Chapter 8

Distributed File Systems

File systems aim to provide a clean interface to users to access files on their local system. In addition, a file system also provides access-control mechanisms to allow only authorized uses have access to the files and perform operations on files. For the former, the standard UNIX file system uses a data structure based on *inodes* and a hierarchy of direct and indirect addressing to store the contents of the files and related metadata. For the latter, the Unix file system uses the popular hierarchy of user, group, and others, and read/write/execute as permissions on files. For more details, we refer the reader to Bach [9].

Let us now consider recent planet-scale applications such as Facebook and YouTube. These applications allow multiple users to create, share, and access content simultaneously. It is estimated that each day, Facebook generates new data of the order of petabytes [1]. Similarly, YouTube adds content of the order of exabytes per day (https://www.wyzowl.com/youtube-stats/). Such applications, viz. Google Photos, TikTok, and Amazon are becoming more common in the present era. As can be noticed, the scale of the data that these systems have to store is tremendously large. Further, unlike the earlier decades of computing, today, most of the data and the users of the data are more likely to be geographically separated yet seamlessly connected via the Internet. For the organizations that own this data, this data is also a source of analytics leading to revenue generation.

However, multiple challenges have to be overcome to support both the user-driven functionality and the organization-driven requirements. Some of them include designing highly scalable systems for storing and retrieving the data, addressing the issues of latency and other network-induced phe-

[1] https://kinsta.com/blog/facebook-statistics/

nomenon, providing mechanisms and guarantees for concurrent data access by multiple users, and ensuring the availability and the durability of data.

The changing application requirements necessitate a fundamental rethink on file systems. Traditional file systems optimize for parameters such as the block size, extensibility of the file size and supporting such seamless extension (see also Question 8.1), read and in-place write operation, and access control mechanisms. This conventional wisdom is now up for a revision due to multiple factors including the decreasing cost of disk drive per GB of storage, reducing latency for each operation, minimizing the number of disk accesses needed for any operation, less or no reliance on in-place write, versioning control, and the like.

The rethink on the conventional wisdom mentioned in the earlier paragraph require the design and implementation of distributed file systems that go beyond the notion of traditional file systems (such as the Unix file system, ext3, Windows NTFS) and are optimized based on the new and emerging application requirements.

In this chapter, we first review some of the common design features of distributed file systems. Subsequently, we provide details of some of the popular distributed file systems starting from NFS [112] to Colossus [64]. We also explain in brief the file systems that other platforms, Haystack [16] and Tectonic [101], that Facebook uses. The detail we provide covers some of the design choices and assumptions, operations, and some implementation details. For simplicity, we do not distinguish heavily between file systems and object stores. While both satisfy a similar set of requirements, the latter are seen to be more application-specific. Being application-specific allows such systems to introduce optimizations driven by the target application.

8.1 Common Design Features

The origin of distributed file systems can be traced to projects such as the NFS [112] from Sun Microsystems and the Andrews File System (AFS) [65] of the Carnegie Mellon University. The basic idea of these distributed file systems is to allow the users to use remote computers also to access and store files as if they are stored locally. Some of the common design features of distributed file systems are as follows.

- **Durability:** Data durability is a key concern for any distributed file system. This refers to the idea that the data can be recovered in case of failures to the underlying hardware. As distributed file systems use multiple hardware resources so as to provide scalability, hardware errors due to device malfunction may corrupt data. Distributed file systems usually employ a form of replication to safeguard against such issues. Replication however comes with a set of challenges involving the creation/ updation/ maintenance/ and consistency of the replicas. Distributed file systems have to therefore understand how to provide support for these mechanisms and their implication on the file system usage.

- **Availability:** Distributed file systems have to contend with possible errors from the users or the file system itself. These errors are not the result of poor system design and implementation but a byproduct of the intended use cases and scale. Distributed file systems cater to clients that are spread over the network and naturally, the requests are received and served over the network. Delays and other network induced vagaries affect the file system and its operations. Similarly, as the file system scales to beyond petascale and zeta scale storage, the number of components that play a role in the entire system increases. This increase will invariably result in an increase in the number of failures. Hence, distributed file systems have to provide for error handling mechanisms and the necessary semantics to recover from errors.

- **Consistency Model:** Distributed file systems should naturally support concurrent operations on files and simultaneous sharing of files by users. This requires the distributed file system to provide mechanisms that allow such concurrent operations while providing reasonable guarantees on the outcome of the concurrent operations. This suggests that distributed file systems have to specify and operationalize the semantics of the concurrent operations.

- **Scalability:** One of the advantages of using a distributed file system is to provide for scale as the storage requirements grow. To this end, distributed file systems should be designed in a way that allows for scaling up the capacity in a seamless and transparent manner without degrading performance.

- **Namespace:** The amount of metadata and the data structures that a file system maintains is dictated by how the system handles the file and directory namespace. The mechanism that the filesystem uses to search for files and update the metadata is based on the data structures that the file system sets up. Further, memory system considerations such as where the metadata is stored, the cost of accessing/updating the metadata, costs associated with keeping the metadata on stable storage, and the like influence the organization of the namespace.

Some distributed file systems use a flat namespace to simplify handling of metadata. This decision is usually a result of aiming for lower latency and higher throughput.

- **Block Size:** Blocks are the unit of storage of files in a file system. The size of a block has a bearing on the size of the metadata that the file system has to keep. In addition, block size also influences the network traffic in a distributed setting, internal fragmentation, cache size and behavior, among other things. Unix file systems typically use a block size of 0.5 KB to 4 KB.

  Depending on the application that distributed file systems target, some use a very large block size of several MB. This allows such file systems to scale to large files while not increasing the amount of metadata. Observe that a small block size prevents internal fragmentation at the expense of more metadata, while a large block size can result in internal fragmentation but reduces metadata.

Beyond the ones mentioned above, there are application-driven concerns such as optimizing for bandwidth, latency, number of files, typical size of files, and the like. The functionality, semantics, system assumptions and support vary across distributed file systems as we will see below.

In this chapter, we first set out to understand the challenges and design space of distributed file systems with respect to the above mentioned properties. This exercise will help us understand the choices that system-designers have in arriving at designs of distributed file systems. Subsequently, we will see the choices made by some of the proposed distributed file systems over the decades.

8.2 The Design Space of Distributed File Systems

One of the fundamental properties of distributed file systems over a traditional file system is to provide support for multiple users to access a set of files over a network, subject to access control. Figure 8.1 shows the typical system abstraction at a very high level.

Consider a typical use case of the system depicted in Figure 8.1. A client wants to access a file and has to contact the server for this purpose. The client has to supply information that allows the server to locate the file that

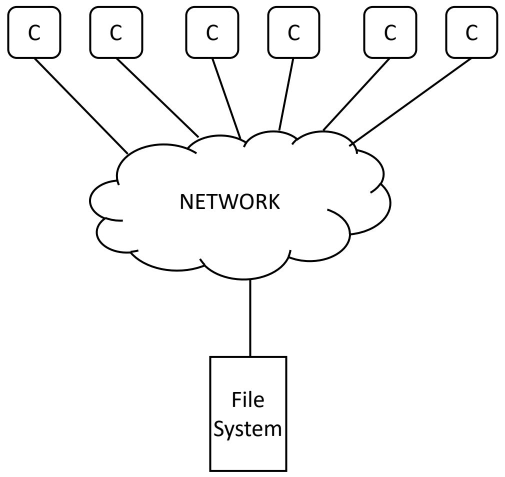

Figure 8.1: Figure illustrating the simplified abstract view of a distributed file system. The entities written as C indicates a client.

the client needs. This is usually specified as a path name if the files are organized in a hierarchical manner. Modern object based storage systems choose to use a flat namespace and each object is given a unique identifier. In this case, the server locates the object with the help of additional data structures.

Once the access model is set up, we now understand the choices in how the client performs other operations such as read/write on the file. There are multiple choices and we detail some of the choices. One choice is for the server to get information from the client on the part of the file that the client wants to access. The other choice is for the server to transfer the file contents, partially/in-full, to the client and let the client work on a local copy of the remote file. In the former case, the system resembles a client-server system with the server being responsible for managing the file system and the contents of the files. This solution is possible when one considers situations where the network bandwidth and latency considerations allow for the server to play a major role in every client operation. In the latter case, the server allows the clients to cache the file in part or in full. The client can then perform operations on the file as if it is a local file. At the end of these operations, the client has to inform the server if the operations change the contents of the file or the metadata related to the file. This model works best in settings where the network bandwidth and throughput are low. After the initial transfer of the file contents, client operations on the file behave similar to operations on a local file.

Maintaining caches of files being used at the client side however introduces a level of indirection[2]. Even as caches improve locality, and in this setting reduce network traffic, they introduce yet another problem that is particular to any distributed system in general.

In a simple setting, subsequent to making any updates to the local copy of the file, when the file is to be closed, the client can contact the server to send the updated file contents. The server will make the changes to its (original) copy and send an acknowledgement of the same. Often, network bandwidth limitations may hamper the throughput of the file system and can introduce delays in when the server can acknowledge the completion of a file update at the server end. The client has to wait for this acknowledgement before moving to the next instruction. To reduce this delay, the file system can instead send an immediate acknowledgement of the file update request from the client, pool multiple write operations, and update the file contents once for the entire pool. This model is called as the *asynchronous update* operation. While this improves on the latency of the update, it poses dangers with respect to loss of update if the server crashes without applying the update. To address this situation, one solution is for the client to treat the acknowledgement from the server as a weak guarantee, keep the data in its local buffers, poll the server for the status of the update. The client can release the local copy only after it knows that the update is applied successfully at the server.

Even with caching, the above is indeed a very simple view. Consider that a distributed file system should naturally allow for multiple clients to share files. In this situation, multiple clients will likely access the same file and perform operations on the files concurrently. In this direction, imagine that two clients access the same file `foo.txt`. When the server allows for parts or the entire file to be cached at a client side, each of the clients may be caching a common portion of the file. Each client is unaware that there may be other clients having the same (portion of the) file also in their cache. Now, if both the clients write to common location of the file, each client will be unaware of the changes made by the other client. Figure 8.2 shows depicts the situation.

The situation demonstrated in Figure 8.2 is symptomatic of concurrent writes to any shared object. As an example, suppose that the file that two clients access concurrently corresponds to a source file in a large project.

