CHAPTER 9

DISTRIBUTED FILE SYSTEMS

9.1 INTRODUCTION

A distributed file system is a resource management component of a distributed operating system. It implements a common file system that can be shared by all the autonomous computers in the system. Two important goals of distributed file systems follow.

Network transparency: The primary goal of a distributed file system is to provide the same functional capabilities to access files distributed over a network as the file system of a timesharing mainframe system does to access files residing at one location. Ideally, users do not have to be aware of the location of files to access them. This property of a distributed file system is known as network transparency.

High availability: Another major goal of distributed file systems is to provide high availability. Users should have the same easy access to files, irrespective of their physical location. System failures or regularly scheduled activities such as backups or maintenance should not result in the unavailability of files.

In recent years, several distributed file systems have been developed. In this chapter, we discuss the common mechanisms and design aspects shared by today's distributed file systems. Section 9.5 discusses the implementation of these distributed file systems. First, we describe the architecture of a typical distributed file system.

Scanned with CamScanner

9.2 ARCHITECTURE

Ideally, in a distributed file system, files can be stored at any machine (or computer) and the computation can be performed at any machine (i.e., the machines are peers). When a machine needs to access a file stored on a remote machine, the remote machine performs the necessary file access operations and returns data if a read operation is performed. However, for higher performance, several machines, referred to as *file servers*, are dedicated to storing files and performing storage and retrieval operations. The rest of the machines in the system can be used solely for computational purposes. These machines are referred to as *clients* and they access the files stored on servers (see Fig. 9.1). Some client machines may also be equipped with a local disk storage that can be used for caching remote files, as a swap area, or as a storage area.

The two most important services present in a distributed file system are the *name server* and *cache manager*. A name server is a process that maps names specified by clients to stored objects such as files and directories. The mapping (also referred to as name resolution) occurs when a process references a file or directory for the first time. A cache manager is a process that implements file caching. In file caching, a copy of data stored at a remote file server is brought to the client's machine when referenced by the client. Subsequent accesses to the data are performed locally at the client, thereby reducing the access delays due to network latency. Cache managers can be present at both clients and file servers. Cache managers at the servers cache files in the main memory to reduce delays due to disk latency. If multiple clients are allowed to cache a file and modify it, the copies can become inconsistent. To avoid this inconsistency problem, cache managers at both servers and clients coordinate to perform data storage

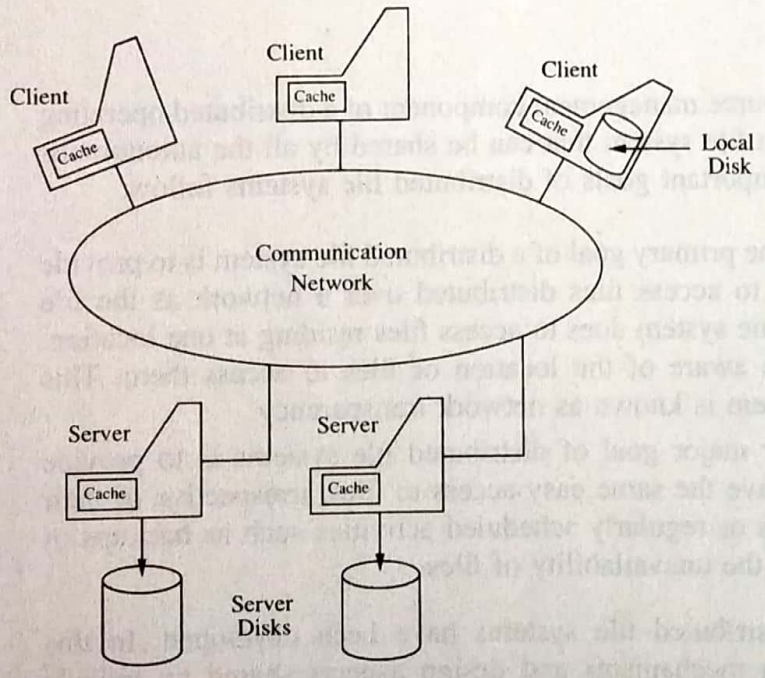

FIGURE 9.1
Architecture of a distributed file system.

