Ihor Kuz, Felix Rauch, Manuel M. T. Chakravarty & Gernot Heiser

Distributed File Systems

In a distributed file system (DFS), multiple clients share files provided by a shared file system. In the DFS paradigm communication between processes is done using these shared files. Although this is similar to the DSM and distributed object paradigms (in that communication is abstracted by shared resources) a major difference between these paradigms and the DFS paradigm is that the resources (files) in DFS are much longer lived. This makes it, for example, much easier to provide asynchronous and persistent communication using shared files than using DSM or distributed objects.

The basic model provided by distributed file systems is that of clients accessing files and directories that are provided by one or more file servers. A file server provides a client with a file service interface and a view of the file system. Note that the view provided to different clients by the same server may be different, for example, if clients only see files that they are authorised to access. Access to files is achieved by clients performing operations from the file service interface (such as create, delete, read, write, etc.) on a file server. Depending on the implementation the operations may be executed by the servers on the actual files, or by the client on local copies of the file. We will return to this issue later.

The key challenges of distributed systems apply to DFS as follows:

Transparency:

- Location: a client cannot tell where a file is located
- Migration: a file can transparently move to another server
- Replication: multiple copies of a file may exist
- Concurrency: multiple clients access the same file

Flexibility: In a flexible DFS it must be possible to add or replace file servers. Also, a DFS should support multiple underlying file system types (e.g., various UNIX file systems, various Windows file systems, etc.)

Reliability:

- Consistency: employing replication and allowing concurrent access to files may introduce consistency problems.
- Security: clients must authenticate themselves and servers must determine whether clients are authorised to perform requested operation. Furthermore communication between clients and the file server must be secured.
- Fault tolerance: clients should be able to continue working if a file server crashes. Likewise, data must not be lost and a restarted file server must be able to recover to a valid state.

Performance: In order for a DFS to offer good performance it may be necessary to distribute requests across multiple servers. Multiple servers may also be required if the amount of data stored by a file system is very large.

Scalability: A scalable DFS will avoid centralised components such as a centralised naming service, a centralised locking facility, and a centralised file store. A scalable DFS must be able to handle an increasing number of files and users. It must also be able to handle growth over a geographic area (e.g., clients that are widely spread over the world), as well as clients from different administrative domains.

Client's Perspective: File Services

The *File Service Interface* represents files as an uninterpreted sequence of bytes that are associated with a set of attributes (owner, size, creation date, permissions, etc.) including information regarding protection (i.e., access control lists or capabilities of clients). Moreover, there is a choice between the *Upload/download model* and the *remote access model*. In the first model, files are downloaded from the server to the client. Modifications are performed directly at the client after which the file is uploaded back to the server. In the second model all operations are performed at the server itself, with clients simply sending commands to the server.

There are benefits and drawbacks to both models. The first model, for example, can avoid generating traffic every time it performs operations on a file. Also, a client can potentially use a file even if it cannot access the file server. A drawback of performing operations locally and then sending an updated file back to the server is that concurrent modification of a file by different clients can cause problems. The second approach makes it possible for the file server to order all operations and therefore allow concurrent modifications to the files. A drawback is that the client can only use files if it has contact with the file server. If the file server goes down, or the network connection is broken, then the client loses access to the files.

File Access Semantics

Ideally, the client would perceive remote files just like local ones. Unfortunately, the distributed nature of a DFS makes this goal hard to achieve. In the following we present the various file access semantics available, and discuss how appropriate they are to a DFS.

The first type of access semantics that we consider are called UNIX *semantics* and they imply the following:

- A READ after a WRITE returns the value just written.

- When two WRITEs follow in quick succession, the second persists.

In the case of a DFS, it is possible to achieve such semantics if there is only a single file server and no client-side caching is used. In practice, such a system is unrealistic because caches are needed for performance and write-through caches (which would make UNIX semantics possible to combine with caching) are expensive. Furthermore deploying only a single single file server is bad for scalability. Because of this it is impossible to achieve UNIX semantics with distributed file systems.