[2]See the quote attributed to David Wheeler, "All problems in Computer Science can be solved by another level of indirection"

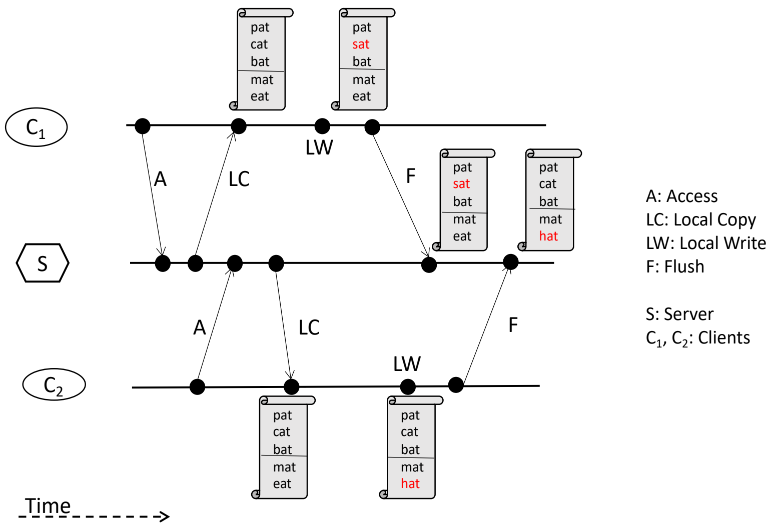

Figure 8.2: Figure illustrating a possible clash between writes by multiple clients.

If these writes overlap, it is not clear as to which one would eventually correspond to the final copy and how to reconcile the differences. Think of systems like `git` and the way they deal with such concurrent updates. In order to distributed file systems have to formalize the semantics of what happens in such concurrent writes. Notice that concurrent reads pose less of a problem compared to concurrent writes.

Surprisingly, there are several ways to formalize the semantics of concurrent updates to a file in a distributed system. Depending on the application need, there exists a spectrum of design choices in this case. The semantics are often called the consistency model of the distributed file system. At one end of the spectrum, a distributed file system may simply forbid sharing of files across multiple clients at the same time. Since there is no concurrent access to files, the server is effectively able to serialize all the updates to the files – in the order the files are accessed. This is very limiting and goes against the grain of a distributed file system. Another possible solution is for the server to allow concurrent access to files but make no guarantees on the outcome of the concurrent write. Referring to the earlier example, the source file may have an inconsistent state after the two writes are affected. This solution is not appealing but has no overhead at the server.

Fortunately, better solutions exist. The server can implement a policy often termed as the “Last Write Wins”. In this case, the contents of the file (in full or partially) that were the last to be acknowledged as successful by the server will be the contents that will be served to later clients. REWORD. This consistency model is similar to a serialized model of updates. (Question: Identify the serialization point for every write). There are other consistency models also such as weak consistency, eventual consistency, strong consistency, that are also possible from a system standpoint. In addition, depending on the nature of applications that the filesystem targets to support, the system may provide better consistency semantics for specific operations.

So far, we have worked under the assumption that the file updates coming from clients will also come with an offset specified. The POSIX style write operation has the file offset as one of the input parameters. On the other hand, in particular with emerging application scenarios, what is more important is to have light-weight guarantees on adding the data to the file whereas the offset where the data is written may not be as important. Examples include log files where usually the timestamp of the log entry help assign a timeline to the data and from a user point of view, having the log entry in the file is more important. In line with this observation, a distributed file system can choose to target specific operations and provide stronger consistency guarantees on those operations. The operations can be chosen by the file system based on the applications that the file system targets.

Faults:

The earlier paragraphs did not cover any discussion on faults. Distributed systems in general have to also consider aspects of fault tolerance due to the their nature. Failures of the network, storage hardware, or client/server machines have the potential to introduce multiple challenges. For instance, if the distributed file system has a single server, crashes of the server or instances when the server is unavailable due to network issues, introduces difficulties with respect to the *availability* of the overall system. In a single-server system, a server crash also renders stateful server based models open to loss of state. This requires systems to provision for mechanisms for server state recovery and addressing the system availability in the interim.

Such mechanisms include a combination of techniques such as provisioning for a shadow server, supporting only for a subset of operations in the interim, committing server state and updates to the server state to durable storage before acknowledgments, checkpointing, having multiple servers, and so on. It must be noted that some of these mechanisms have been found useful in other distributed system designs also.

Faults of the storage hardware in the context of distributed file systems have the potential to create loss of user data. A standard mechanism to handle this is to create replicas of the data. In a solution that uses replication, the system divides each file into logical pieces and replicates each piece by a chosen factor. This protects against data loss and corruption provided all replicas are maintained as identical copies. Doing so however requires additional overheads when the file contents are being updated. For every file update, for the corresponding piece, each replica has to be updated. Notice that each replica may be on a different physical device and these update operations may also fail while some of these succeed. This creates further complications that systems have to solve. Systems have a choice between targeting higher throughput and allow for inconsistencies in the file contents versus aiming for stronger consistency guarantees and possibly affecting throughput.

Another fault that arises in the distributed file system is about network faults that may potentially induce partitions in the set of clients and the server(s). In this situation, a server may be temporarily unable to reach some set of clients or vice-versa. Such a situation may result in stale information at server or the client while messages from one do not reach the other.

In the above, we discussed multiple use cases and challenges around a distributed file system. Other design considerations surround the data structure(s) that the file system maintains to store metadata about the files. Recall that the Unix file system has an elaborate three level data structure consisting of a file descriptor table, the file table, the per file inode table, that promote a great deal of flexibility in how the Unix operating system handles files and directories. These data structures allow the Unix file system to maintain a hierarchical namespace.

Even as the hierarchical namespace works well for generic file systems, the benefit of the hierarchical organization is often offset by other application-specific requirements. For instance, when the system has to only deal with specific kinds of files, such as images or videos, certain optimizations are possible. For instance, a set of such objects can be stored in a single file in the file system with offsets indicating the position of each object in the combined file. Similarly, it may not be worthwhile to support write operations on image or video data. However, changes to image or video can be instead supported by a versioning mechanism that stores the full content for each change. Doing so also reduces the time needed for a server to serve any version of the file instead of preparing the required version through the change log. Notice that the two possibilities depicted here present a tradeoff in terms of the amount of storage needed versus the latency of the access operation at the server.

A related aspect that goes along with namespace considerations is about the design choice of what metadata the system keeps about files. The traditional Unix file system keeps information such as access control and access timestamps as part of the metadata. As application specific distributed file systems cater to specific file classes, they do not need efficient support for particular operations. In such cases, it is possible to forgo some parts of the metadata to gain efficiency even while losing out on certain other operations. The specific pieces of metadata that the system can skip may be guided by the needs of the application.

The concerns in those decades corresponded to making efficient use of the available storage space and limited throughput of the disk drives. Of late, the cost of storage drives has plummeted to AAA per GB of storage.

8.2.1 Sample Realizations

In the above section, we have described multiple possible choices for each of the design aspects of distributed file systems. Now, we will provide quick examples of how some of the extant distributed file systems fare in this design space and the choices they make.

A simple example is the SUN NFS [112] system which acts mostly as a facilitator for clients to use a remote file system. It uses a single server and does not provide any replication on its own. It also is stateless and hence provides no guarantees on how the updates are visible to other clients. It however allows clients to keep blocks of the file in their local cache and push updates to the local cache before sending the updates to the server. The Andrew File System (AFS) [65] is very similar to NFS except allowing for the clients to keep the entire file as a local copy.

The post-Internet era distributed file systems such as the Google File Systems [52], Colossus [64], and Tectonic [101] are examples of highly scalable file systems that use replication, provides for availability, guarantee a notion of consistency, and so on. An example of an application specific distributed file system is the Facebook Haystack [16] object store that trades-offs metadata for faster access to the files.

More details of the above systems are presented in the sequel of this chapter.

8.3 Network File System (NFS)

NFS [112] started as a project at the Sun Microsystems in 1984. As with any file system, NFS too supports typical operations on files such as open/ read/ write/ close, and seek, and on directories such as create file/ make directory/ rename file/ rename directory/ delete file, and delete directory. The idea is to create a consistent namespace for files irrespective of the location of the file, whether the file is on a local machine or on a remote machine.

The first version of NFS, NFS v1.0, is an in-house project. The second version, NFS v2.0, is what is released for public use.

8.3.1 Goals and Assumptions

NFS aims to provide a machine and system independent interface and transparent access to the remote file system. With the popularity of Unix at that time, NFS adopts a Unix style interface to its calls and semantics. In addition, NFS underscores the fact that distributed systems are prone to failures, and hence the file system should be able to recover from server and client crash failures, and network failures.

NFS assumes that most file operations will be to read and there are fewer write operations. This assumption is validated by the statistics of that era collected by Sun Microsystems [112, Appendix 2]. In addition, NFS assumes that there is little sharing of files across users and the predominant use case is concerned with users accessing files stored at a remote server. NFS therefore aims to optimize its performance on single user operations. This design model corresponds to user-oriented design. Early versions of NFS did not support locking of remote files too – an operation that would be required more in a setting where multiple users share a file.

8.3.2 Operation and Semantics

NFS uses Remote Procedure Calls (RPC) to support remote file operations. There are two main components in the implementation: a server and a client. A server program runs on a machine that stores and serves files, and a client program runs on the user's machine. NFS relies on network calls and transfers each client request to the server. Figure 8.3 shows the basic architectural diagram of NFS.

The protocol is stateless, synchronous, and blocking. The server program does not store any details of past accesses of clients. So, every client request

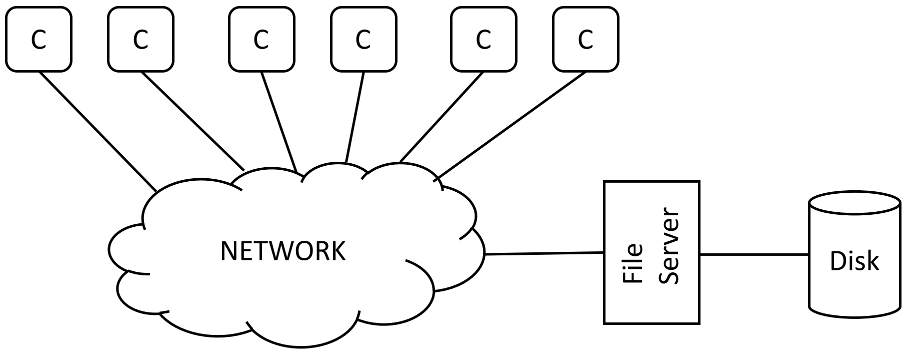

Figure 8.3: Figure illustrating the overall design of NFS. The entities written as C indicates a client.

has to include all the parameter values needed to complete the request. The client program waits until the response is obtained from the server. Having a stateless server model simplifies some of the fault recovery issues such as (i) a server crash handled by the client by a retry, and (ii) a client crash need not be handled by the server.

NFS used the TCP/IP suite of protocols for its network communication and uses the UDP protocol with its concomitant possibility of packet loss. However, this is not a limitation since the protocol is stateless. Any packet losses will be handled by retry.