Scanned with CamScanner

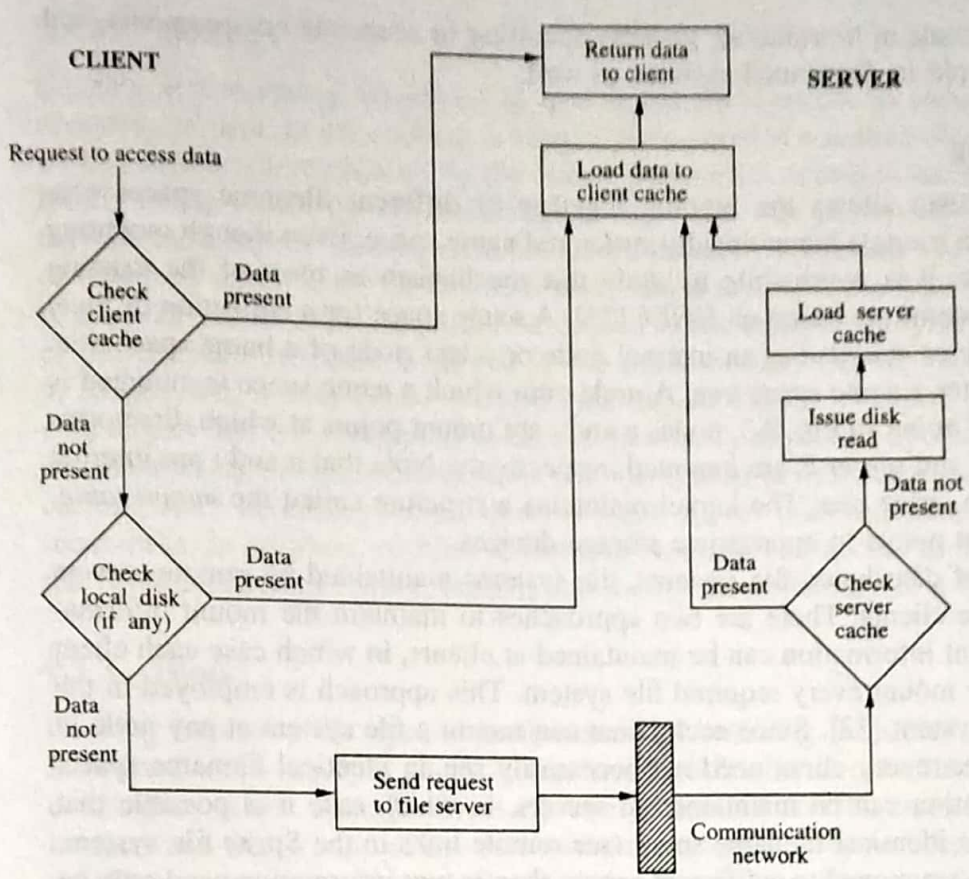

FIGURE 9.2
Typical data access actions in distributed file systems.

and retrieval operations. Typically, data access in a distributed file system proceeds as shown in Fig. 9.2.

A request by a process to access a data block is presented to the local cache (client cache) of the machine (client) on which the process is running (see Fig. 9.1). If the block is not in the cache, then the local disk, if present, is checked for the presence of the data block. If the block is present, then the request is satisfied and the block is loaded into the client cache. If the block is not stored locally, then the request is passed on to the appropriate file server (as determined by the name server). The server checks its own cache for the presence of the data block before issuing a disk I/O request. The data block is transferred to the client cache in any case and loaded to the server cache if it was missing in the server cache.

9.3 MECHANISMS FOR BUILDING DISTRIBUTED FILE SYSTEMS

In this section, the basic mechanisms underlying the majority of the distributed file systems operating today are presented [33]. These mechanisms take advantage of the observations made in previous studies on file systems. We cite these observations along with the mechanisms that exploit them. A crucial point to note here is that these ob-

Scanned with CamScanner

servations were made in timesharing systems operating in academic environments, and are assumed to hold in distributed systems as well.

9.3.1 Mounting

A mount mechanism allows the binding together of different filename spaces (see Sec. 9.4.1) to form a single hierarchically structured name space. Even though mounting is UNIX† specific, it is worthwhile to study this mechanism as most of the existing distributed file systems are based on UNIX [33]. A name space (or a collection of files) can be *bounded to* or *mounted at* an internal node or a leaf node of a name space tree. Figure 9.3 illustrates a name space tree. A node onto which a name space is mounted is known as a *mount point*. In Fig. 9.3, nodes a and i are mount points at which directories stored at server Y and server Z are mounted,,respectively. Note that a and i are internal nodes in the name space tree. The kernel maintains a structure called the *mount table*, which maps mount points to appropriate storage devices.

