Design Notes

Design and implementation

The 3FS system has four components: cluster manager, metadata service, storage service and client. All components are connected in an RDMA network (InfiniBand or RoCE).

Metadata and storage services send heartbeats to cluster manager. Cluster manager handles membership changes and distributes cluster configuration to other services and clients. Multiple cluster managers are deployed and one of them is elected as the primary. Another manager is promoted as primary when the primary fails. Cluster configuration is typically stored in a reliable distributed coordination service, such as ZooKeeper or etcd. In our production environment, we use the same key-value store as file metadata to reduce dependencies.

File metadata operations (e.g. open or create files/directories) are sent to metadata services, which implement the file system semantics. Metadata services are stateless, since file metadata are stored in a transactional key-value store (e.g. FoundationDB). Clients can connect to any metadata service.

Each storage service manages a few local SSDs and provides a chunk store interface. The storage service implements Chain Replication with Apportioned Queries (CRAQ) to ensure strong consistency. CRAQ's write-all-read-any approach helps to unleash the throughput of SSDs and RDMA network. A 3FS file is split into equally sized chunks, which are replicated over multiple SSDs.

Two clients are developed for applications: FUSE client and native client. Most applications use FUSE client, which has a low adoption barrier. Performance-critical applications are integrated with the native client.

File system interfaces

Object store is becoming a popular option for data analytics and machine learning. However, file system semantics and a unified namespace where files are organized in directories provide greater flexibility for applications.

- *Atomic directory manipulation* An object store can approximate hierarchical directory structures by using slashes (/) in object keys. However, it doesn't natively support operations like atomically moving files/directories, or recursively deleting entire directories. Actually a common pattern in our internal applications involves creating a temporary directory, writing files to it, and then moving the directory to its final location. When handling a large number of small files, the recursive delete for directories is crucial. Without it, applications have to traverse each directory and remove files one by one.

- *Symbolic and hard links* Our applications utilize symbolic and hard links to create lightweight snapshots of dynamically updated datasets, where new data is appended as individual files.

- *Familiar interface* The file interface is well known and used everywhere. There is no need to learn a new storage API. Many datasets are stored as CSV/Parquet files. Adapting file-based data loaders to use the 3FS FUSE client or native client is straightforward.

Limitations of FUSE

FUSE (Filesystem in Userspace) simplifies file system client development by redirecting I/O operations to user-space processes through the FUSE kernel module. It creates the illusion that applications are accessing the remote file system as if it were a local file system. However, it has performance limitations:

- *Memory copy overhead* The user-space file system daemon cannot access application memory. Data transfer between kernel and user spaces consumes memory bandwidth and increases end-to-end latency.

- *Primitive multi-threading support* When an application initiates I/O requests, FUSE places these requests into a multi-threaded shared queue, protected by a spin lock. The user-space file system daemon then retrieves and processes requests from this queue. Due to lock contention, FUSE's I/O processing capability fails to scale with the number of threads. Our benchmark results indicate that FUSE only handles approximately 400K 4KiB reads per second. Further increasing concurrency does not improve performance as lock contention intensifies. `perf` profiling reveals that the kernel-space spin lock consumes a significant amount of CPU time.

Most applications, e.g. data analytics, perform large block writes on 3FS or they can buffer data in memory and flush it to 3FS when write buffer is full. However, FUSE on Linux 5.x does not support concurrent writes to the same file[^1]. Applications overcome this limitation by writing to multiple files concurrently, maximizing the total throughput.

Read operations exhibit more complex patterns. Some training jobs require random access to dataset samples, with read sizes varying from a few kilobytes to several megabytes per sample. And samples are typically not 4K-aligned in files. Data loaders are specifically designed to fetch batches of samples. But they perform poorly when handling small random reads on FUSE-mounted 3FS. Bandwidth of SSDs and RDMA network are not fully utilized.

Asynchronous zero-copy API

Implementing the file system client as a VFS kernel module avoids performance issues mentioned above. But kernel module development is significantly more challenging than user-space system programming. Bugs are difficult to diagnose and can lead to catastrophic failures in production environments. For example, machines may crash and leave no log message for debugging. When upgrading a kernel module, all processes using the file system must be stopped cleanly; otherwise, a machine restart is required.

For these reasons, we have chosen to implement a native client within the FUSE daemon. This client offers an interface that supports asynchronous zero-copy I/O operations. File

meta operations are still handled by FUSE daemon (e.g. open/close/stat files). Applications call `open()` to obtain a file descriptor (fd) and register it via native API. They can then perform I/O operations on the file with native client. This approach ensures consistency in metadata operations with the POSIX API, making it easier to migrate existing code.

