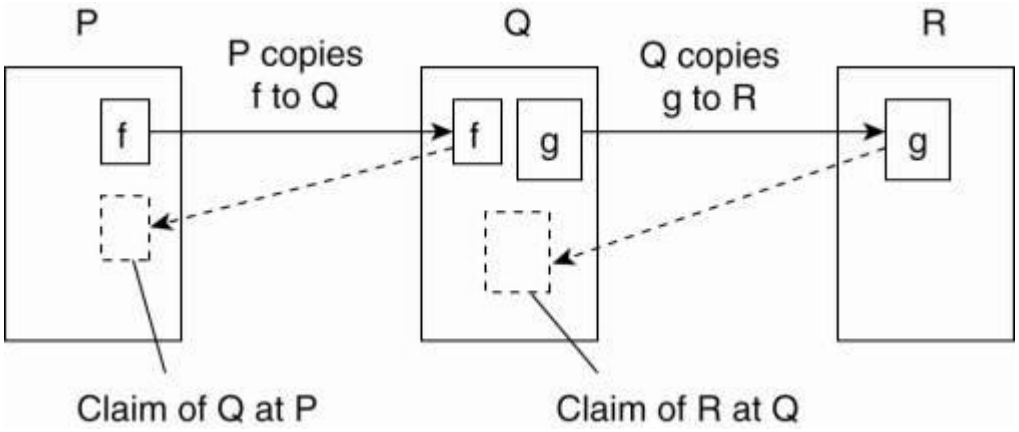
Finally, to defend against message-forwarding attacks, a node may simply forward messages along several routes. One way to do this is to initiate a lookup from different source nodes.

Secure Collaborative Storage

However, the mere fact that nodes are required to collaborate introduces more problems. For example, collaboration may dictate that nodes should offer about the same amount of storage that they use from others. Enforcing this policy can be quite tricky. One solution is to a apply a secure trading of storage, as for Samsara, as described in Cox and Noble (2003).

The idea is quite simple: when a server P wants to store one of its files f on another server Q, it makes storage available of a size equal to that of f, and reserves that space exclusively for Q. In other words, Q now has an outstanding claim at A, as shown in Fig. 11-33.

Figure 11-33. The principle of storage claims in the Samsara peer-to-peer system.



To make this scheme work, each participant reserves an amount of storage and divides that into equal-sized chunks. Each chunk consists of incompressible data. In Samsara, chunk ci consists of a 160-bit hash value hi computed over a secret passphrase W concatenated with the number i. Now assume that claims are handed out in units of 256 bytes. In that case, the first claim is computed by taking the first 12 chunks along with the first 16 bytes of next chunk. These chunks are concatenated and encrypted using a private key K. In general, claim Cj is computed as
[Page 541]

Cj = K(hk,hk+1, . . ., hk+11,hk+12[0], . . . , hk+12[15])

where k=jx13. Whenever P wants to make use of storage at Q, Q returns a collection of claims that P is now forced to store. Of course, Q need never store its own claims. Instead, it can compute when needed.

The trick now is that once in a while, Q may want to check whether P is still storing its claims. If P cannot prove that it is doing so, Q can simply discard P's data. One blunt way of letting P prove it still has the claims is returning copies to Q. Obviously, this will waste a lot of bandwidth. Assume that Q had handed out claims Cj1, . . . , Cjk to P. In that case, Q passes a 160-bit string d to P, and requests it to compute the 160-bit hash d 1 of d concatenated with Cj1. This hash is then to be concatenated with Cj2, producing a hash value d2, and so on. In the end, P need only return dn to prove it still holds all the claims.

Of course, Q may also want to replicate its files to another node, say R. In doing so, it will have to hold claims for R. However, if Q is running out of storage, but has claimed storage at P, it may decide to pass those claims to R instead. This principle works as follows.

Assume that P is holding a claim CQ for Q, and Q is supposed to hold a claim CR for R. Because there is no restriction on what Q can store at P, Q might as well decide to store CR at P. Then, whenever R wants to check whether Q is still holding its claim, Q will pass a value d to Q and request it to compute the hash of d concatenated with CR. To do so, Q simply passes d on to P, requests P to compute the hash, and returns the result to R. If it turns out that P is no longer holding the claim, Q will be punished by R, and Q, in turn, can punish P by removing stored data.