In the case of distributed file systems, file systems maintained by remote servers are mounted at the clients. There are two approaches to maintain the mount information [33]: (1) Mount information can be maintained at clients, in which case each client has to individually mount every required file system. This approach is employed in the Sun network file system [32]. Since each client can mount a file system at any node in the name space tree, every client need not necessarily see an identical filename space. (2) Mount information can be maintained at servers, in which case it is possible that every client sees an identical filename space (see remote links in the Sprite file system, Sec. 9.5.2.). If files are moved to a different server, then mount information need only be updated at the servers. In the first approach, every client needs to update its mount table.

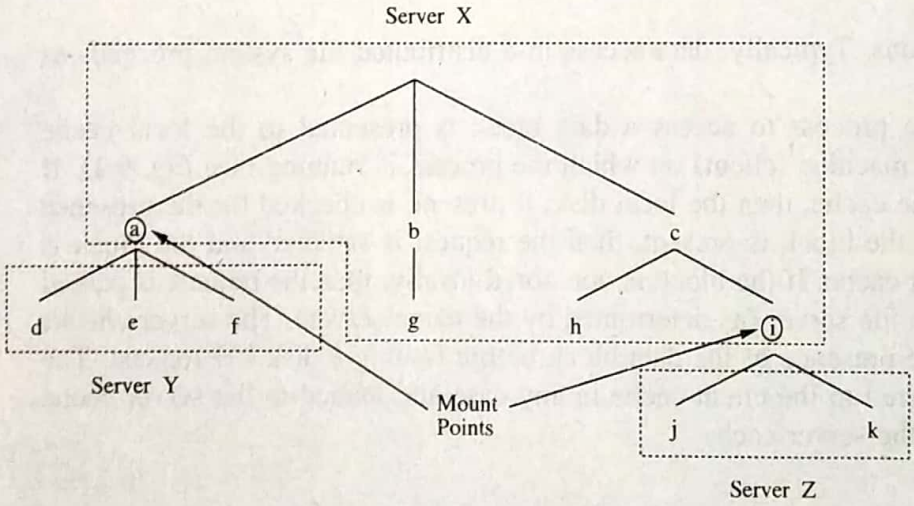

FIGURE 9.3
Name space hierarchy.

†UNIX is a trademark of Novell, Inc.

Scanned with CamScanner

9.3.2 Caching

Caching is commonly employed in distributed file systems to reduce delays in the accessing of data. In file caching, a copy of data stored at a remote file server is brought to the client when referenced by the client. Subsequent access to the data is performed locally at the client, thereby reducing access delays due to network latency. Caching exploits the temporal locality of reference exhibited by programs. The temporal locality of reference refers to the fact that a file recently accessed is likely to be accessed again in the near future. Data can either be cached in the main memory or on the local disk of the clients. Also, data is cached in the main memory (server cache) at the servers to reduce disk access latency. These data include blocks swapped out at clients and data blocks that are contiguous to blocks previously requested by clients, as these data are more likely to be accessed soon. The file system performance can be improved by caching since accessing remote disks is much slower than accessing local memory or local disks. In addition, caching reduces the frequency of access to the file servers and the communication network, thereby improving the scalability of a file system.

9.3.3 Hints

Caching improves file system performance by reducing the delay in accessing data. However, when multiple clients cache and modify shared data, the problem of cache consistency arises. Specifically, it must be guaranteed that the cached data is valid (up-to-date) and that a copy of the data—recently updated in some other client cache or in the file server—does not exist. Guaranteeing consistency is expensive in distributed file systems as it requires elaborate cooperation between file servers and clients.

An alternative approach is to treat the cached data as hints [16, 42]. In this case, cached data are not expected to be completely accurate. However, valid cache entries improve performance substantially without incurring the cost of maintaining cache consistency. The class of applications that can utilize hints are those which can recover after discovering that the cached data are invalid. For example, after the name of a file or a directory is mapped to the physical object, the address of the object can be stored as a hint in the cache. If the address fails to map to the object in the following attempt, the cached address is purged from the cache. The file server consults the name server to determine the actual location of the file or directory and updates the cache.