The asynchronous, zero-copy API is inspired by Linux `io_uring`. Below are the key data structures in the API:

- *Iov* A large memory region for zero-copy read/write operations, shared between the user process and the native client. InfiniBand memory registration is managed by the client. In native API, all read data will be read into Iov, and all write data should be written to Iov before calling the API.

- *Ior* A small shared ring buffer for communication between user process and native client. The usage of Ior is similar to Linux `io_uring`, where the user process enqueues read/write requests, and the native client dequeues these requests for completion. The requests are executed in batches, with their sizes controlled by the `io_depth` parameter. Multiple batches are processed in parallel, whether from different rings or the same ring. However, multiple rings are still recommended for multi-threaded applications, as sharing a ring requires synchronization, which can impact performance.

Within the native client, multiple threads are spawned to fetch I/O requests from the Iors. These requests are batched and dispatched to storage services, reducing RPC overhead caused by small read requests.

File metadata store

Location of file chunks

3FS divides file data into equally sized chunks and stripes them across multiple replication chains (replication chains and chain tables are defined in Section [Data placement](#data-

placement)). Users can specify the chain table, chunk size, and stripe size for files on a per-directory basis. Each chunk is independently stored on multiple storage services, with its chunk ID generated by concatenating the file's inode id and chunk index.

When creating a new file, the metadata service employs a round-robin strategy to select consecutive replication chains from the designated chain table, based on the stripe size. Next, a random seed is generated to shuffle the selected chains. This allocation strategy ensures balanced data distribution across chains and SSDs.

When an application opens a file, the client contacts the meta service to obtain the file's data layout information. Then the client can independently compute chunk IDs and chains for data operations, minimizing the involvement of the meta service in the critical path.

File metadata on transactional key-value store

3FS uses FoundationDB as its distributed storage system for metadata. FoundationDB provides a key-value store interface and supports transactions with Serializable Snapshot Isolation (SSI). 3FS stores all metadata as key-value pairs in FoundationDB. Meta services follow a stateless architecture, greatly enhancing maintainability by allowing administrators to seamlessly upgrade or restart services without disruption. When clients experience request failures or timeouts, they can automatically fail over to other available services.

The file system metadata primarily consists of two core structures: inodes and directory entries. Inodes store attribute information for files, directories, and symbolic links, each identified by a globally unique 64-bit identifier that increments monotonically. Inode keys are constructed by concatenating the "INOD" prefix with the inode id, which is encoded in little-endian byte order to spread inodes over multiple FoundationDB nodes. The inode values vary by its type:

- All inode types contain basic attributes: ownership, permissions, access/modification/change times.

- Additional attributes for file inodes: file length, chunk size, selected range in chain table, shuffle seed.

- Additional attributes for directory inodes: the parent directory's inode id, default layout configurations for subdirectories/files (chain table, chunk size, stripe size). The parent's inode id is required to detect loops when moving directories. When moving `dir_a/dir_b` to `dir_c/`, we need to ensure that `dir_c` is not a descendant of `dir_b`, which can be achieved by checking all ancestors of `dir_c` upward.

- Additional attributes for symbolic link inodes: target path string.

Directory entry keys are composed of a "DENT" prefix, the parent inode ID, and the entry name. Directory entry values store the target inode id and inode type. All entries within a directory naturally form a contiguous key range, allowing efficient directory listing via range queries.

The meta operations leverage FoundationDB's transactions:

- Read-only transactions used for metadata queries: fstat, lookup, listdir etc.

- Read-write transactions used for metadata updates: create, link, unlink, rename etc.

For write transactions, FoundationDB tracks the read/write key sets to form conflict detection sets. When concurrent transaction conflicts are detected, the meta service automatically retries the transaction. This design enables multiple meta services to process requests in parallel while maintaining file system metadata consistency.

Dynamic file attributes

On most local file systems, deleting an opened file is deferred until all associated file descriptors are closed. Consequently, it is necessary to track all file descriptors of the file. Training jobs open a large number of files during startup. Storing all file descriptors would impose heavy load on meta service and FoundationDB. Since training jobs do not depend on this feature, 3FS does not track file descriptors opened in read-only mode.

3FS maintains a file session for each file descriptor (fd) opened in write mode since deleting write opened files may lead to unreclaimable garbage chunks from concurrent writes. When a file with active write sessions is deleted, meta service delays the deletion until all its fds are closed. To prevent lingering sessions from offline clients, the 3FS meta service periodically checks client liveness and cleans up sessions of offline clients.

