Contents

A Survey on General and Special Purpose Filesystems

Junlong Gao

The concept, design, implementation, and use cases of file systems has long been challenged by a more structured approach of data like SQL databases and key-value stores. Now, with the rise of object (blob) storage and block storage (virtual SAN), its use cases are further being reconsidered and new variants of file systems are implemented for specific tasks. This article explores various modern filesystems and their similar storage system solutions, both in terms of their design considerations, implementation variations, and the use cases they are targeting for.

I Filesystems = Namespaces Management + Data Management

What exactly are filesystems? Their essence are brought out only after their replacements and variants are implemented. Consider Haystack, an implementation for photo storage at Facebook [1]. They moved from their initial NFS based implementation to their home-brew filesystem-like storage system, namely Haystack, only to eliminate the namespace management provided by the backing filesystems in NFS. If we look at a more traditional general-purpose file system like Ceph, their paper cautiously admits its data allocation scheme in fact reduced much of the system complexity [8]. Yet, their metadata system design (namespace and its operations) are still subject to the complexity of filesystem tree sharding/partitioning (scaling the namespace).

A tree-like (DAG, considering hardlinks) namespace structure, and various operations on them like atomic, cyclic-free renames are challenges in its own right for implementing a scalable, robust namespace storage. Failure modes are more complicated, and it is harder to implement concurrency. Once this is solved, data management

itself is also challenging as files are essentially a map from logical offset to data. Fragmentation, small IO, concurrency and consistency in concurrent read/write access, all brings even the most rudimentary design to its knees. To be POSIX-like filesystems with large scale, most of the implementations [4, 7, 8] opts-in comprises like reduced POSIX guarantees, and large units of data allocation to reduce mapping tables, or even no allocation table at all [9].

These trade-offs are made to pave ways for scalability, yet harder problems are yet to be solved. Both namespace and data have to be protected against failure, and implementations of file systems are usually interleaved with failure detection and data redundancy. In GFS and HDFS, writes are required to participate in the replication protocol and have to deal with failures. Ceph took a similar approach yet provides a way to acknowledge writes waiting for propagation of all of the replication. The difference is how Ceph explicitly took the system into 2 layers: the upper layer is the namespace management and the lower layer is the object store (Reliable Autonomic Distributed Object Store) which is built on a collection of plain files. In this way, Ceph could worry about the scalability of the data and metadata (namespace) separately, and build reliability in each layer (RADOS does the underlying replication for data).

Yet with the rise of the object store, it seems apparent that the complicated tree-like namespace hierarchy in a distributed storage system is more of a burden than a help. Where are the namespaces? Photos names or video names are structured inside a different store: the databases. They are not filenames within a folder, rather are column fields in a SQL schema. Along with other application-specific metadata, names are no longer part of the data storage.

Sequential read and writes are not only best for throughput, but also an advantage in storage wearing, metadata management (structure sizing and caching), and reducing fragmentation. Workloads in bulks are always better than random, small frequent operations. Applications like databases usually uses various techniques like write-ahead-logging with `fsync(2)` and preallocation (`fallocate(1)`) for its storage access: it somehow has its own management of data, and only needs a collection of files for namespace purposes. On the other hand, thin provisioning, or sparse files, provides a great level of flexibility as provisioning total space usage before hand is hard and sometimes expensive. A good design should trade-off between the granularity of metadata and the flexibility of thin provisioning.

II Space Allocations

The ideal workload for a file system in a large scale, distributed manner, is to have IO access with good space locality (sequential, or in batch). Any such file system implementing data management will have metadata to find where is the actual physical storage of that piece of data. There are 3 questions around the organization of this metadata:

- 1) What is the space granularity in terms of allocation?
- 2) What is the write pattern for new spaces and overwrites?
- 3) How will space get freed?

In short, the discussion of space allocation is the discussion of data lifecycle, which is an integral part of space management.