To maintain the statelessness of the server, the server has to ensure that all modifications to the file state are committed to a stable storage before returning results to the client. In addition, as the server is stateless, there is no provision to support file locking of remote files. Such lock information would require the server to keep track of some state information. A fallout of the lack of support for locking is in the lack of consistency guarantees offered by NFS on concurrent write operations. Two clients that issue a file write request to the same file can see inconsistent results.

In the simple design that is apparent from the above discussion, it appears that every client request has to necessarily reach the server for completion. This naturally limits the performance that NFS can achieve. To address this problem, NFS uses caching at the client side and the server side. This caching allows the client to cache blocks of files and make modifications at the client side instead of sending these as a request to the server. The client can then push all the updates when the client is done using the file. This is termed as `flush-on-close` semantics. Many file operations tend to also query attribute information on flies, such as the `stat` command in Unix. These operations do not modify the contents of the file. Hence, NFS keep a cache of the file attribute data on the client for open files. This step also reduces the need to contact the NFS server for every file operation. Time limits are used to refresh the attribute data so as to keep such data current. NFS server also maintains caches of the inodes, buffers, and directory lookup, so as to gain performance on these operations.

The above indicates the trade-offs between performance and consistency issues introduced by caching. Two clients that read the same file at the same time may be getting different results based on what these clients have in their caches.

Using stateless protocols also necessitates another common practice that is used by most stateless protocols in distributed computing. All operations are idempotent in nature. This is ensured by having, for instance, unique identifiers on files (and directories). If a client wants to delete a file named "test" with an identifier of "0x18370399", then the delete operation is issued as `delete(0x18370399)` and not as `delete(test)`. (see Question 8.3 at the end of the chapter on why the latter is not idempotent and potentially risky.)

Design Features of NFS 2.0

NFS essentially acts as an interface between an existing file system and a client accessing the file system via a network. Therefore, NFS also does not have any in-built mechanisms to address the scalability issue. For the same reason, NFS does not have to make any choices with respect to features such as the namespace and the block size. These will be the corresponding mechanisms that the underlying file system uses.

Ensuring durability also is left to the underlying file system. If, for instance, the underlying file system uses techniques such as RAID, then the benefits of such techniques carry over.

With respect to availability, notice that NFS adopts the stateless model so that failures are handled by retries. So, NFS does not by itself have any mechanisms to deal with failures and improving availability.

NFS also assumes that most files are not used in a shared mode. Therefore, it does not provide any guarantees on the outcome of concurrent read and write operations. As NFS allows clients to cache data, it is possible that two clients that both cache a file may be getting different results on a read operation to the same offset depending on what is cached at each client.

8.3.3 NFS v3 and NFS v4

In the above, we largely discussed NFS v2.0. Later versions of NFS, in particular, NFS v3.0 (cf. IETF RFC 1813) and NFS v4.0 (cf. IETF RFC 7530), added more features on top of NFS v2.0. We briefly mention those here.

The NFS v3.0 server is still stateless, but is designed to use fewer network operations, achieves a faster write, and aims to reduce server load. A particularly new operation in NFS v3.0 is the *asynchronous write* operation. In the semantics of asynchronous write operation, the client sends the write data to the server and the server acknowledges the receipt of the data. However, the server need not write the data to stable storage before sending the reply. At the server end, asynchronous writes provide the server with several options so as to determine the best policy to synchronize the data. The server may never schedule the write or may wait for multiple writes to be performed together. The latter option inherently supports better buffering and parallelism. The client issuing an asynchronous write operation keeps the copy of the data in its cache. When the client wishes to release the data from its cache, it polls the server with a COMMIT message to check if the write is completed. In this case, the server sends a positive reply only if the data under question is written to stable storage and sends an error otherwise. The client, on receiving an error message, can resend the data asynchronously.

NFS v3.0 also added a rudimentary locking feature of remote files via an ancillary protocol called the Network Lock Manager (NLM). Calls to lock/unlock files are prepared as RPC calls to NLM. Unlike NFS v3, notice that NLM has to be a stateful protocol since the information on which process has locks to a particular file is to be maintained. The NLM protocol also has particular mechanisms to address a network partitioning, a client crash and a server crash.

NFS v4.0 is a stateful protocol unlike the previous versions of NFS. In addition, NFS v4.0 introduces three key functionalities: Compound operations, leases, and locking. Compound operations allow a for combining a sequence of operations to a single remote operation thereby reducing network traffic. In this case, atomicity of the entire set of operations is not guaranteed. On the first error that the server encounters in a compound operation, the server does not execute the remaining operations in the compound operation and replies with the results of the successful operations.

Finally, NFS v4 allows for clients to lock remote files and this support is part of the protocol instead of delegating this support to auxiliary protocols such as Network Lock Manager (NLM) that NFS v3.0 does. The server in NFS v4.0 being stateful can keep track of the lock information on files. Locking is coupled with leasing where the server grants a lease of a certain duration to the client. The duration of the lease is set at installation time and is usually of the order of few tens of seconds (60s to 90s is a fairly commonly used value). The leases are automatically renewed as the client performs any active operation on the files. If the server experiences a crash and comes back up, the NFS v4 server requires clients to reestablish locks within a given time window. During this time window, service is interrupted.

Design Features of NFS v3.0 and NFS v4.0

NFS v3.0 and v4.0 still essential acts as an interface between an existing file system and a client accessing the file system remotely via a network. Therefore, these versions of NFS also does not have any in-built mechanisms to address the scalability issue. For the same reason, these versions of NFS does not have to make any choices with respect to features such as the namespace and the block size. These will be the corresponding mechanisms that the underlying file system uses.

In NFS v3.0 and v4.0, ensuring durability also is left to the underlying file system. If, for instance, the underlying file system uses techniques such as RAID, then the benefits of such techniques carry over.

With respect to availability, notice that NFS v4.0 adopts a stateful model. Hence, it includes some steps to address the availability issue. The NFS server maintains the state of files and the owners of locks on these files. This state information needs to be recreated if the server crashes and recovers. Service interruptions and lack of availability can ensue in this interregnum.

NFS v4.0 in particular uses a stateful protocol and supports locking. It appears that this allows for a more stricter consistency model for concurrent writes. However, notice that the exact consistency achieved will also depend on the NFS server implementation and the support that the underlying file system provides. For instance, advisory locks on a file by a client still allow for writes by other clients. RFC 5661[3] provides more information on advisory locks and mandatory locks in the context of NFS v4.0.

[3]https://datatracker.ietf.org/doc/html/rfc5661

8.3.4 Summary of NFS

From the above, we conclude that NFS introduced a pioneering idea to the design and development of distributed file systems. NFS works well under the assumption that most files are not shared, very few concurrent operations exist, and most operations are read based, and the like. In such a setting, a simple consistency model coupled with client and server side caching helped NFS address issues such as performance, scalability, and consistency. However, the NFS server being involved in all operations on every file does become a bottleneck. Subsequent designs of distributed file systems remove some of these limitations and also change the set of assumptions in part owing to changes in the landscape of distributed computing.

8.4 The Andrew File System (AFS)

In this section, we describe the Andrew File System (AFS) and its development and features. The Andrew File System started as the ITC file system in 1985 with a subsequent new version termed as the Andrew File System, AFS v2.0 or just AFS. The ITC distributed file system is often termed as AFS v1.0. We will describe both the versions, starting with v1.0.

AFS, just like NFS, assumes that most files are not shared across users. It also assumes that most files are read in full and sequentially by applications/clients. These assumptions drive the design of AFS to optimize for these operations.

8.4.1 AFS v1.0

AFS v1.0 allows clients to access files on a remote file system just as they are on a local disk. To create this transparency, AFS supports a mechanism called whole-file caching unlike NFS. In this mechanism, the first time a remote file is opened by a client, the entire file is transferred to the client side and kept locally by the client. This allows all subsequent file operations to be performed locally. On the file close operation, the entire file is again sent over the network to the server. This simple description misses a few details which we elaborate now.

AFS uses the namespace structure of the underlying (remote) file system. Clients have to specify the full path name to access a file. The AFS server reads through the file system hierarchy and transfers the entire file to the client so that the client can keep a local copy. Further read() and write(offset, data) operations do not go through the server. Each operation however follows after the client checks the validity of the file from the server. On a client close() operation, the client checks if the contents of the file changed between its open and close operation. If so, the client sends the entire file to the server for updating the remote copy.

If the client still has the file in local space, a subsequent open() operation need not fetch the entire file always. The client can check with the server if the file changed in the interim: if not, the local copy can be reused.

The simple design of AFS v1.0 however has a few drawbacks. Primary among them are the contention at the server, cost of operations on the server including the cost of replying to the client call to check the validity of the file(s) they own locally, load imbalance issues, and the limitations of handling a large number of clients. In the early version of AFS v1.0, the server could engage with up to 20 clients. [**?**].

8.4.2 AFS v2.0

One of the ways to reduce the number of calls from the clients to check the validity of the file that they hold locally is to turn around the initiator of the check. In other words, if files are not usually shared often, it is safe to imagine that there would be fewer instances when the client call to validate the status of a file returns a response from the server that corresponds to indicating that the local copy is valid. Thus, several calls of this kind are redundant and offer a scope for improvement.

In AFS v2.0, the server sends an invalidate message to clients indicating that a copy of the file that they own locally is no longer valid. This mechanism, essentially like a push instead of a pull, requires the server to send a message to clients that hold a copy of the file whose contents change. This mechanism requires the server to maintain the required state information in terms of the list of clients that have a copy of a given file. Each client will presume that the local files that it holds are valid unless it hears from the server otherwise. AFS v2.0 names this mechanism as a **callback**.

Another optimization that AFS v2.0 introduced beyond v1.0 is to provide each file with an identifier in addition to the path name.

8.4.3 Consistency

AFS follows the “last write wins” consistency model where the updates of the last client that performs a write is what is visible to other clients. The above is true so long as the clients are on different machines. However, if the clients are on the same machine, the updates of one client are immediately visible to the other client. The latter model essentially resembles what typical UNIX based file systems follow.

There is one difference between how NFS and AFS treat file updates. In AFS, the entire file has to be sent back to the server at the time of close whereas in NFS, only blocks of a file are sent back to the server at the time of close. This distinction also happens to match with how NFS and AFS differ in how they cache files locally. Notice that this distinction also has some implications for how the files get impacted due to writes. If multiple clients modify parts (blocks) of a file in non-overlapping areas, it is not always clear that the updates can be merged. For instance, if the file corresponds to an image, and if all the updates are taken together into a common file, clients (users) will see portions of the file that may not be a valid image file or an expected image file. Hence, in some settings, it is important to write back the entire file as in AFS as opposed to only blocks in NFS.

8.4.4 Fault Tolerance

Like NFS, AFS has mechanisms to recover from server and client failures. If a client is unavailable while the server sends a invalidate callback, the client would not be aware of such messages. Therefore, upon recovering from any errors, clients have to treat all their local file copies as invalid and seek the validity of files from the server.