The file length is stored in the inode. For files being actively updated, the length stored in inode may diverge from the actual length. Clients periodically (5 seconds by default) report to meta service maximum write position of each file opened in write mode. If this position exceeds the length in inode and there is no concurrent truncate operation, this position is adopted as the new file length.

Due to the possibility of concurrent writes from multiple clients, the method described above ensures only eventual consistency for file lengths. When processing close/fsync operations, the meta service obtains the precise file length by querying the ID and length of the last chunk from the storage service. Since file data is striped across multiple chains, this operation incurs non-negligible overhead.

Concurrent updates to the same file's length by multiple meta services may cause transaction conflicts and lead to repeated file length computation. To mitigate this, meta service distributes file length update tasks across multiple meta services using inode IDs and the rendezvous hash algorithm.

Our production environments use a large stripe size: 200. For small files, the number of chains containing file chunks is well below this number. The number of potentially used chains is stored in file inode and used as a hint when updating the length. It starts with an

initial value of 16 and is doubled each time additional file chunks are written to more chains. This allows us to avoid querying all 200 chains when updating lengths of small files. This optimization can also be extended to the deletion of small files.

Chunk storage system

The design goal of chunk storage system is to achieve the highest bandwidth possible even when there are storage medium failures. The read/write throughput of 3FS should scale linearly with the number of SSDs and bisection network bandwidth between clients and storage services. Applications access storage services in a locality-oblivious manner.

Data placement

Each file chunk is replicated over a chain of storage targets using chain replication with apportioned queries (CRAQ). In CRAQ write requests are sent to the head target and propagated along a chain. Read requests can be sent to any of the storage target. Usually the read traffic is evenly distributed among all targets in a chain for better load balance. Multiple storage targets are created on each SSD and the targets join different chains.

Suppose there are 6 nodes: A, B, C, D, E, F. Each node has 1 SSD. Create 5 storage targets on each SSD: 1, 2, ... 5. Then there are 30 targets in total: A1, A2, A3, ..., F5. If each chunk has 3 replicas, a chain table is constructed as follows.

Chain	Version	Target 1 (head)	Target 2	Target 3 (tail)
1	1	`A1`	`B1`	`C1`
2	1	`D1`	`E1`	`F1`
3	1	`A2`	`B2`	`C2`
4	1	`D2`	`E2`	`F2`

| 5 | 1 | `A3` | `B3` | `C3` |

| 6 | 1 | `D3` | `E3` | `F3` |

| 7 | 1 | `A4` | `B4` | `C4` |

| 8 | 1 | `D4` | `E4` | `F4` |

| 9 | 1 | `A5` | `B5` | `C5` |

| 10 | 1 | `D5` | `E5` | `F5` |

Each chain has a version number. The version number is incremented if the chain is changed (e.g. a storage target is offline). Only the primary cluster manager makes changes to chain tables.

A few chain tables can be constructed to support different data placement requirements. For example, two chain tables can be created, one for batch/offline jobs and another for online services. The two tables consist of storage targets on mutually exclusive nodes and SSDs.

Logically, the state of each chain changes independently. Each chain can be included in multiple chain tables. The concept of chain table is created to let metadata service pick a table for each file and stripe file chunks across chains in the table.

Balanced traffic during recovery

Suppose read traffic is evenly distributed among all storage targets in the above chain table. When A fails its read requests would be redirected to B and C. Under heavy load the read bandwidth of B, C is immediately saturated and B, C become the bottleneck of the entire system. Replacing a failed SSD and syncing data to the new SSD can take several hours. The read throughput is impaired during this period.

To reduce the performance impact, we can have more SSDs share the redirected traffic. In the following chain table, A is paired with every other SSDs. When A fails, each of the other SSDs receives 1/5 of A's read traffic.

Chain	Version	Target 1 (head)	Target 2	Target 3 (tail)
1	1	`B1`	`E1`	`F1`
2	1	`A1`	`B2`	`D1`
3	1	`A2`	`D2`	`F2`
4	1	`C1`	`D3`	`E2`
5	1	`A3`	`C2`	`F3`
6	1	`A4`	`B3`	`E3`
7	1	`B4`	`C3`	`F4`
8	1	`B5`	`C4`	`E4`
9	1	`A5`	`C5`	`D4`
10	1	`D5`	`E5`	`F5`

To achieve maximum read throughput during recovery, the load balance problem can be formulated as a balanced incomplete block design. The optimal solution is obtained by using integer programming solver.

Data replication

CRAQ is a write-all-read-any replication protocol optimized for read-heavy workloads. Utilizing read bandwidth of all replicas is critical to achieve highest read throughput in an all-flash storage system.