- Let's start with the classic Google File System. It is more of a special-purpose filesystem than a general-purpose one. Its assumption is large, batched and sequential append, and large streaming reads [4]. In this light, space granularity is generously large static-sized 64MB chunks. New space is resulted in append records to the file with aggressive client-side batching and retry. This also makes space management easy, as chunk size is both large and fixed, resulting in a simple table-based approach, mapping file offsets to chunk locators.
- A modern design like PolarFS [2] takes the idea in GFS and advances it further. Data are stored in the unit of chunks and chunks are of 10GB sizing. A filesystem volume can contain multiple chunks for scaling data, and each chunk has a super block and a space allocation table, which maps:

  $$\texttt{chunk block offset} \rightarrow (\texttt{file inode}, \texttt{file offset})$$

  This allows each chunk to individually handle space allocation, and effectively shards the entire data management acrosses chunks. Of course, random writes and unmap will also be horrible in this schema. This mapping table size is governed by the physical granularity: 64KB. So a 10GB chunk can have 128k entries at most, which fits in memory comfortably. Yet PolarFS is designed to support MySQL database workload, and database applications will always preallocate files and journals and effectively manage their data in these large files on their own.

- Ceph intentionally divides the whole filesystem into two layers: the metadata store MDS and the underlying storage devices ODS (object device store). ODS is implemented as RADOS (Reliable Autonomic Distributed Object Store), which handles most of the data allocation requests. Clients buffer write operations aggressively (using a form of lease called "Capability Bits" [8]) and generate large-sized of objects. The file layout is from file offset to object ID. Finally, a sequence of placement and replication layer will assign an ODS node in RADOS for storing that object. Then we have 2 types of mappings, first is handled by MDS layer (`file offset` to `object ID`), and the second, `object ID` to `OSD offset`, is handled in the RADOS layer, both allocation and locating. The paper went on describing EBOFS for implementing each OSD, which is a plain large file in nodes' local EBOFS. I assume it has a table or tree-ish structure per file to translate and allocate objects.

- Finally, consider HayStack, evolved from load-balanced, NAS-based NFS [1]. The key idea in HayStack is it is an object store with a relatively flat namespace (since, structure information of the photos are captured in a different database), writes are always new writes, no overwrite of existing data but lots of random reads, and finally, there is no concept of file offset: a file is a photo and its data in entirety. The difficulties of the original NFS solution is the "dilemma of the right RAM-to-disk ratio". Still, all metadata must be able to reside in memory for serving random, long-tailed read requests. Data are appended in a store and a store can be thought as a large file with some index structure in memory for management of data. Allocation of data in the store is a flat table small enough to reside in memory, and new write always appends, both data and metadata. Read thus can be served with 1 disk IO using the in-memory table to find the offset in the store. Finally, such an append-only structure requires compaction for cleaning up deleted files. This is effectively a log-structured filesystem [6].

III IO Consistency and Metadata Sychonization

The file-based abstraction provided by a filesystem is also responsible for implementing sharing across different users. The other aspect, durability, can also be thought as a form of sharing: sharing with itself across different time points, while concurrent access is sharing with others at the same time. Any form of sharing effectively implements a communication channel. Indeed, local filesystems in UNIX-like kernels are heavily

used as a form of IPC (inter-process communication).

The semantics of concurrent read-write to a file defines what it means to be shared, yet the most straightforward, layman's definition, is "reads must reflect the latest write". On the other hand, sometimes the workload does not have this use-case: it does not care what happens the same file is shared, as the workload is inherently conflict-free (used as a producer-consumer queue like GFS), or exclusive access (photos in Haystack cannot be read after written, and immutable after written), or single process access with its own concurrency control (MySQL on top of PolarFS). Nevertheless, it is still worthwhile to discuss what are the consistency levels and how they are implemented. Note in a distributed file system with standard failure model, concurrency control of file access with replication is as hard as distributed locking and subjects to CAP theorem [5].

- Ceph implements a form of range locking over objects and use a lease-like approach for read-write synchronization. By default clients are encouraged to buffer writes and cache reads aggressively, using the "Capability Bits" provided from MDS as a form of lease. The consistency issue of these caching is solved by revoking the read/write permission in the bits when the MDS detects a mixed reader-writers have opened the same file. Writes to objects are controlledd by object locks, and buffering and caching will turn into synchronized operations after lease is revoked. Since Ceph is a general-purpose filesystem, it must favor these consistency guarantees in order to not break POSIX semantics. On the other hand, it does support `O_LAZY` flag with syscalls to manually trigger caching and buffering, if applications neeed.