Similarly, clients have to handle a server crash by again checking for the validity of the files that they have a local copy of. The server maintains state information corresponding to the list of files that each client has open. However, this information is kept in the volatile part of the server memory and is susceptible to information loss on a server crash. Thus, the server may not be able to recreate the required state information on a crash. One possibility is for the server to immediately notify on recovery that all client held copies are invalid.

8.4.5 Design Features and AFS

We notice from the above discussion that AFS is not very different in its features with respect to NFS. The AFS server acts as a layer between the clients and the underlying remote file system. Therefore, features such as durability, metadata, and block size follow from the corresponding features of the underlying file system. The consistency offered by AFS is essentially the last write wins for clients on different machines. Compared to NFS, as the entire file is copied back to the server during a close operation where the file has changed, the last write wins is applicable to the entire file and not a block of the file. The AFS protocol handles faults via suitable retry and invalidate/validate mechanisms. AFS, both v1.0 and v2.0, suffer from scalability issues due to the large amount of messages between the clients and the server. While AFS v2.0 improves on the scalability compared to AFS v1.0, both fail to achieve the desires of scalability for current generation systems.

8.5 The Google File System (GFS)

From the section on NFS, we understand that it is possible to make some gains in performance by trading off on consistency. However, a few deep issues exist in the design of NFS. Every file write operation eventually requires action from the server. This will become a bottleneck as the system scales. Further, each file read operation involves multiple disk operations that may impact the performance of the file system. To be able to process the requests of multiple clients, one also needs a powerful server supported with other software mechanisms such as the ability to handle multiple connections, load balance, fault tolerance, and so on. The presence of the server also makes it a single point of failure.

In addition, the nature of workloads and system assumptions changed over the early 2000s. The rapid inroads made by internet and online platforms for shopping, banking, search, and entertainment, generated large volumes of digital data. It is estimated that organizations such as Google and Facebook produce and process data of the order of a few petabytes every day.

This large volume of data and its processing created workloads with very different characteristics compared to the earlier decades. For instance, appending to a file has become a useful operation instead of the standard write operation. In the former, the location of the write is not specified by the user so that the file system can identify a suitable location to add the data specified by the user, whereas in the latter the user specified the offset (location) and the data. Another such change in the nature of workloads is the use of concurrency and parallelism. The large volume of data that workloads deal with requires recourse to parallelism. Workloads may also produce huge amounts of data as a by-product. The file system should therefore be able to support concurrent operations on files with appropriate consistency guarantees.

There have been tremendous changes to hardware too in the post-2000 era. It has become easy and cheaper to prepare systems with off-the-shelf commodity machines rather than invest in expensive high-end hardware which also becomes obsolete over time. However, off-the-shelf commodity systems come with a higher rate of component failure, and hence system software has to be prepared to deal with component failures.

The above considerations led to the design of the Google File System (GFS) in 2005 [52]. The design principle can be termed as *workload-oriented* design where the main consideration is to cater to the nature of most frequent workloads that use the distributed file system. The Google File System therefore incorporates an operation called `record-append` whose semantics is different from the `append` operation supported by Unix style file systems. In the latter, the `append` operation adds the data to the end of the file position – the position that the client believes to be the end of the file. In GFS, the location where the data is written is chosen by the system and the system provides certain guarantees on the append operation (see also Section 8.5.1). For this reason, GFS is *not* POSIX compliant.

8.5.1 Design and Implementation

The GFS design includes three main components: a master server, a set of chunk servers, and the client. One of the main design principles of GFS is to not require the server (the master server) to be playing an end-to-end role in any operation. To this end, the server keeps only metadata related to the files and does periodic system monitoring. All client operations are offloaded to the chunk servers after a quick handshake at the server.

Basic Design

GFS stores files using fixed-size *chunks*. Each chunk has a unique 64 bit identifier called the *chunk handle*. Chunk servers store the chunks on their local disk and each chunk is replicated on multiple chunk servers for purposes of reliability. Note that a commodity machine can act as a chunk server and hence prone to component failures. Replication of chunks across multiple chunk servers addresses this aspect. Figure 8.4 shows a schematic architectural view of GFS.

The metadata that the master server maintains includes information related to the file namespace, access control information, information about the chunk handles of the chunks pertaining to the file, and the location information of the chunk servers that store these chunks.

For every chunk, one of the replicas storing the chunk is also marked as

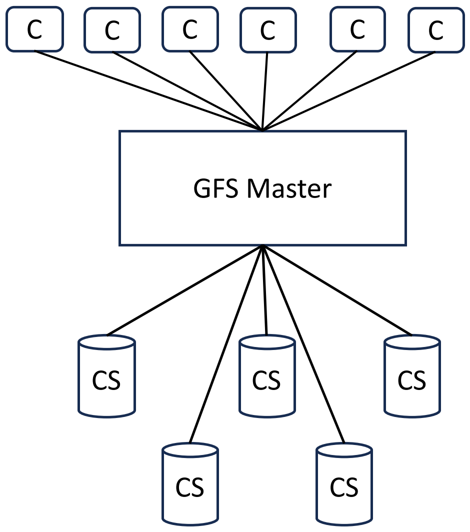

Figure 8.4: Figure illustrating the overall design of GFS. The entity marked with ‘C’ corresponds to a client. The entity marked with ‘CS’ corresponds to a chunk server.

the *primary* chunk server and others as the secondary chunk servers. The GFS master server nominates the primary and the role is assigned as a lease. The master server monitors the health of the primary chunk server so that in case of failures of the primary chunk server, the master server can reassign the role to another chunk server containing the chunk. The master can also reassign the role of the primary for a chunk to another chunk server at the end of a specified lease period.

Operations

We will now illustrate how GFS performs operations such as read/ write/ and record-append.

- **Read:** A read operation starts with the client issuing a read request indicating the file name and the offset to be read from. On receiving this information, the server replies to the client with the chunk handle location(s) of the chunk server(s) holding the chunk corresponding to the desired offset. With this reply, the server essentially offloads the read operation to the appropriate chunk server and does not have to play a role in the actual transfer of data.

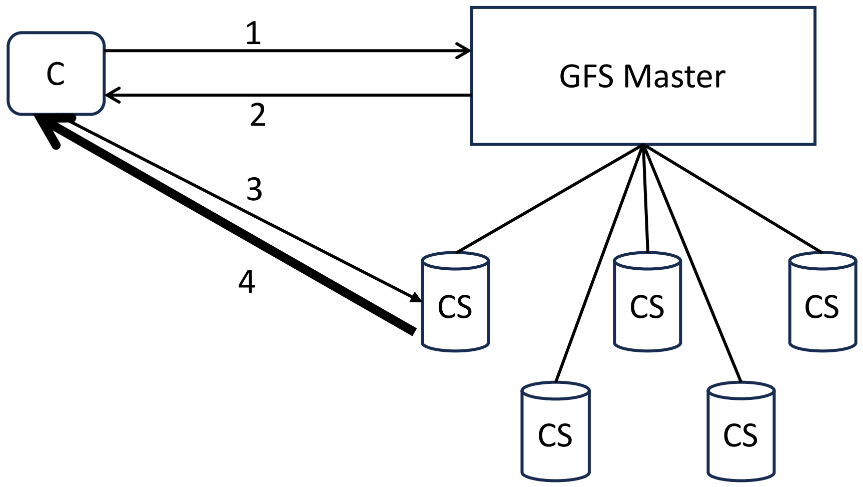

Figure 8.5: Figure illustrating the steps in a read operation in GFS. CS refers to a Chunk Server. Thin lines indicate transfer of control/metadata information and thick lines indicate transfer of data.

The client contacts the chunk server and provides information on the starting and ending byte numbers to be read. The chunk server can reply to the client with the corresponding data.

Notice that the GFS master server is only replying with metadata information whereas the actual data movement is from the chunk server to the client. Figure 8.5 illustrates the various steps involved.

Failures: Failures of any of the involved entities in the entire operation can be handled by a suitable retry. The client can cache the information about the location of the chunk servers containing the chunks corresponding to the file. This allows the client to use the cached information to read directly from the chunk server instead of asking the Master for the location of the chunk servers. The client can use its own cache policy as to how long the location information is kept in the cache.

An additional optimization that is possible is for the server to send the location of chunk servers of chunks of the file beyond the requested chunk. This optimistic buffering reduces network traffic and is suitable for applications that read files sequentially.

- **Write:** The implementation for the write operation is a bit more involved than the read operation. This is due to the fact that a read operation, and even a concurrent read operation, does not modify the state of the file. A write changes the contents of the file and hence the system should make clear the guarantees it provides on a write operation, in particular a concurrent write.

The client initiates a write operation by specifying the file name and the offset where it intends to write the data. The master server responds with the location of the primary and the secondary chunk servers that store the required chunk. Assume for now that the chunk has enough space from the specified offset to hold the data to be written.

There are two steps in the operation: (i) sending the data to *all* the chunk servers, and (ii) ensuring correctness of the operation in coordination with the primary chunk server. The client can perform the first of these steps by sending the data to all the chunk servers in any order it chooses to. This step completes when all the chunk servers acknowledge the client about receiving the data. The chunk servers keep the data in their local buffers at this point.

For the second step, the client approaches the primary server with a write request so that the primary chunk server can assign a serial number to the write operation. These serial numbers help in supporting the necessary serialization in case multiple clients attempt to write to the file simultaneously at overlapping locations. The primary sends the write request to all the secondaries along with the serial number. The secondaries have to apply the write updates in the same serial order specified by the primary. (See also Question xx)

Figure 8.6 shows the steps involved in the write operation. The secondaries report to the primary about the success or failure of the write operation. The primary accordingly informs the client of the success or failure. If any secondary encounters an error in the write operation, then the primary responds to the client that the write operation has failed. Notice that the write might have succeeded at the primary and some secondaries. So, if some secondaries report a failure to complete the write operation, then the corresponding region of the file stays in an inconsistent state. Applications are expected to handle such errors and inconsistencies. If the client gets an error from the primary for the write operation, the client can choose to retry.