When a write request is received by a storage service, it goes through the following steps:

1. The service checks if the chain version in write request matches with the latest known version; reject the request if it's not. The write request could be sent by a client or a predecessor in the chain.

2. The service issues RDMA Read operations to pull write data. If the client/predecessor fails, the RDMA Read operations may time out and the write is aborted.

3. Once the write data is fetched into local memory buffer, a lock for the chunk to be updated is acquired from a lock manager. Concurrent writes to the same chunk are blocked. All writes are serialized at the head target.

4. The service reads the committed version of the chunk into memory, applies the update, and stores the updated chunk as a pending version. A storage target may store two versions of a chunk: a committed version and a pending version. Each version has a monotonically-increasing version number. The version numbers of committed version and pending versions are `v` and `u` respectively, and satisfy `u = v + 1`.

5. If the service is the tail, the committed version is atomically replaced by the pending version and an acknowledgment message is sent to the predecessor. Otherwise, the write request is forwarded to the successor. When the committed version is updated, the current chain version is stored as a field in the chunk metadata.

6. When an acknowledgment message arrives at a storage service, the service replaces the committed version with the pending version and continues to propagate the message to its predecessor. The local chunk lock is then released.

Suppose there are 3 targets in the chain: `A, B, C`. A write request has just entered step 5 at `A`. `A` forwards the request to successor `B`. Then `B` instantly fails and the forwarded write request is lost. When cluster manager detects `B`'s failure, it marks `B` as offline and moves it to the end of chain and broadcasts the updated chain table. Once `A` receives the latest chain table, it forwards the write request to the new successor `C`.

`C` may not receive the latest chain table yet and rejects the request. But `A` can keep forwarding the request to `C`. Eventually `C` gets the latest chain table and accepts the request.

When a read request arrives at a storage service:

1. When the service only has a committed version of the chunk, this version is returned to the client.

2. Unlike CRAQ, our implementation does not issue version query to the tail target. When there are both committed and pending versions, the service replies a special status code to notify the client. The client may wait for a short interval and retry. Or the client can issue a relaxed read request to get the pending version.

Failure detection

The cluster manager relies on heartbeats to detect fail-stop failures. Cluster manager declares a service failed if it does not receive heartbeats from it for a configurable interval (e.g. T seconds). A service stops processing requests and exits if it cannot communicate with cluster manager for T/2 seconds. The heartbeat can be seen as a request to *renew a lease* granted by the manager.

The metadata services are stateless. The list of online meta services provided by cluster manager is a simple service discovery mechanism that helps clients create connections to metadata services. If one meta service is down, the clients may switch to any other metadata service.

Cluster manager plays a more critical role in membership changes of storage services. It maintains a global view of chain tables and storage targets' states. Each storage target has a public state and a local state.

Public state indicates if it's ready to serve read requests and if write requests would be propagated to it. Public states are stored with chain tables and distributed to services and clients.

Public State	Read	Write	Notes
serving	Y	Y	service alive and serving client requests
syncing	N	Y	service alive and data recovery is in progress
waiting	N	N	service alive and data recovery not started yet
lastsrv	N	N	service down and it was the last serving target
offline	N	N	service down or storage medium failure

Local state is only known by storage services and cluster manager, and it's stored in the memory of cluster manager. If a storage target has medium failure, the related service sets the target's local state to offline in heartbeat. If a storage service is down, storage targets managed by the service are marked offline.

Local State	Notes
up-to-date	service alive and serving client requests
online	service alive and target in syncing or waiting state
offline	service down or storage medium failure

A storage target can change from one public state to another in response to the latest local state. The local state plays the role of a triggering event. The cluster manager periodically scans every chain and updates the public states of targets on the chain according to a state-transition table.

- The chain version is incremented if the chain is updated.

- If a storage target is marked offline, it's moved to the end of chain.

- If a storage service finds public state of any local storage target is lastsrv or offline, it exits immediately. The service may be isolated from the cluster manager by network partition error.

- Once the data recovery of a storage target in syncing state is completed, the storage service set the target's local state to up-to-date in subsequent heartbeat messages sent to cluster manager.

Local State	Current Public State	Predecessor's Public State	Next Public State
up-to-date	serving	(any)	serving
	syncing	(any)	serving
	waiting	(any)	waiting
	lastsrv	(any)	serving
	offline	(any)	waiting
online	serving	(any)	serving
	syncing	serving	syncing
		not serving	waiting
	waiting	serving	syncing
		not serving	waiting
	lastsrv	(any)	serving
	offline	(any)	waiting
offline	serving	has no predecessor	lastsrv

		has predecessor	offline
	syncing	(any)	offline
	waiting	(any)	offline
	lastsrv	(any)	lastsrv
	offline	(any)	offline

Data recovery

When a storage service exits (e.g. process crashes or restarts during upgrade), or a storage medium failure occurs, all related storage targets will be marked as offline and moved to the end of chains by cluster manager. Once the service restarts, each target on the service enters into the recovery process independently. The entire recovery process overlaps with normal activity and minimizes any interruption.