- GFS has to cope with its replication protocol. In GFS, the replication protocol of chunks (chunkserver redundancy) is explicit in the datapath, unlike PolarFS or Ceph. This opens a range of challenges, as clients have to deal with failed replications (half write and resync), in-order delivery (data can be applied, like append record, in different order across the replicas), duplication of data (retries) and placement decisions (should I ask for a new replica from the name node or should I wait?). Yet, writes are linearizable in the same chunk when there is no failure. This is achieved via lease-based locking: a single chunk server will be picked as the primary, and the client will push data to all replicas before talking to the primary. This push will not result in written-data being visible until the client commits it in the primary node. The primary node gathers all the different clients' mutation requests, assign them serial numbers, and ask the replicas

for each request to apply them in the serial order, hence achieves linearizable reads. However, when write requests (offset + size) are across multiple chunk boundaries, this can result in non-atomic write or interleaved with other append requests. Record append can be done in an atomic manner, but limited to the chunk size. This is less ideal compared to the modern replicated state machine approach.

- PolarFS uses a journal file (implemented by the Disk Paxos protocol [3]) to serialize metadata mutation (like punching holes in file or preallocation) and implement coordination. When node's journal anchor changes, writes will be able to refresh its state of the metadata. These only serializes "new writes", majority of which belong to MySQL preallocated database files. Readers does not need this synchronization, and readers are usually slave instances of MySQL databases for scaling reads.

IV Layered Design and Data Placement

Subsequent works after GFS takes the approach of separation of concerns. PolarFS and Ceph intentionally separates the data path and the replication protocol. The lower layer builds the abstraction of large, chunk-based storage service. This service is fault tolerant in the sense that it implements its own redundancy, without the upper layer's coordination. Upper layer can focus on data placement, sharding, metadata scale-out or synchronization, and leave the complicated consistency issue of data replication to the lower layer.

- Ceph uses its home-brew replication protocol. Data writes are journaled first in RADOS, and OSD will sync with each other on failure recovery. The cluster state is updated via heartbeats from OSD self-report, and controls data placement in terms of object ID. The most significant difference between Ceph and other distributed system is placement is not persisted, yet calculated by data signature and cluster state. There is no centralized service for looking up where is the data, yet still scale-out its capacity. I have read the failure detection section and recovery section closely, but I still believe it is less ideal than a more established approaching using replicated logs and a centralized service.

- PolarFS does the placement and replication in the lower layer. Data writes into chunk are replicated using its own proprietary *ParallelRaft* protocol. Placement is

in the unit of chunks, and it has its control plane for the chunk layer to dynamically move chunks around to mitigate hot spot issues.

- GFS does not have this layered approach. The datapath is fully aware the replication complexity and has to ask for replica and placement explicitly. GFS also does explicit chunk server movement and increase the number of replica to handle hot spot issues, and implements client side exponential random backoff to mitigate the thundering herd problem. All the placement and replication decisions can be done in the name node, which collects chunkserver heartbeats and scans for failures or latency/capacity issues.
- HayStack has its own directory service for placement decisions, and uses RAID for storage fault-tolerance. Proactive health check for storage layer access (*PitchFork* [1]).

V Namespace Operations

In work.

References

[1] Doug Beaver, Sanjeev Kumar, Harry C Li, Jason Sobel, Peter Vajgel, et al. Finding a needle in haystack: Facebook's photo storage. In *OSDI*, volume 10, pages 1–8, 2010.

[2] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. Polarfs: an ultra-low latency and failure resilient distributed file system for shared storage cloud database. *Proceedings of the VLDB Endowment*, 11(12):1849–1862, 2018.

[3] Eli Gafni and Leslie Lamport. Disk paxos. *Distributed Computing*, 16(1):1–20, 2003.

[4] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.

[5] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News*, 33(2):51–59, 2002.

[6] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[7] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. Ieee, 2010.

[8] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, 2006.

[9] Sage A Weil, Scott A Brandt, Ethan L Miller, and Carlos Maltzahn. Crush: Controlled, scalable, decentralized placement of replicated data. In *SC'06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, pages 31–31. IEEE, 2006.