For writes that straddle a chunk boundary, the operation is split into multiple write operations. Each such operation is treated as a separate

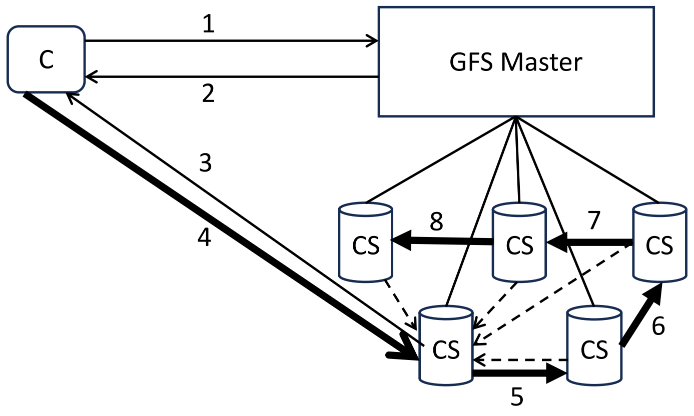

Figure 8.6: Figure illustrating the steps in a write operation in GFS. The arrows indicate the direction of the data transfer. The arrows with dashed lines indicate acknowledgments from chunk servers to the primary chunk server. These acknowledgments may arrive in any order and hence numbers are not provided.

write and hence could be interleaved with write operations from other clients with different serial numbers. Therefore, the contents of the file may be a mixture of writes from different clients. However, as long as all the operations succeed, the order of data in all the replicas is exactly identical since these are written in the serial order assigned by the primary.

Failures: Notice that it is likely that the primary or a subset of the secondary chunk servers may fail to complete the write operation. In this case, the client can retry the operation. However, on subsets of chunk servers that the operation succeeds, the modified region is left in an inconsistent state.

- **Record-Append:** This is an operation that is driven by the nature of workloads. In this operation, the client provides only the data to be written, and the file system adds the data at an offset chosen by the file system. The file system guarantees however that the data will be written at least once atomically.

  Just like the write operation, here too, the client gets the location of the primary and the secondary chunk servers, holding the last chunk of the file, from the master server. The client then sends the data to be written to all the chunk servers. Assume for a moment that the data to be appended fits within the last chunk. In this case, the primary adds the data to the end of the chunk, informs the secondary chunk servers about the offset to use for this append, collects status of the operation from the secondaries, and then informs the client of the status.

In the case that the space in the current chunk is not enough to accommodate the append, the primary server takes the following steps: pad the space in the current chunk, inform the secondaries to do so, and inform the client that the append operation failed due to lack of space. The client can then retry the operation.

Failures: Notice that if the record-append operation fails at one or more of the replicas of the chunk, then the client can retry the record-append operation. This means that the replicas of a chunk may contain different data. However, the record-append operation is successful if all the secondaries report a success on the operation. Atomicity is guaranteed due to the primary choosing the offset of the append. Hence, GFS supports the append operation with *at least once atomically* semantics.

It is up to the applications to handle the case of duplicate records and padded data. The former is usually handled with having unique identifiers for each record. The latter is easily handled with techniques such as checksums which can flag padded data.

- **Other Operations:** GFS supports other operations such as a `snapshot`. In this operation, the goal is to make a copy of a file or a directory. GFS does this in a way that this copy is instantaneous irrespective of the size of the file (or the directory). This operation is useful in cases where a user wants to create a checkpoint or a copy of a file.

  The snapshot operation is instantaneous since GFS adopts a copy-on-write approach. This copy-on-write approach is a standard systems practice that saves the expense of writing out the copy when the resource does not undergo any changes subsequently [4].

  To perform a snapshot operation on a file f, the GFS Master first invalidates/revokes all the leases on all the chunks that correspond to the file f. This means that any future write operation to f will need

[4]Most Unix style operating systems use this copy-on-write practice when a child process is created. The child process shares all the pages of the parent process and the copy-on-write flag is set on all these pages [9].

to go through the Master server. At this point, the Master can make a copy of the chunk by creating a new chunk to enforce the copy-on-write semantics. Notice that all the chunk servers that hold the chunk will also need to make a copy. These copies are created on the same chunk server where the original chunk resides so as to reduce the network traffic involved in copying the data. The Master uses reference counts to know if a file is part of a snapshot request. A reference count bigger than 1 indicates that the file is part of a snapshot request.

8.5.2 Leases

For every chunk, GFS assigns the role of a primary chunk server to one of the three chunk servers that store a copy of the chunk. This role of a primary chunk server is assigned as a lease for a duration of 60 seconds. The currently designated primary chunk server can get extensions of the lease indefinitely. The extension request and grant is sent along with the periodic heart-beat messages that go between the master server and the chunk servers. The master can revoke the current lease for various reasons including instances when the file is being renamed and the master wants to prevent changes to the file contents, the master loses communication with the primary server due to any failures, and so on.

8.5.3 Chunk Size

GFS set the size of each chunk to be 64 MB. This choice is justified given that the system expects to store very large files. In such cases, the large chunk size helps in several ways. For instance, the client need not interact with the server until the operations (read/write/record-append) cross a chunk boundary. All of these operations can be performed based on the identifier of the primary chunk server that the client obtains from the Master server during an earlier request to the same chunk. There is also the possibility that the client can keep a persistent network connection to the chunk server across multiple operations directed at the same chunk and the chunk server, thereby optimizing on network traffic. In addition, a large chunk size also reduces the size of the metadata information that the Master has to keep.

There are some usual disadvantages of using a large chunk size. One potential issue is internal fragmentation where the file size is much smaller than the chunk size. GFS handles this by actually storing chunks as regular files in the underlying operating system. The space allotted to the chunk is increased on demand. GFS calls this as *lazy space allocation*. Lazy space allocation however does not handle one potential problem, hotspots. If a file contains only one chunk, and many clients want to read from this file, all these requests will be directed to the primary chunk server storing the corresponding chunk. GFS handles this issue by other means such as a combination of increasing the replication factor, spreading read operations to not just the primary chunk server but also the replicas/clients, stagger the requests, and the like.

8.5.4 The GFS Master

We now provide more details of the GFS master server. The master server keeps the metadata corresponding to the files and the chunk servers that hold the chunks of the files in its primary (volatile) memory. This mapping is not stored in any permanent storage. The information on which chunk server is the primary chunk server for a given chunk is kept fresh by constant polling of the chunk servers. Doing so also takes care of master server handling the failures of the chunk servers. Chunk servers that do not respond to the poll messages are assumed to be failed and the responsibility is quickly reassigned to a secondary chunk server.

The master server also maintains a log of the operations that result in changes to metadata. This log is kept on persistent storage, provided with checkpointing so as to minimize time for recovery, and also serves to define the order in which concurrent operations are affected. Response to client operations is sent out only after the corresponding log record is flushed to permanent storage.

Unlike Unix style file systems, GFS does not support per-directory data structure of the list of files in that directory. File names are stored as a lookup table that maps the full pathname of the file to its metadata.

For operations by the master server that require changes to the namespace, the master server uses locks over appropriate regions of the namespace. This avoids the situation that time consuming operations at the master lead to unavailability in servicing of any file.

8.5.5 Fault Tolerance

GFS is designed for a scale where there are several commodity processors acting as chunk servers or as a master server. At this scale, components failures are the norm. A detailed study by Pinheiro et al. [106] shows that disk drives experience an baseline Annualized Failure Rate (AFR) ranging from 1.7% to 8.6% over the first year of their lifetime to their fifth year of lifetime. Hence, the system must take steps to ensure operations continue even in the presence of intermittent failures. GFS takes the following steps to address this aspect.

- Replicating the Master State: Just as a chunk server can fail, the master server that is also a commodity server can fail. This situation can render the whole file system unavailable. To prevent such an occurrence, GFS replicates the state of the master. The state of the master includes metadata about the file namespace, information on chunk servers, and the mapping between files and chunk servers. In addition to replicating the state of the master, GFS also keeps the file namespace and the chunk namespace in persistent memory by logging modifications to the two namespaces into persistent storage on the master's disk and also on to the disk at other remote machines. A change to state is said to be committed only after such a change of state is recorded to a persistent storage.

  On the failure of a master, the replicated log is useful in quickly creating a new master process. To keep this change transparent to the user, clients use only an indirect way of addressing the master. For instance, the indirect address of the GFS master could be a DNS alias instead of an IP address.

- Replication of Chunks: Each chunk is replicated three times by default. Users can increase the replication factor based on their application requirements. Other mechanisms such as erasure codes could be tried instead of full replication.

- Data Integrity: Each chunk is treated as a logical collection of 64 KB blocks. Each such 64 KB block is provided with a checksum so that the chunk server can verify each block with the checksum before returning data to any request. The checksums are kept in memory and also stored in persistent storage with logging.

  If there is a failure to match the checksum, the chunk server can declare the chunk to be in error to the master and report a failure to the request. This allows the master to create a new replica of the chunk from other chunk servers holding the chunk and inform the chunk server that has the wrong chunk to delete the chunk.

  Having a large chunk size reduces the overhead of computing the checksum on every read request. The verification of the check sum can happen in parallel to the read. As GFS is geared more towards Record

Append instead of write, checksum can be updated incrementally during the Record Append operation. This choice also means that a regular write is slow compared to the Record Append since a regular write requires recomputing the checksum.

Chunk servers also use their idle time to verify the checksums of the chunks they hold so as to identify any *bad* chunks. By reporting to the master, these bad chunks can be removed from the system by replicating the bad chunks with good chunks.

- Shadowing : GFS maintains *shadow master* shadow master processes that tails the progress of the master up to a small delay. The shadow processes can allow read access to files in case of unavailability of the master server. Shadow master processes update their state frequently based on the logs written by the master server and apply the updates from the log in the same order to their local data structures. Notice however that the content served by shadow processes can be stale.

8.5.6 Design Features and GFS

Let us now understand how GFS fares with respect to the common design features listed in Section 8.1.

To ensure data durability, GFS replicates each chunk of the file at three different chunk servers. The degree of replication can be increased. Notice from the GFS operational summary that the replicas are held consistent to a great extent. (Question: When can replicas diverge?) This ensures that the file contents can be recovered when a chunk server crashes. Even the state of the GFS Master is replicated so as to ensure that the file system state can be quickly recreated in the event of a crash of the Master server.

GFS maintains availability by having mechanisms such as (i) replication of chunks at multiple chunk servers, and (ii) a shadow master to keep the system available in the event of crashes to the master server. The presence of a single master may mean a single-point of failure, mechanisms such as shadowing and state replication allow for increased availability and fault-tolerance and recovery.

GFS ensures that concurrent writes and record-appends come with guarantees. The write operation is serialized by the primary chunk server by providing and enforcing the update order. This ensures that all clients see the data in the same order. The primary chunk server assigns at the offset for each record-append operation and all replica chunks are updated by using the assigned offset. This ensures that each record-append is performed at least once atomically.

To ensure scalability, GFS allows for having multiple chunk servers store the data.

GFS makes the choice of using a flat namespace instead of an hierarchical namespace that is more common to Unix type of file systems. The namespace and metadata that GFS maintains is more lightweight and this allows for keeping the metadata in volatile storage of the GFS Master allowing for quicker access to the metadata.

8.5.7 Summary

The design of GFS is geared towards a system that works well with batch jobs. Characteristics of batch jobs such as having a long job time, tolerance to short failures, ability to view the computation as record processing, make it suitable for systems such as GFS.