9.3.4 Bulk Data Transfer

The bulk of the delay in transferring data over a network is due to the high cost of executing communication protocols (such as the assembly and disassembly of packets, the copying of buffers between layers of the communication protocols, etc.). In fact, actual transit time across a local area network can be insignificant. Transferring data in bulk reduces the protocol processing overhead at both servers and clients [33]. In bulk data transfer, multiple consecutive data blocks are transferred from servers to clients instead of just the block referenced by clients.

While caching amortizes the high cost of accessing remote servers over several local references to the same information, bulk transfer amortizes the protocol processing

Scanned with CamScanner

overhead and disk seek time over many consecutive blocks of a file. Bulk transfers reduce file access overhead through obtaining a multiple number of blocks with a single seek; by formatting and transmitting a multiple number of large packets in a single context switch; and by reducing the number of acknowledgments that need to be sent. Bulk transfers exploit the fact that most files are accessed in their entirety [33].

9.3.5 Encryption

Encryption is used for enforcing security in distributed systems [33]. The work of Needham and Schroeder [23] is the basis for most of the current security mechanisms in distributed systems (see Sec. 15.8). In their scheme, two entities wishing to communicate with each other establish a key for conversation with the help of an authentication server. It is important to note that the conversation key is determined by the authentication server, but is never sent in plain (unencrypted) text to either of the entities.

9.4 DESIGN ISSUES

We now discuss various issues that must be addressed in the design and implementation of distributed file systems. By studying these design issues, one can better understand the intricacies of a distributed file system.

9.4.1 Naming and Name Resolution

A name in file systems is associated with an object (such as a file or a directory). *Name resolution* refers to the process of mapping a name to an object or, in the case of replication, to multiple objects. A *name space* is a collection of names which may or may not share an identical resolution mechanism.

Traditionally, there have been three approaches to name files in a distributed environment [29]. The simplest scheme is to concatenate the host name to the names of files that are stored on that host. While this approach guarantees that a filename is unique systemwide, it conflicts with the goal of network transparency. Another serious problem with this approach is that moving a file from one host to another requires changes in the filename and in the applications accessing that file. That is, this naming scheme is not *location-independent*. (If a naming scheme is location-independent, the name of a file need not be changed when the file's physical storage location changes [18].) The main advantage of this scheme, however, is that name resolution is very simple as a file can be located without consulting any other host in the system.

The second approach is to mount remote directories onto local directories. (See Sec. 9.3.1 for details.) Mounting a remote directory requires that the host of the directory be known. Once a remote directory is mounted, its files can be referenced in a *location-transparent* manner. (A naming scheme is said to be location-transparent if the name of a file does not reveal any hint as to its physical storage location [18].) This approach can also resolve a filename without consulting any host.

The third approach is to have a single global directory where all the files in the system belong to a single name space. Variations of this scheme are found in the Sprite

Scanned with CamScanner

and Apollo systems (see Sec. 9.5). This approach does not have the disadvantages of the above two naming schemes. The main disadvantage of this scheme, however, is that it is mostly limited to one computing facility or to a few cooperating computing facilities. This limitation is due to the requirement of systemwide unique filenames, which requires that all the computing facilities involved cooperate [6]. Thus, this scheme is impractical for distributed systems that encompass heterogeneous environments and wide geographical areas, where a naming scheme suitable for one computing facility may be unsuitable for another.

THE CONCEPT OF CONTEXTS. To overcome the difficulties associated with systemwide unique names, the notion of *context* has been used to partition a name space. A context identifies the name space in which to resolve a given name. Contexts can partition a name space along the following: geographical boundary, organizational boundary, specific to hosts, a file system type, etc. In a context based scheme, a filename can be thought of as composed of a context and a name local to that context. Resolving a name involves interpreting the name with respect to the given context. The interpretation may be complete within the given context or may lead to yet another context, in which case the above process is repeated. If all files share a common initial context, then unique systemwide global names result.

The x-Kernel logical file system (see Sec. 9.5.5) is a file system that makes use of contexts. In this file system, a user defines his own file space hierarchy. The internal nodes in this hierarchy correspond to the contexts.

The *Tilde* naming scheme is another variant of the naming scheme using contexts [6]. In the Tilde naming scheme, the name space is partitioned (based on projects which people are associated with) into a set of logically independent directory trees called *tilde trees*. Each process running in the system has a set of tilde trees associated with it that constitute the process's tilde environment. When a process tries to open or manipulate a file, the filename is interpreted with respect to the process's tilde environment.