When a previously offline storage service starts:

1. The service periodically pulls latest chain tables from cluster manager. But it does not send heartbeats until all its storage targets have been marked offline in the latest chain tables. This ensures all its targets would go through the data recovery process.

2. When a write request arrives during recovery, the request is always a full-chunk-replace write. The local committed version is updated and any existing pending version is abandoned. Since current service is the tail, an acknowledgment message is sent to the predecessor. The full state of the predecessor is copied to the returning service through a continuous stream of full-chunk-replace writes.

3. Before the data recovery of a storage target starts, the predecessor sends a dump-chunkmeta request to the returning service. Then the service iterates the local chunk metadata store to collect the ids, chain versions and committed/pending version numbers of all chunks on the target, and replies the collected metadata to the predecessor.

4. When a sync-done message arrives, the service knows that the storage target is up-to-date. It sets local state of the target to up-to-date in heartbeat messages sent to cluster manager.

When a storage service finds a previously offline successor is online:

1. The service starts to forward normal write requests to the successor. Clients may only update a portion of the chunk, but the forwarded write requests should contain the whole chunk, i.e. a full-chunk-replace write.

2. The service sends a dump-chunkmeta request to the successor. Once the metadata of all chunks on the successor target are received, it collects the chunk metadata on its local target. Then it compares the two copies of chunk metadata to decide which chunks should be transferred.

3. The selected chunks are transferred to the successor by issuing full-chunk-replace write requests.

- The chunk lock is first acquired for each chunk.

- The chain version, committed version number and chunk content are read and transferred to successor by sending a full-chunk-replace request.

- The chunk lock is released.

4\. When all required chunks have been transferred, a sync-done message is sent to the successor.

The rules used to decide which chunks should be transferred are:

- If a chunk only exists on the local target, it should be transferred.

- If a chunk only exists on the remote target, it should be removed.

- If the chain version of local chunk replica is greater than that of the remote chunk replica, it should be transferred.

- If the chain versions of local/remote chunk replicas are the same but local committed version number does not equal to the remote pending version number, it should be transferred.

- Otherwise, two chunk replicas are either the same or being updated by in-progress write requests.

Chunks and the metadata

File chunks are stored in the chunk engine. On each SSD, the persistent storage of the chunk engine consists of a fixed number of data files for storing chunk data, and a RocksDB instance for maintaining chunk metadata and other system information. Additionally, the chunk engine maintains an in-memory cache of chunk metadata to enhance query performance. A chunk allocator is implemented for fast allocation of new chunks. The chunk engine interface provides thread-safe access through the following operations:

1. *open/close* Initializes the engine by loading metadata from RocksDB and reconstructing chunk allocator states.

2. *get* Retrieves chunk metadata and reference-counted handle through a hashmap cache, enabling concurrent access with O(1) average complexity.

3. *update* Implements copy-on-write (COW) semantics by allocating new chunks before modifying data. Old chunks remain readable until all handles are released.

4. *commit* Commit the updated chunk metadata to RocksDB via write batches to ensure atomic updates; synchronously refresh the chunk metadata cache.

The chunk data will ultimately be stored on physical blocks. Physical block sizes range from 64KiB to 64MiB in increments of powers of two, totaling 11 distinct sizes. The allocator will assign physical blocks whose sizes most closely match the actual chunk size. A resource pool is constructed for each physical block size, with each pool containing 256 physical files. The usage status of physical blocks is maintained in memory using bitmaps. When a physical block is reclaimed, its bitmap flag is set to 0. The actual storage space of the block remains preserved and will be prioritized for subsequent allocations. When no available physical blocks remain, `fallocate()` will be used to allocate a contiguous large space in physical files, creating 256 new physical blocks - this approach helps reduce disk fragmentation.

When performing write operations on a chunk, the allocator first assigns a new physical block. The system then reads existing chunk data into a buffer, applies the update, and writes the updated buffer to the newly allocated block. An optimized process is implemented for appends, where data is directly added in-place at the end of the existing block. A new copy of metadata is constructed from the new block's location and existing chunk metadata. Subsequently, both the new chunk metadata and statuses of new and old physical blocks are atomically updated in RocksDB.