GFS coupled with Map-Reduce allowed Google to support a strong search engine and other Google products. As the nature of products and services moved to more real-time, the deficiencies of GFS such as having a single point of failure at the master server began to make an impact.

8.6 Colossus

With the increase in the amount of data that Google deals with and also with the increase in the variety of services with different requirements, a file system that can scale better than GFS is seen to be desirable. This led to the development of Colossus [64], the enhanced version of GFS. Colossus is a cluster-level file system and is one of the main building blocks of supporting both Cloud and Google products.

The Colossus file system is designed to scaled for beyond Exascale data while at the same time offering high availability and being able to support specific needs of multiple applications. The goals of Colossus is to arrive at more predictable tail latency beyond being faster and bigger. Some of the functionality of Colossus is made possible due to the distributed metadata model that is uses. The main components of the Colossus are (i) the client library, (ii) the Colossus Control Plane, (iii) the Metadata database, (iv) the data servers, and (v) the Custodians. Figure 8.7 shows the schematic architecture of Colossus. In the following, we elaborate on each of the above components.

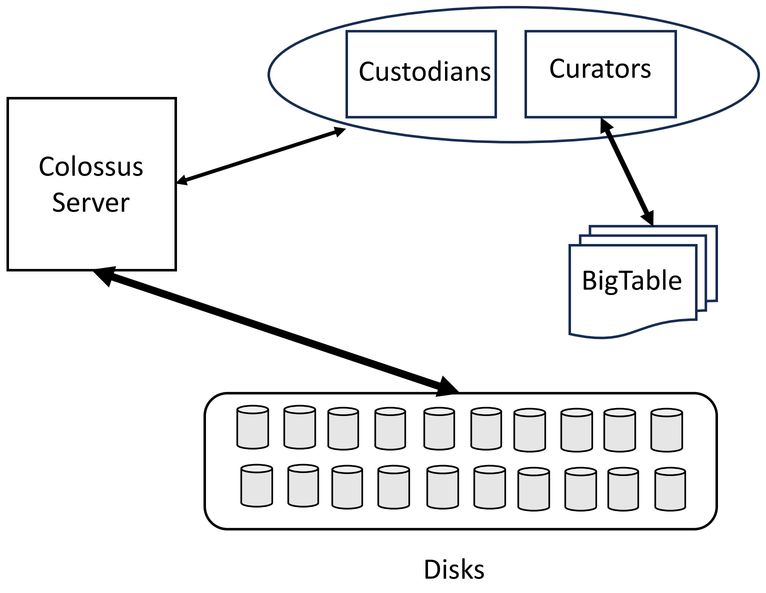

Figure 8.7: Figure illustrating the schematic architecture of the Colossus file system.

The Client Library

The client library packs a lot of functionality and applications use the client library as their primary interface to use Colossus. The client library has support for a variety of functionality that applications can tailor based on their requirements and their cost-performance trade-offs.

The Colossus Control Plane

The consists of Colossus Curators which keep the file system metadata. The number of curators in the system can scale horizontally. Clients interface with the curators for file control operations such as file creation.

The Metadata Database

Curators keep the metadata in a Google No-SQL style database, Big Table [23]. (See also Chapter 9.5 of this book.) This is one marked departure from GFS. Using only main memory for metadata as done by GFS is not scalable. On the other hand, moving the metadata to a database helps scale the file system to a factor of 100 or more.

The Data Servers

Colossus system is built to work with a variety of data servers. Notice that with horizontal scaling, new disks are added to the system and old disks are retired. The characteristics of the disks that are in use at any given time vary in their size and access models too. In Colossus, the disks are attached to the network so as to minimize the number of hops that data stays on the network.

The Custodians

The scale of operation of Colossus suggests that periodic background services are needed to keep the system up and running. These background mangers are called as Custodians and support operations for maintaining the durability and availability of data, overall system efficiency, disk space balancing, and RAID reconstruction.

8.6.1 Overall System

Colossus ensures that data that is likely to be more frequently accessed, hot data, is kept in flash drives. This ensures that hot data is served with less latency. Other data is moved to disk storage. Further, newly written data is spread evenly across the drives in a cluster. Periodic data rebalancing also ensures that data is moved to newer and larger disks eventually. All these tasks are possible since Colossus aims to take benefit of the fact that across applications, the data access patterns vary significantly.

The scale at which Colossus operates essentially means that it uses a large amount of commodity hardware. In practice this comes with failures to components are common. Any time instant, there will be some components that have failed. To address fault tolerance, Colossus custodians undertake fast background recovery and keep failures transparent to the user and provides highly durable and available storage.

8.6.2 Other Considerations

Given the deluge of data and the nature of its access, two phrases that are popular today in the systems community are hot data and cold data. Hot data refers to the data that is being accessed more frequently presently, whereas cold data refers to data that is not frequently accessed. This classification is important when one looks at applications such as photos being accessed online, e.g. Google Photos, or Facebook multimedia posts, where such items are accessed very frequently around the time they are made available (posted), and over time are accessed very rarely.

The classification of data into hold and cold data is important from a systems perspective in the following way. In a file system with multiple disks, it is essential to keep both kinds of data on all disks. Otherwise, if a disk largely has cold data, then the usage of that disk is disproportionately low while a disk with largely hot data get used more often potentially leading to faster failures. Colossus, therefore, aims to keep a percentage of each disk with hot data and the rest of the disk with cold data. To ensure this condition over time, Colossus rebalances disk contents and distributes new data evenly across available disks.

In addition, a subset of the data, the most frequently being accessed, is put on flash drives to minimize access latency.

8.6.3 Design Features and Colossus

Colossus is prepared with scale in mind. The system allows for the addition of disks to enhance the storage capacity of the system. Current usage levels already exceed the exabyte mark [**?**].

In a departure from GFS, Colossus brought down the chunk size from 64 MB to 1 MB. The design of Colossus is towards having a large number of distributed storage nodes, each storing a large number of small files. As part of this restructuring, notice that the size of the metadata grows compared to that of GFS and the expected number of files that Colossus stores. To address this issue, the metadata is no longer stored in the volatile memory of the Master, but in a database, Bigtable.

Having a distributed master node design also improves the availability of Colossus and makes the system resilient to the single point of failure that GFS has to contend with.

Colossus provides for data durability by using replication.

8.7 Systems for Object Storage – Haystack

Apart from Google, Facebook is another platform that deals with massive amount of data. Files handled by Facebook include audio/video files and photographs as the platform is used primarily to share photos and audio/video content across friends. So, the file system should support quick delivery of content at a planet scale. While Content Distribution Networks (CDNs) also offer a solution that appears similar to that of serving files across the planet, CDNs are typically optimized for frequently used data items. On the other hand, the kind of workloads that Facebook deals with will have a long tail and for overall user experience, it is important to also see how the latency of requests corresponding to the long tail can also be minimized. In this case, CDNs may not offer a good solution since CDNs mostly rely on caching mechanisms and objects in the long tail do not make it to the cache by definition.

In 2010, Facebook rolled out its distributed object store called HayStack [16] that is primarily targeted at storing photos efficiently. We now describe the shortfalls of existing systems and describe the mechanisms that Haystack uses to store and serve photos efficiently.

8.7.1 Shortfalls

One of the natural ways to cater to the storage and serving of photos would be to store each photo as a separate file in a file system and use NFS to scale to the large number of photos that are shared on the platform. However, this comes with multiple drawbacks summarized in the following.

In a traditional file system, each file is stored with a large amount of metadata including access control. The typical Unix file system creates access control to the owner, the group(s) the owner belongs to, and others. But, a system that has a large number of read operations (viewing photos) and very few writes does not require elaborate mechanisms for keeping file access control information. Typically, only the creator of the file is the owner and other users can only read the file and will not be able to write to the file. Such unused fields, especially at billion scale files, bloat the metadata that will be read every time the corresponding file is accessed increasing the bottlenecks and reducing the throughput.

As mentioned previously, CDNs do a good job of storing and serving objects that are accessed most frequently. With systems such as Facebook also getting significant volumes of requests to stale/infrequent data, CDNs do not cater to such traffic thereby increasing the latency on such requests. Thus, relying on CDNs alone which using principles from caching may not cover all possible use cases.

For higher throughput, it is an advantage to keep as much metadata in memory as possible. However, if the metadata corresponding to each file is large, this limits the amount of metadata that can be stored in the main memory. So, it is important to limit the size of the metadata and see which fields of a traditional file system metadata are extendable for the chosen workloads.

8.7.2 The Haystack System

The Haystack system [16] has three main components: the Haystack store, the Haystack Directory, and the Haystack Cache. The Haystack Store has the filesystem metadata and encapsulates the persistent storage system for photos. The Haystack Directory maintains the logical to physical mapping along with application metadata. The Haystack Cache functions as an internal CDN and serves requests for hot photos – these requests need not be served from the Haystack Store. The Haystack Store sits on top of an existing file system. Facebook used XFS as XFS has certain advantages to the considered workload. In the following, we will detail these components.

The Haystack Store

The Haystack Store is arranged via physical volumes. Each physical volume holds millions of photos. Haystack stores multiple photos in a single file and hence the files are are very large. Physical volumes across machines are grouped into logical volumes. Haystack replicates a photo on a logical volume by writing the contents of the photo is written to all the physical volumes that constitute the logical volume. This replication allows Haystack to address fault tolerance due to failures of hard drives, disk controllers and the like.

The Haystack Store machine accesses a photo using only the id of the corresponding logical volume and the file offset at which the photo resides in the logical volume. A HayStore machine represents a physical volume as one large file. The file has one superblock followed by a sequence of needles. Each needle represents a photo stored in Haystack.

Retrieving needles quickly is an essential ability of the Haystack Store. To this end, each Haystack Store machine keeps open file descriptors for each physical volume that it manages and also an in-memory mapping of photo ids to the filesystem metadata (i.e., file, offset and size in bytes) that is useful in retrieving that photo. Each machine also maintains an in-memory data structure for each of its volumes. That data structure maps pairs of (key, alternate key) to the corresponding needle's flags, size, and offset.

The Haystack Directory

The Haystack directory has four main functions as listed below.

- The directory provides a mapping from the logical volumes to the physical volumes. This mapping is useful for web servers to construct the URLs for photos being requested and is also useful in processing an upload.

- The directory also balances the load across the logical volumes and spread reads across the physical volumes.

- The Directory determines whether a CDN or a Cache handles the request to access a given photo.

- The Directory marks certain physical volumes as read-only once these physical volumes reach their full capacity or because of operational reasons.

Typically, over time, the system will add more physical volumes as the existing volumes reach their storage limits. The new volumes are marked for write. Only volumes that are marked for write can receive uploads.