NAME SERVER. In a centralized system, name resolution can be accomplished by maintaining a table that maps names to objects. In distributed systems, *name servers* are responsible for name resolution. A name server is a process that maps names specified by clients to stored objects such as files and directories. The easiest approach to name resolution in distributed systems is for all clients to send their queries to a single name server which maps names to objects. This approach has the following serious drawbacks: first, if the name server crashes, the entire system is drastically affected. Second, the name server may become a bottleneck and seriously degrade the performance of the system.

The second approach involves having several name servers (on different hosts) wherein each server is responsible for mapping objects stored in different domains. This approach is commonly used in the distributed file systems operating today. When a name (usually with many components such as "a/b/c") is to be mapped to an object, the local name server (such as a table maintained in the kernel) is queried. The local name server may point to a remote name server for further mapping of the name. For example, querying /a/b/c may require a remote server mapping the /b/c part of

Scanned with CamScanner

the filename. This procedure is repeated until the name is completely resolved. By replicating the tables used by name servers, one can achieve fault tolerance and higher performance.

9.4.2 Caches on Disk or Main Memory

The benefits obtained by employing file caches at clients were discussed in Sec. 9.3.2. This section is concerned with the question of whether the data cached by a client should be in the main memory at the client or on a local disk at the client. The advantages of having the cache in the main memory are as follows [24]:

- Diskless workstations can also take advantage of caching. (Note that diskless workstations are cheaper.)
- Accessing a cache in main memory is much faster than accessing a cache on local disk.
- The server-cache is in the main memory at the server, and hence a single design for a caching mechanism is applicable to both clients and servers.

The main disadvantage of having client-cache in main memory is that it competes with the virtual memory system for physical memory space. Thus, a scheme to deal with the memory contention between cache and virtual memory system is necessary. This scheme should also prevent data blocks from being present in both the virtual memory and the cache. A consequence of this fact is a more complex cache manager and memory management system. A limitation of caching in main memory is that large files cannot be cached completely in main memory, thus requiring the caching to be block-oriented. Block-oriented caching is more complex and imposes more load at the file servers (see Bulk Data Transfer) relative to entire file caching.

The advantages of caching on a local disk are: large files can be cached without affecting a workstation's performance; the virtual memory management is simple; and it facilitates the incorporation of portable workstations into a distributed system (see Coda, Section 9.5.4). A workstation, before being disconnected from the network for portable use, will cache all the required files onto its local disk.

9.4.3 Writing Policy

The writing policy decides when a modified cache block at a client should be transferred to the server. The simplest policy is *write-through*. In write-through, all writes requested by the applications at clients are also carried out at the servers immediately. The main advantage of write-through is reliability. In the event of a client crash, little information is lost. A write-through policy, however, does not take advantage of the cache.

An alternate writing policy, *delayed writing policy*, delays the writing at the server [24]. In this case, modifications due to a write are reflected at the server after some delay. This approach can potentially take advantage of the cache by performing many writes on a block present locally in the cache. Another motivation for delaying the writes is that some of the data (for example, intermediate results) could be deleted

Scanned with CamScanner

in a short time, in which case data need not be written at the server at all. In fact, it has been reported that twenty to thirty percent of new data is deleted within thirty seconds [26]. One factor that needs to be taken into account when deciding the length of the delay period is the likelihood of a block not being modified after a given period. While the delayed writing policy takes advantage of the cache at a client, it introduces the reliability problem. In the event of a client crash, a significant amount of data can be lost.

Another writing policy delays the updating of the files at the server until the file is closed at the client. In this policy, the traffic at the server depends on the average period that files are open. If the average period for which files are open is short, then this policy does not greatly benefit from delaying the updates. On the other hand, if the average period for which files are open is long, this policy is also susceptible to losing data in the event of a client crash. Note that it has been reported that a majority of the files are open for a very short time [26].

9.4.4 Cache Consistency

The problem of cache consistency was introduced in Sec. 9.3.3. This section is concerned with the schemes that can guarantee consistency of the data cached at clients. There are two approaches to guarantee that the data returned to the clients is valid [42].

- In the *server-initiated* approach, servers inform cache managers whenever the data in the client caches become stale. Cache managers at clients can then retrieve the new data or invalidate the blocks containing the old data in their cache.
- In the *client-initiated* approach, it is the responsibility of the cache managers at the clients to validate data with the server before returning it to the clients.