Alternative semantic models that are better suited for a distributed implementation include:

- session semantics,

- immutable files, and

- atomic transactions.

Session Semantics. In the case of session semantics, changes to an open file are only locally visible. Only after a file is closed, are changes propagated to the server (and other clients). This raises the issue of what happens if two clients modify the same file simultaneously. It is generally up to the server to resolve conflicts and merge the changes. Another problem with session semantics is that parent and child processes cannot share file pointers if they are running on different machines.

Immutable Files. Immutable files cannot be altered after they have been closed. In order to change a file, instead of overwriting the contents of the existing file a new file must be created. This file may then replace the old one as a whole. This approach to modifying files does require that directories (unlike files) be updatable. Problems with this approach include a race condition when two clients try to replace the same file as well as the question of what to do with processes that are reading a file at the same time as it is being replaced by another process.

Atomic Transactions.   In the transaction model, a sequence of file manipulations can be executed indivisibly, which implies that two transactions can never interfere. This is the standard model for databases, but it is expensive to implement.

Server's Perspective: Implementation

Observations about the expected use of a file system can be used to guide the design of a DFS. For example, a study by Satyanarayanan found the following usage patterns for UNIX systems at a university:

- Most files are small—less than 10k

- Reading is much more common than writing

- Usually access is sequential; random access is rare

- Most files have a short lifetime

- File sharing is unusual

- Most process use only a few files

- Distinct files classes with different properties exist

These usage patterns (small files, sequential access, high read-write ratio) would suggest that an update/download model for a DFS would be appropriate. Note, however, that different usage patterns may be observed at different kinds of institutions. In situations where the files are large, and are updated more often it may make more sense to use a DFS that implements a remote access model.

Besides the usage characteristics, implementation tradeoffs may depend on the requirements of a DFS. These include supporting a large file system, supporting many users, the need for high performance, and the need for fault tolerance. Thus, for example, a fault tolerant DFS may sacrifice some performance for better reliability guarantees, while a high performance DFS may sacrifice security and wide-area scalability in order to achieve extra performance.

Stateful vs Stateless Servers

The file servers that implement a distributed file service can be *stateless* or *stateful*. Stateless file servers do not store any session state. This means that every client request is treated independently, and not as part of a new or existing session. Stateful servers, on the other hand, do store session state. They may, therefore, keep track of which clients have opened which files, current read and write pointers for files, which files have been locked by which clients, etc.

The main advantage of stateless servers is that they can easily recover from failure. Because there is no state that must be restored, a failed server can simply restart after a crash and immediately provide services to clients as though nothing happened. Furthermore, if clients crash the server is not stuck with abandoned opened or locked files. Another benefit is that the server implementation remains simple because it does not have to implement the state accounting associated with opening, closing, and locking of files.

The main advantage of stateful servers, on the other hand, is that they can provide better performance for clients. Because clients do not have to provide full file information every time they perform an operation, the size of messages to and from the server can be significantly decreased. Likewise the server can make use of knowledge of access patterns to perform read-ahead and do other optimisations. Stateful servers can also offer clients extra services such as file locking, and remember read an write positions.

Replication

The main approach to improving the performance and fault tolerance of a DFS is to replicate its content. A *replicating* DFS maintains multiple copies of files on different servers. This can prevent data loss, protect a system against down time of a single server, and distribute the overall workload.

There are three approaches to replication in a DFS:

Explicit replication : The client explicitly writes files to multiple servers. This approach requires explicit support from the client and does not provide transparency.

Lazy file replication : The server automatically copies files to other servers after the files are written. Remote files are only brought up to date when the files are sent to the server. How often this happens is up to the implementation and affects the consistency of the file state.

Group file replication : WRITE requests are simultaneously sent to a group of servers. This keeps all the replicas up to date, and allows clients to read consistent file state from any replica.