The Haystack Cache

The cache keeps a mapping of the photo id to the location of the photo. The cache is essentially a distributed hash table. Haystack uses two rules to determine if an object/photo is kept in the cache: (i) the request being served from the store comes directly from a user and not a CDN, and (ii) the photo is fetched from a machine that is enabled to receive uploads. The second of the two conditions indicates that most recent photos are the ones that are recently uploaded and hence are being written to a write enabled machine.

File System

Haystack can be viewed as a distributed object store that sits on top any generic file system. However, Facebook used XFS as the underlying file system. First, the block maps for several contiguous large files can be small enough to be stored in main memory. Second, XFS provides efficient file pre-allocation, mitigating fragmentation and reining in how large block maps can grow. Using XFS, Haystack can eliminate disk operations for retrieving filesystem metadata when reading a photo. However, it does not imply that Haystack can guarantee every photo read will incur exactly one disk operation.

8.7.3 Operations

In this section, we discuss the steps that Haystack uses to store a photo, serve a photo, delete a photo, and so on.

Storing/Uploading a Photo

When uploading a photo into Haystack web servers provide attributes such as the logical volume id, key, alternate key, cookie, and data to the Haystack Store machines. Each machine then synchronously appends needle images to its physical volume files and updates in-memory mappings as needed.

Operations on photos such as rotations have to create a new needle. This new needle will have the same key and alternate key as the original photo. There are now two possibilities. The new needle is stored in the same logical volume. In this case, the new needle is appended to the same physical volumes as the original needle. The Haystack store uses the rule that the needle with a larger offset is considered as a later version. If the new needle is stored in a different logical volume, then the Haystack Directory updates the metadata entries corresponding to the original needle so that all future requests fetch the new version.

Serving a Photo

Consider a user requesting for a photo through a browser. This request goes to the web server and can be served by the external CDN of the internal Haystack Cache if the request missed the CDN. To facilitate a smooth hand-off between the CDN and the Cache, the web server constructs a URL for the request as follows. Each URL is of the form `http://<CDN-Name>/<Cache-Name>/Machine-Id/<Logical Volume, Photo>`. In this encoding, notice that the first part of the URL specifies the CDN that can possibly serve the request. If the request is not satisfied at the specified CDN, the CDN can remove the corresponding part of the URL and forward the request to the Haystack Cache specified in the URL. If the Cache fails to serve the request, then the Cache can remove the corresponding part of the URL and forward the request to the machine (Haystack Store) that can serve the request. At this level, if the photo is found in the specified volume at the given offset, it is served. Requests that go directly to the Haystack Cache follow a similar process except that the URL is missing the CDN specific information. Figure 8.8 shows the interactions between the user, web server, the CDN, and the Haystack for serving a request.

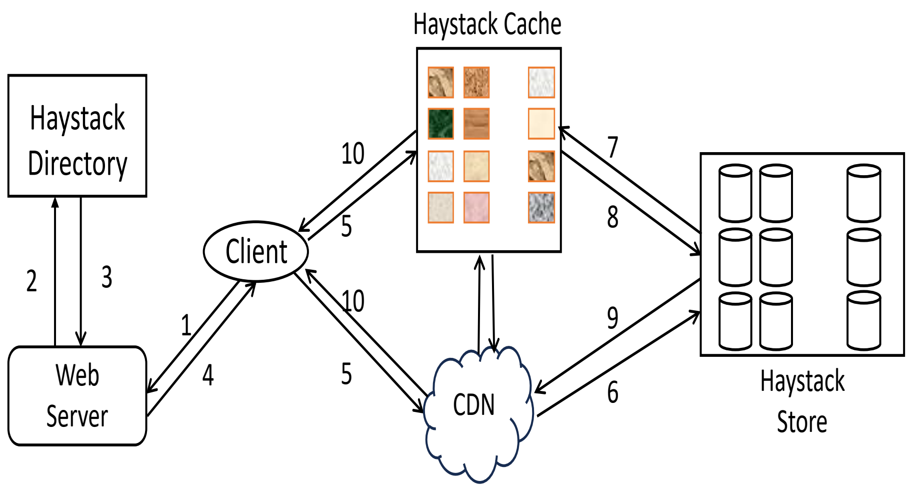

Figure 8.8: Steps involved in reading a photo via haystack.

The process of serving the photo from the haystack Store follows the steps below. The request to the Store is accompanied with the logical volume id, alternate key, and the cookie fields. Recall that these attributes are part of the needle data fields. The cookie field, being a random number assigned at the time of upload and is embedded in the URL, prevents guessing attacks that can prepare valid URLs to mount denial-of-service attacks. The Haystack Store reads the volume from the provided offset, makes sure that the photo is not (marked) deleted, verifies the cookie, and then passes the photo to the Haystack Cache.

Delete a Photo

Deleting a photo is a logical operation. The Haystack Store machine that has the photo sets the delete flag in both the in-memory mapping and synchronously in the volume file. Since requests to retrieve photos first check the in-memory flag, such photos are never returned as the answer to the request.

As with any logical delete models, the space held by deleted photos has to be reclaimed via a future internal operation. This is called compaction and reclaims space held by deleted or duplicate needles. Periodically, each Haystack Store machine goes through the entire set of needles, copies the valid needles to a new file, and at the end of this process sets the in-memory structures to the new file.

8.7.4 Optimizations

Haystack includes the following optimizations to improve performance.

- **Batch upload:** Notice that disks are good at batch updates since positioning the disk for access is a significant cost. So, Haystack prefers to batch uploads to the disks as much as possible. From a workload perspective too, most users upload a set of photos than a single photo. This enables Haystack to use batch updates in an album together.

- **Memory Footprint Minimization:** Instead of setting flag field on for deleted photos, Haystack opts to write a 0 in the offset for such photos. Further, Haystack Store machines do not store the cookie field in memory and checking the cookie is done based on the needle read from the disk.

- **Index File:** This is a (re)boot time optimization used by Haystack. When a Haystack Store starts up, it can read the entire set of physical volumes and build the required mappings to be held in-memory. This will be slow and time-consuming as this involves reading Terabytes of data from disks. The index file is akin to a checkpoint of the last known in-memory mappings. Restarting using this checkpoint version of the can result in some inconsistencies especially if the index file corresponds to stale information. These inconsistencies can create orphan needles, needles that do not have corresponding index records. To cater to this situation, the Haystack Store machine maps orphans to valid index records. If an index record corresponds to a deleted photo, the Store machine relies on the usual procedure of reading the entire needle for a photo and checking the delete flag on the needle. If the flag is set, then the index record is updated in the in-memory mapping and also notified to the Haystack Cache. REDO

8.7.5 Fault Tolerance

Systems such as Haystack by virtue of running on multiple commodity machines are prone to failures of various kinds including faulty hard drives, erroneous RAID controllers, bad motherboards, and so on. However, the scale of the systems is such that availability is key. To address this situation, Haystack has two techniques for detection and repair.

Fault detection is usually done by background processes, called *pitchfork* in Haystack. These background processes tests the health of every component and flags the faulty ones for follow-up action. This is only a diagnostic measure. The real repair has to be done offline.

8.7.6 Design Features and Haystack

We now summarize the design features of Haystack in the context of the features listed in Section 8.1. One of the distinguishing features of Haystack is that its design is more oriented towards minimizing, or eliminating, the size of metadata and the number of disk accesses needed to serve a photo request. This motivates the design of Haystack to keep metadata so small that the entire metadata fits in the main memory.

Haystack keeps photos in its store and interfaces with the store more as an object store with each on disk XFS file storing multiple photos. Haystack keeps the files large and relies on XFS to use a large block size.

Given that Haystack is built to serve a large user base, durability and availability concerns are paramount. For purposes of durability, Haystack relies on replication. Each photo is replicated in multiple geographical locations and also at all physical volumes corresponding to a logical volume. Recall that a collection of physical volumes on different Store machines is grouped as a logical volume. This redundancy allows for data durability and protects against loss of data due to failures of the hard drives, hard drive controllers, RAID errors, and so on. Access to Haystack is via an embedded URL that directs the browser to a CDN or a Haystack Cache. This model ensures high availability with simple mechanisms.

In the typical usage model of Haystack, updates to photos are the only way to modify the contents of the objects. These updates can be done only by the owner or creator of the object. These updates are treated as creating new needles for referring to the new content and the most recent needle for a given photo has the largest offset. So, Haystack does not place much of design emphasis on concurrent operations.

Haystack addresses the scale issue by allowing for machines to be added to its Store. In fact, Haystack marks all full volumes to be read-only while new volumes deal with writes. This subtle distinction allows Haystack to optimize for read-only and write-only operations to volumes and also lays out a cache policy that is tailored to fresh data.

8.8 The Facebook Tectonic Distributed Storage System

Haystack [16] is used for storing blobs and supports one kind of applications that Facebook deals with. As Facebook had to deal with multiple such applications (aka tenants), each with its own distributed storage and file system, the ensuing collection of small specialized storage systems resulted in inefficiencies including the development, optimization, and maintenance of multiple systems. For instance, tenants based on data analytics at warehouse scale data were using HDFS and are optimized for batch processing and hence aiming for better throughput over latency. However, HDFS clusters are limited in size and results in managing tens of HDFC clusters per data center just to store the data required by the data analytic jobs. Blob storage that is spread across Haystack and f4 for hot and warm blobs, respectively, resulted in poor resource utilization.

To address this issue, Facebook prepared the Tectonic distributed storage system [101] that can scale up to exabyte storage and allow for multiple tenants while making it possible to support isolation across tenants and optimizations specific to each tenant. Tectonic is arranged as a set of clusters that are local to a data center and provides a fault-tolerant storage model. Applications/tenants can also choose to georeplicate an entire cluster for better reliability.

A tectonic cluster has three main parts: a chunk store, a metadata store, and a stateless metadata services system. The chunk store is a collection of storage devices (hard drives) that store and access data chunks. The metadata store keeps metadata related to the files and chunks, and the metadata services component that run in the background to maintain consistency across metadata layers in addition to others.

8.8.1 The Chunk Store

The Chunk Store is essentially a collection of disk drives. For scaling, any number of disk drives can be added to the chunk store. In addition, the chunk store is oblivious to the actual higher-level abstractions such as files and blocks. This separation between how the storage nodes work and how the files and blocks are handled by the file system client is an important aspect in providing seamless multi-tenant support.