Both of these approaches are expensive and unattractive as they require elaborate cooperation between servers and cache managers. In both approaches, communication costs are high. The server-initiated approach requires the server to maintain reliable records on what data blocks are cached by which cache managers. The client-initiated approach simply negates the benefit of having a cache by checking the server to validate data on every access. It also does not scale well, as the load at the server caused by client checking increases with the increase in the number of clients.

A third approach for cache consistency is simply not to allow file caching when *concurrent-write sharing* occurs. In concurrent-write sharing, a file is open at multiple clients and at least one client has it open for writing [24]. In this approach, the file server has to keep track of the clients sharing a file. When concurrent-write sharing occurs for a file, the file server informs all the clients to purge their cached data items belonging to that file. Alternatively, concurrent-write sharing can be avoided by locking files (see the Apollo file system, Sec. 9.5.3).

Another issue that a cache consistency scheme needs to address is *sequential-write sharing*, which occurs when a client opens a file that has recently been modified and closed by another client [24]. Two potential problems with sequential-write sharing are: (1) when a client opens a file, it may have outdated blocks of the file in its cache, and

Scanned with CamScanner

(2) when a client opens a file, the current data blocks may still be in another client's cache waiting to be flushed. This can happen when the delayed writing policy is used.

To handle the first problem, files usually have timestamps associated with them. When data blocks of a file are cached, the timestamp associated with the file is also cached. An inconsistency can be detected by comparing the timestamp of the cached data block with the timestamp of the file at the server.

To handle the second problem, the server must require that clients flush the modified blocks of a file from their cache whenever a new client opens the file for writing.

9.4.5 Availability

Availability is one of the important issues in the design of distributed file systems. The failure of servers or the communication network can severely affect the availability of files. *Replication* is the primary mechanism used for enhancing the availability of files in distributed file systems.

REPLICATION. Under replication, many copies or replicas of files are maintained at different servers. Replication is inherently expensive because of the extra storage space required to store the replicas and the overhead incurred in maintaining all the replicas up to date. The most serious problems with replication are (1) how to keep the replicas of a file consistent and (2) how to detect inconsistencies among replicas of a file and subsequently recover from these inconsistencies. Some typical situations that cause inconsistency among replicas are (a) a replica is not updated due to the failure of the server storing the replica and (b) all the file servers storing the replicas of a file are not reachable from all the clients due to network partition, and the replicas of a file in different partitions are updated differently. *Ironically, potential inconsistency problems may preclude file updates, thereby decreasing the availability as the level of replication is increased.*

UNIT OF REPLICATION. A fundamental design issue in replication is the *unit* of replication. The most basic unit is a *file*. File is the most commonly used replication unit and has been used in the Roe [9], Sprite [25], and Cedar [40] file systems. While this unit allows the replication of only those files that need to have higher availability, it makes overall replica management harder. For example, the protection rights associated with a directory have to be individually stored with each replica; replicas of files belonging to a common directory may not have common file servers and hence require extra name resolutions to locate the replicas in the case of modifications to the directory or the file.

Alternatively, the replication unit can be a group of all the files of a single user or the files that are in a server, etc. The group of files is referred to as a *volume* [38]. This scheme is used in Coda (see Sec. 9.5.4). The main advantage of volume replication is that replica management is easier. Protection rights can be associated with the volume instead of with each individual file replica. However, volume replication may be wasteful as a user typically needs higher availability for only a few files in the volume.

A compromise between volume replication and single file replication, used in Locus [43], captures the advantages of the above two schemes. In this scheme, all the

Scanned with CamScanner

files of a user constitute a filegroup called a *primary pack*. A replica of a primary pack, called a pack, is allowed to contain a subset of the files in the primary pack. With this arrangement, a different degree of replication for each file in the primary pack can be obtained by creating one or more packs of the primary pack.

REPLICA MANAGEMENT. Replica management is concerned with the maintenance of replicas and in making use of them to provide increased availability. Replica management depends on whether consistency is guaranteed by the distributed file system. Here we are concerned with the consistency among replicas only (which is also known as *mutual consistency*), and not with the consistency within a file. The consistency within a file was discussed in Secs. 9.3.3 and 9.4.4.