Caching

Besides replication, *caching* is often used to improve the performance of a DFS. In a DFS caching involves storing either a whole file, or the results of file service operations. Caching can be performed at two locations: at the server and at the client. Server-side caching makes use of file caching provided by the host operating system. This is transparent to the server and helps to improve the server's performance by reducing costly disk accesses. Client-side caching comes in two flavours: on-disk caching, and in-memory caching. On-disk caching involves the creation of (temporary) files on the client's disk. These can either be complete files (as in the upload/download model) or they can contain partial file state, attributes, etc. In-memory caching stores the results of requests in the client-machine's memory. This can be process-local (in the client process), in the kernel, or in a separate dedicated caching process.

The issue of cache consistency in DFS has obvious parallels to the consistency issue in shared-memory systems, but there are other tradeoffs (for example, disk access delays come into play, the granularity of sharing is different, sizes are different, etc.). Furthermore, because write-through caches are too expensive to be useful, the consistency of caches will be weakened. This makes implementing UNIX semantics impossible. Approaches used in DFS caches include, *delayed writes* where writes are not propagated to the server immediately, but in the background later on, and *write-on-close* where the server receives updates only after the file is closed. Adding a delay to write-on-close has the benefit of avoiding superfluous writes if a file is deleted shortly after it has been closed.

Example: Network File System (NFS)

NFS is a remote access DFS that was introduced by Sun in 1985. The currently used version is version 3, however a new version (4) has also been defined. NFS integrates well into UNIX's model of mount points, but does *not* implement UNIX semantics. NFS servers are stateless (i.e., NFS does not provide OPEN & CLOSE operations). It supports caching, but no replication.

NFS has been ported to many platforms and, because the NFS protocol is independent of the underlying file system, supports many different underlying file systems. On UNIX, an NFS server runs as a daemon and reads the file `/etc/export` to determine what directories are exported to whom under which policy (for example, who is allowed to mount them, who is allowed to access them, etc.). Server-side caching makes use of file caching as provided by the underlying operating system and is, therefore, transparent.

On the client side, exported file systems can be explicitly mounted or mounted on demand (called *automounting*). NFS can be used on diskless workstations so does not require local disk space for caching files. It does, however, support client-side caching, and allows both file contents

as well as file attributes to be cached. Although NFS allows caching, it leaves the specifics up to the implementation. As such, file caching details are implementation specific. Cache entries are generally discarded after a fixed period of time and a form of delayed write-through is employed.

Traditionally, NFS trusts clients and servers and thus has only minimal security mechanisms in place. Typically, the clients simply pass UNIXuser ID and group ID of the process performing a request to the server. This implies that NFS users must not have root access on the clients, otherwise they could simply switch their identity to that of another user and then access that user's files. New security mechanisms have been put in place, but they also have their drawbacks: Secure NFS using Diffie-Hellman public key cryptography is more complex to implement and to manage the keys, and the key lengths used are too short to provide security in practice. Using Kerberos would turn NFS more secure, but it has high entry costs.

Please read [PJS+94] for more detailed information about NFS and the NFS protocol.

Example: Andrew File System (AFS)

The Andrew File System (AFS) is a DFS that came out of the Andrew research project at Carnegie Mellon University (CMU). Its goal was to develop a DFS that would scale to all computers on the university's campus. It was further developed into a commercial product and an open-source branch was later released under the name "OpenAFS". AFS offers the same API as UNIX, implements UNIX semantics for processes on the same machine, but implements write-on-close semantics globally. All data in AFS is mounted in the /afs directory and organised in *cells* (e.g. /afs/cs.cmu.edu). Cells are administrative units that manage users and servers.

Files and directories are stored on a collection of trusted servers called *Vice*. Client processes accessing AFS are redirected by the file system layer to a local user-level process called *Venus* (the AFS daemon), which then connects to the servers. The servers serve whole files, which are cached as a whole on the clients' local disks. For cached files a callback is installed on the corresponding server. After a process finishes modifying a file by closing it, the changes are written back to the server. The server then uses the callbacks to invalidate the file in other clients' caches. As a result, clients do not have to validate cached files on access (except after a reboot) and hence there is only very little cache validation traffic. Data is stored on flexible *volumes*, which can be resized and moved between the servers of a cell. Volumes can be marked as read only, e.g. for software installations.