Each storage node can have multiple disk drives and each such node implements XFS [58]. Each storage node is provided with a large solid state drive for keeping XFS metadata and caching hot chunks. Keeping XFS

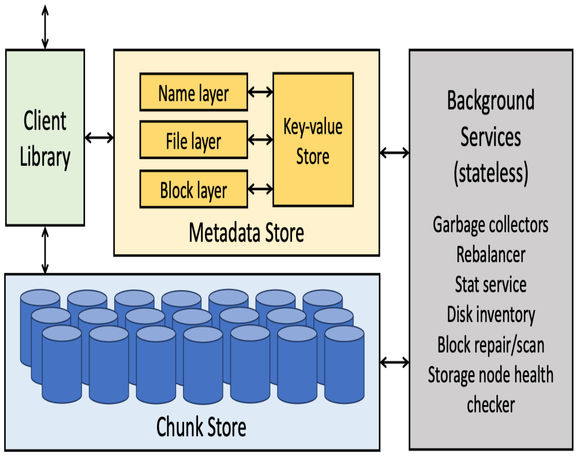

Figure 8.9: The architecture of tectonic file system. Figure taken from [101].

metadata on SSD storage is useful for applications that use blob storage since most blobs are stored written as appends that only affect the chunk size.

Tectonic supports each block to be stored as a set of N replicated chunks or to be stored as a Reed-Solomon encoding where the block is replicated as N chunks and K parity chunks. This choice can be made by the tenant based on its requirements. This flexibility of tectonic to allow for tenant-specific optimizations is crucial for supporting a unified system that can cater to the needs of multiple tenants.

8.8.2 The Metadata Store

The metadata store keeps key-value stores over three components: (i) name layer, (ii) file layer, and (iii) block layer, and the metadata store is sharded across these three layers. Each of these key-value stores is implemented as stateless microservices using ZippyDB [99].

The key-value store are stored on metadata nodes in SSD and are also sharded with replication at the shard level. The nature of sharding for the various key-value stores is shown in Table 8.1. Shards are replicated with Paxos [84] for fault tolerance. Each metadata node can host several shards. This allows shards to be redistributed in parallel to new metadata nodes to react to failures and reduce recovery time. Any replica can serve reads to the key-value store and reads that are required to protect strong consistency

Layer	Name	File	Block
Key	(dir_id, subdir_name) (dir_id, filename)	(file_id, block_id)	block_id (disk_id, block˙id)
Value	subdir_info, sudir_id (file_info, file_id)	block_info	list<disk_id> chunk_info
Sharded By	dir_id dir_id	file_id	block_id block_id
Mapping of	dir to subdirectories dir to list of files	file to list of blocks	blocks to chunks disk to list of blocks

Table 8.1: Metadata layers

are served by the primary.

The Name layer maps directories to the files and sub-directories they contain. The namespace is flat or expanded. For instance, if a directory `test` contains two files `villa` and `tower`, then there are two keys (`test`, `villa`) and (`test`, `tower`). The file layer maps files to the set of blocks corresponding to the file. A file is an ordered list of blocks where blocks serve as an abstraction for a contiguous string of bytes.

The key-value store does not provide cross-shard transactions. This limits certain filesystem metadata operations and complicates some multi-step operations. For instance, consider renaming a file `test` in a directory as file `new` even as the the file `test` and the new file will be on different shards. If this rename overlaps with the creation of a new file with the same name in the same directory, then care should be taken so as not to leave the file system in an inconsistent state. A possible inconsistency can arise due to the following play of events. For renaming the file, the three steps needed are to get the file id for the file `test`, add the file named `new` as a new file, and map the file name `new` with the file id of the file `test`. To create a file with name `new`, the steps involved are to create the new file ID, and map the file id to the filename. If the above operations of renaming a file and creating a file interleave such that the steps of creating the file are executed after the first step of the rename operation and before the second and third steps of the rename operation. In this case, the third step of the rename operation can erase the mapping introduced by the file creation.

One of the issues with applications that use blob storage is the possibility of hotspots in searching within the directory structure. The layered metadata approach of Tectonic's aims to avoid such hotspots in directories and other layers by separating searching and listing directory contents from the name layer and reading file data through the file and block layers.

8.8.3 The Metadata Services Component

The scale of the Tectonic system and its usage implies that repairing inconsistencies/loss of data, garbage collection, and health monitoring of devices are essential. For instance, an action that is left incomplete due to an intervening client failure can lead to inconsistencies in the metadata. Similarly, performing lazy deletion, which is usually done to reduce latency in real-time operations, requires a later phase of cleaning up. Storage capacity limits, load imbalances are also possible due to the scale of the operations.

The Metadata services component performs several services in the background to address the above situations. Some of the services include rebalancer, garbage collector, statistics, storage node health monitoring, disk inventory, and block repair/scan.

Copysets is another technique that tectonic uses to improve fault tolerance and data availability even in the presence of coordinated disk failures. When using replication, there is a tradeoff between the likelihood of data loss and how much data is lost per failure event. For instance, under a three way replication with uniform failure rates, the probability that a particular block is lost is increases as any set of three failures will result in the loss of a block that is stored on these three devices only. However, the amount of data lost will be low since it is also highly likely that some replicated device will have the data that is stored on only one or two of the failed devices.

To decrease the probability of data loss, the idea of copysets is to create independent subsets of the devices. Each such independent subset is called as a copyset. Each block is then replicated in the devices in a chosen copyset. In this case, the event that results in a data loss corresponds to the event where all (or a subset of) the devices in the copyset fail. On the downside, using copysets however increases the amount of data lost per failure event. Another downside of using copysets is that repair traffic is now all directed to the devices in the same copyset. This skews the repair traffic too. There are a few ways to deal with this skew [26].

8.8.4 The Tectonic Client

For the tectonic client, the unit of abstraction for read/write is a chunk. The Client Library interacts with the Metadata Store to locate chunks, and updates the Metadata Store for filesystem operations. Tectonic supports only single-writer semantics. The exact support for write operations is discussed in the next section. If a tenant needs support for multiple/concurrent write, should build their own serialization semantics on top of the support offered by tectonic.

8.8.5 Operations and Semantics

Tectonic supports single-write, append-only semantics. Thus, tectonic does not support concurrent writes to a file. All writes, even by the single writer, also are appends. On opening a file, clients are given a token the information corresponding to which is stored with the file in the metadata layer. For performing a write operation, the client has to mention the token along with the data. Only the most recent write token is allowed to mutate the file metadata and write to its chunks.

A write (append) is deemed to be complete if a majority of all chunk replicas of a block in order to commit the block are written durably. This is a quick optimization to reduce the tail latency for the write operation [33]. For full block writes, tectonic adopts another technique to know which of the chunk replicas are written first. With unpredictable delays, it is not possible for a client to know what are the best replicas to write to. So, tectonic uses a policy called as reservation requests . In this solution, the client sends small ping style requests to multiple storage nodes. The ones that respond quickly to the client request are used to write as part of the majority. This technique reduces the tail latency.

For partial/small writes (appends), the model used is slightly different. In this case too, the write is deemed to be complete once a majority of the nodes in the replica perform the write operation. This can lead to a problem if multiple concurrent partial writes append their data on different replicas. This results in inconsistencies even if the metadata seems correct since the order in which the replicas are written to can be different across the clients.

To avoid this problem, tectonic enforces two rules. Firstly, only the client that created a block is allowed to append to that block. Secondly, before deeming the operation complete, update to the metadata including the new block length and the checksum has to be finished.

The above mechanisms mean that applications can expect read-after-write consistency. This follows since the block-length and checksum after each quorum append are committed to durable storage and mutate before acknowledging to the application.

8.8.6 Design Features and Tectonic

Tectonic is designed to be scalable and scales to Exabyte scale storage levels due to its decentralized nature and ability to add more hardware resources as needed. We notice that except for NFS, the ones covered in the chapter with respect to distributed fle systems or distributed object stores do address the scalability issue.

Data consistency in Tectonic is ensured since the only operation that changes the contents is via the append-only operation that can be done by the data owner. Applications, also called as tenants in Tectonic, can provide for more diverse write support, such as the ability for multiple users to write and enforce consistency guarantees. To ensure consistency of the metadata, Tectonic employs periodic background checks.

Tectonic provides data durability by using replication. Each data block is replicated close to three times – the authors of tectonic mention that this is a small improvement over the replication factor of haystack. Beyond the system default, replication Replication for durability, geo-replication

Availability:

Namespace/Metadata To support a wide variety of applications and to provide for more scalable design, tectonic uses a nearly flat two level namespace. Tectonic supports the notion of a directory and a file. However, the contents of directory are stored in an expanded form for ease of access update unlike Unix type file systems. Unlike GFS, metadata is now stored in separate metadata stores.

Tectonic uses a block size that is of the order of KBs of data as per statistics [101].

8.9 Chapter Summary

The complete NFS v2.0 protocol specification is provided by Sandberg [112]. Several online and textbook resources describe the operation and implementation details of NFS. See [119] for an example. NFS v3.0 and NFS v4.0 are part of RFC 1813 and RFC 7530, respectively. The Amazon EC2 file system is an implementation of NFS v4.0. There are several organizations that support various implementations of NFS; IBM, Oracle, NetApp.

Distributed file systems other than the ones that we described in this chapter include the Lustre file system [126], GPFS [113], ANY OTHER. Lustre and GPFS are POSIX-compliant whereas GFS is not. For this reason, both of these also maintain and manage metadata that may pose to be a burden for systems such as GFS and Tectonic that are workload-oriented.

In addition to designs of distributed file systems, there are several studies on how to store and search the metadata of highly scalable file systems. Some of the include [85, 92, 105].

A noticeable feature among the distributed file system designs that we summarized in this chapter is extensibility. Extensible systems allow for adding more resources such as disk drives in a seamless and transparent manner.

In recent years, there is an emerging set of applications that are leading to rethink on distributed file system and storage system design. In particular, traditional file systems supported an in-place write operation that allows the user to pick the offset at which to write the desired contents. New generation applications such as email stores, photo stores, rarely need to support in-place writes. Rather, such applications benefit from an append operation like the record-append that GFS introduced. Having record-append allows the file system to also order the updates to achieve consistency across multiple concurrent writes in addition to making it easier to version control. In addition, with these client facing applications, durability, availability, and consistency guarantees take higher priority.

Another noticeable difference in the way current generation users interact with file systems. Unlike the advanced users of the yesteryears who needed to interface with distributed file systems via operations such as mount. The current trend is to make the file system completely transparent to the user.

Coupled with the diminishing cost of storage per byte, some of the learnings with respect to system optimizations from the traditional era with respect to internal fragmentation and the like start to lose significance.

These changes in the conventional wisdom to new wisdom open up multiple opportunities in the design and implementation of distributed file and storage systems. Some of the insights into the design of highly scalable distributed file systems such as Colossus and Tectonic is at `https://queue.acm.org/detail.cfm?id=1594206` and also at `https://paulcavallaro.com/blog/facebook-tectonic-filesystem/`, respectively.

Questions

Question 8.1. Study the data structures that a typical Unix file system maintains. Assuming a block size of 2 KB, find the largest size a file can have in such a file system.

Question 8.2. Compare and contrast between the various RAID levels