To ensure mutual consistency among replicas, a weighted voting scheme can be used. We explain this scheme only briefly here as it is discussed in detail in Sec. 13.6. In this scheme, some number of votes and a timestamp are associated with each replica. A certain number of votes r or w must be obtained before a read or write, respectively, can be performed. Only votes from current (i.e., up-to-date) copies are valid. Reads can be from any current copy and writes update all the current copies. Timestamps of all the participating replicas (i.e., only current copies) are updated when a copy is updated. By keeping $w > r$ and $r + w >$ 'total number of votes' of all the replicas, it is possible to maintain at least one current copy. An important point to observe is that it is not necessary to keep all replicas up-to-date as long as sufficient votes can be obtained to perform reads and writes. This key feature provides for increased availability and fault tolerance during system failures. Voting is used in the Roe file system [9] to maintain mutual consistency.

Another scheme to maintain consistency among replicas is to designate one or more processes as agents for controlling the access to replicas of files. This approach has been used in Locus [43]. In Locus, each filegroup has a designated site that enforces the global synchronization policy. This designated site is referred to as the *current synchronization site* (CSS). The file open and file close requests are routed through the CSS to a storage site which has the copy of the requested file. A disadvantage of this approach is that the agent processes can potentially become bottlenecks; hence it has poor scalability.

In the Harp file system [19], the designated site (server) for controlling the access to replicas is referred to as primary, and the other sites (servers) are referred to as backups. The primary enforces the global synchronization policy to maintain consistency in consultation with the backups.

In the Coda file system, mutual consistency among replicas is not assured. For details on its replica management, see Sec. 9.5.4.

9.4.6 Scalability

The issue of scalability deals with the suitability of the design of a system to cater to the demands of a growing system. Currently, client-server organization is a commonly used approach to structure distributed file systems (see Case Studies, Sec. 9.5). Caching, which reduces network latency and server-load, is the primary technique used in client-

Scanned with CamScanner

server organization to improve the client response time. Caching, however, introduces the cache consistency problem, as many clients can cache a file.

Server-initiated cache invalidation is the most commonly used approach to maintain cache consistency. In this scheme, a server keeps track of all the clients sharing files stored on the server. This information forms a part of the server state. As the system grows larger, both the size of the server state and the load due to invalidations increase.

The server state and the server load can be reduced by exploiting knowledge about the usage of files [4, 34]. An important observation in this regard is that many widely used and shared files are accessed in read-only mode. Note that there is no need to check the validity (stale or up-to-date) of these files or to maintain the list of clients at servers for invalidation purposes.

Another observation is that the data required by a client is often found in another client's cache [4]. Since clients have more free cycles compared to servers [34], a client can obtain required data from another client rather than from a server. This of course raises the question of how to find the client which has cached the required data.

Blaze and Alonso [3] have proposed a scheme wherein a server serves (providing data and invalidating in case of updates) only Δ number of clients for a file at any time. New clients after the first Δ clients are informed of the Δ clients from whom they can obtain data. These Δ clients will also serve Δ number of clients, after which the new clients are informed of the identity of the Δ clients they served, and so on. The hierarchy of who is serving who forms a tree of maximum degree Δ. Cache misses and invalidation messages propagate up-and-down in this hierarchy where each internal node serves as a mini-file server for its children.

The structure of the server process also plays a major role in deciding how many clients a server can support. If the server is designed with a single process, then many clients have to wait for a long time whenever a disk I/O is initiated. These waits can be avoided if a separate process is assigned to each client. In this case, however, significant overhead due to the frequent context switches to handle requests from different clients can slow down the server. Lightweight processes (threads) have been proposed to reduce the context switch overhead. Threads are discussed in greater detail in Sec. 17.4.

9.4.7 Semantics

The semantics of a file system characterizes the effects of accesses on files. The basic semantics easily understood and easy to handle by programmers is that a read operation will return the data (stored) due to the latest write operation.

Guaranteeing the above semantics in distributed file systems, which employ caching, is difficult and expensive. Consider a file system employing server-initiated cache invalidation for the guarantee of cache consistency. In such a system, because of communication delays, invalidations may not occur immediately after updates and before reads occur at clients. To guarantee the above semantics, all the reads and writes from various clients will have to go through the server, or sharing will have to be disallowed either by the server, or by the use of locks by applications. Observe that in the first approach, the server can potentially become a bottleneck and the overheads are high because of

Scanned with CamScanner