AFS does not trust UNIX user IDs and instead uses its own IDs which are managed at a cell level. Users have to authenticate with Kerberos by using the klog command. On successful authentication, a token will be installed in the client's cache managers. When a process tries to access a file, the cache manager checks if there is a valid token and enforces the access rights. Tokens have a timp stamp and expire, so users have to renew their token from time to time. Authorisation is implemented by directory-based ACLs, which allow finer grained access rights than UNIX.

More details about AFS can be found in [HKM+88].

Example: Coda

Coda is an experimental DFS developed at CMU by M. Satyanarayanan's group, it is the successor of the Andrew File System (AFS) but supports disconnected, mobile operation of clients. Its design is much more ambitious than that of NFS.

Coda has quite a number of similarities with AFS. On the client side, there is only a single mount point /coda. This means that the name space appears the same to all clients (and files therefore have the same name at all clients). File names are location transparent (servers cannot be distinguished). Coda provides client-side caching of whole files. The caching is implemented in a user-level cache process called *Venus*. Coda provides UNIX semantics for files shared by processes on one machine, but applies write-on-close (session) semantics globally. Because high availability is one of Coda's goals access to a cached copy of a file is only denied if it is known to be inconsistent.

In contrast to AFS, Coda supports disconnected operation, which works as follows. While disconnected (a client is disconnected with regards to a file if it cannot contact any servers that serve copies of that file) all updates are logged in a *client modification log (CML)*. Upon reconnection, the operations registered in the CML are replayed on the server. In order to allow clients to work in disconnected mode, Coda tries to make sure that a client always has up-to-date cached copies of files that they might require. This process is called *file hoarding*. The system builds a user hoard database which it uses to update frequently used files using a process called a *hoard walk*. Conflicts upon reconnection are resolved automatically where possible, otherwise, manual intervention becomes necessary.

Files in Coda are organised in organisational units called *volumes*. A volume is a small logical unit of files (e.g., the home directory of a user or the source tree of a program). Volumes can be mounted anywhere below the `/coda` mount point (in particular, within other volumes).

Coda allows files to be replicated on read/write servers. Replication is organised on a per volume basis, that is, the unit of replication is the volume. Updates are sent to all replicas simultaneously using multicast RPCs (Coda defines its own RPC protocol that includes a multicast RPC protocol). READ operations can be performed at any replica.

A more detailed overview of Coda (focusing, in particular, on disconnected operations) can be found in [KS91]. A lighter overview of Coda from the user's (i.e., client's) perspective is also available [Bra98].

Example: Google File System

The Google File System (GFS) is a distributed file system developed to support a system with very different requirements than traditionally assumed when developing file systems. GFS was designed and built to support operations (both production and research) at Google that typically involve large amounts of data, run distributed over very large clusters, and include much concurrent access to files. GFS assumes that most data operations are large sequential reads and large concurrent appends. One of the key assumptions driving the design is that, because very large clusters (built from commodity parts) are used, failure (of hardware or software resulting in crashes or corrupt data) is a regular occurence rather than an anomaly. A good overview of the design of GFS is provided in [GGL03]. A discussion of how the workloads for GFS have changed leading to different requirements is provided in [MQ09].

References

[Bra98]   Peter J. Braam. The Coda distributed file system. *Linux Journal*, (50), June 1998.

[GGL03]   Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on OS Principles (SOSP)*, Lake George, NY, USA, October 2003.

[HKM+88]   John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6:51–81, 1988.

[KS91]   James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, Pacific Grove, CA, USA, October 1991.

[MQ09]   Marshall Kirk McKusick and Sean Quinlan. Gfs: Evolution on fast-forward. *ACM Queue*, 7(7), August 2009.

[PJS+94]   Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, and Dave Hitz. NFS version 3: Design and implementation. In *Proceedings of the USENIX Summer 1994 Technical Conference*, Boston, MA, USA, June 1994.