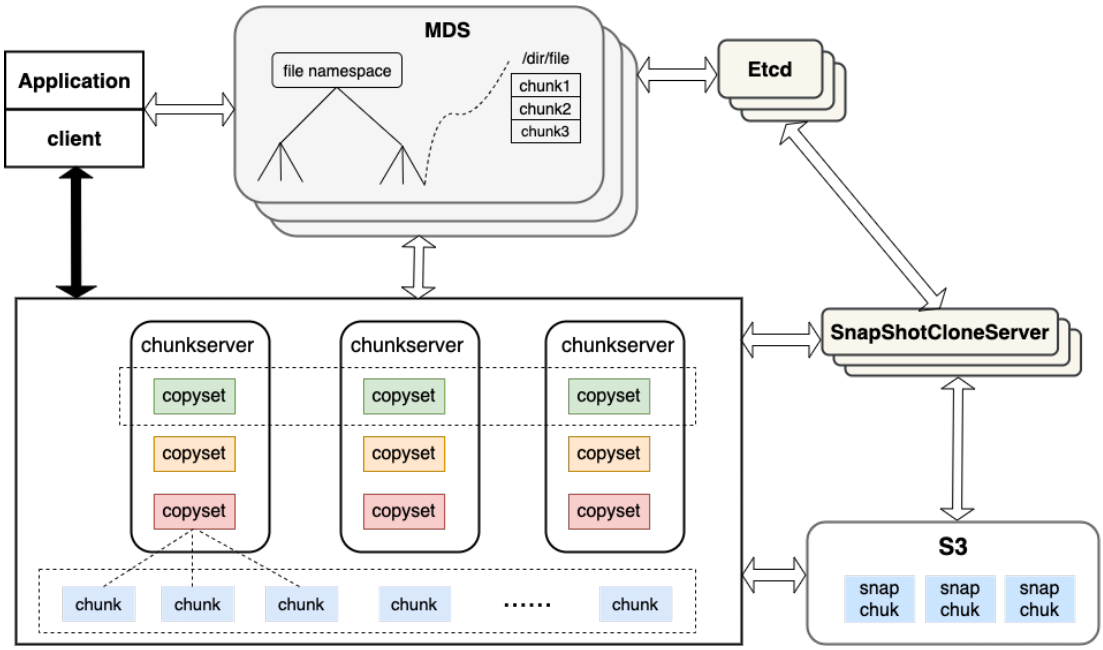
CurveBS I/O processing flow

Before introducing IO processing flow, we first describe the overall architecture, data organization and topology structure of CURVE.



CurveBS uses the central node Metadata Server (MDS) to manage virtual disk mapping and data replicas distribution. Decentralization currently leads to excessive complexity. With the continuous development of software and hardware, the need for decentralization is not particularly strong in most cases.

The CurveBS core consists of three parts:

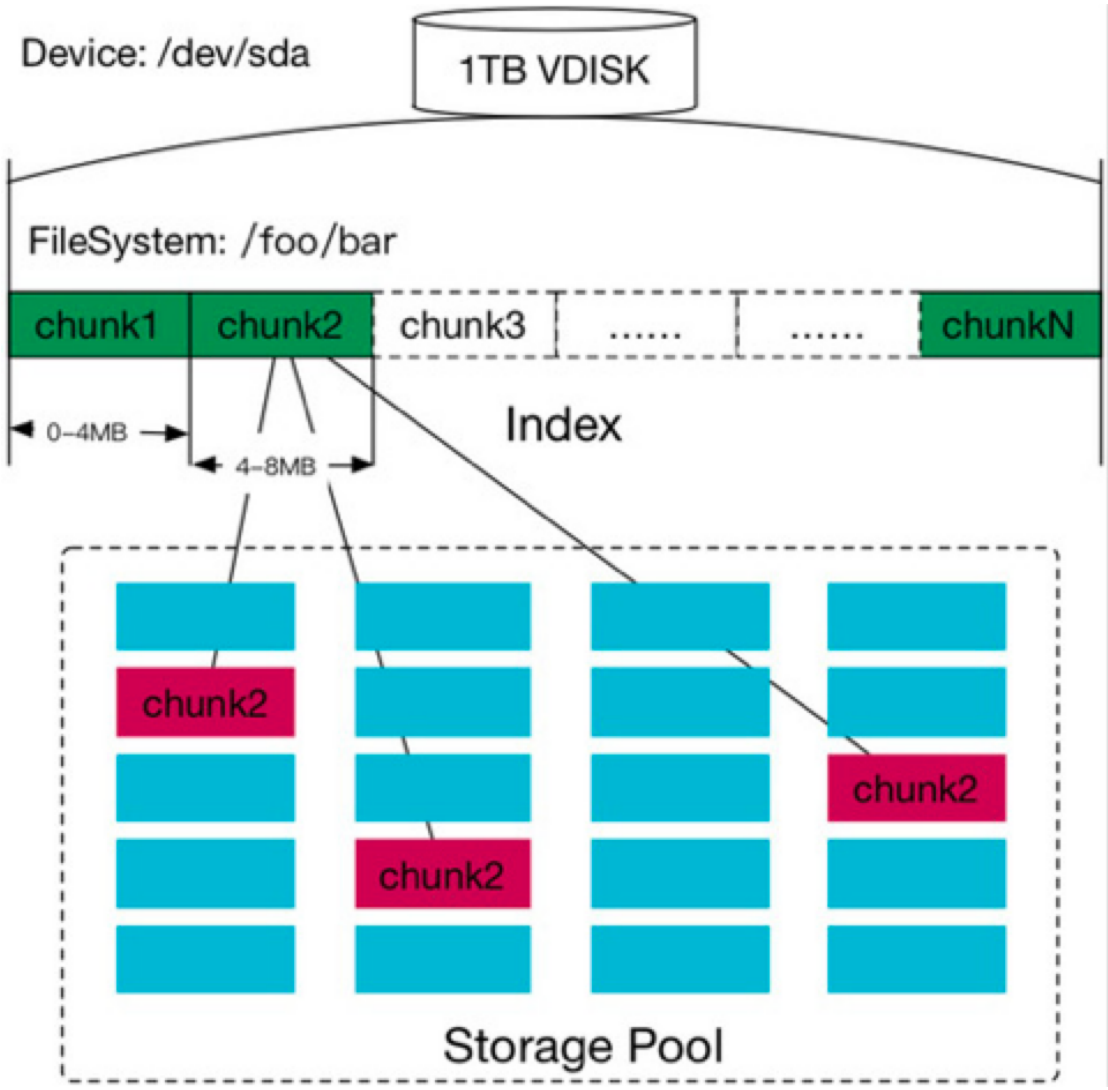
1. Metadata Server (MDS)
   - Manages and stores metadata information and persists the data in ETCD
   - Collect cluster status and schedule.
2. Chunkserver

- Responsible for data storage
- Multi-replicas consistency

3. The client
   - Provides read and write data interfaces for upper-layer applications
   - Interacts with MDS to add, delete, modify, and query metadata
   - Interacts with the chunkServer to read and write data
4. Snapshotcloneserver
   - Independent of core services
   - Snapshot data is stored in object storage which supports S3 apis. Therefore, there is no limit of snapshots
   - Support asynchronous and incremental snapshot
   - Support lazy (pre-allocated space) and non-lazy (allocated space on needs) clones from snapshots/mirrors
   - Support rollback from snapshot

CurveBS data organization

Data organization of virtual block device in CurveBS

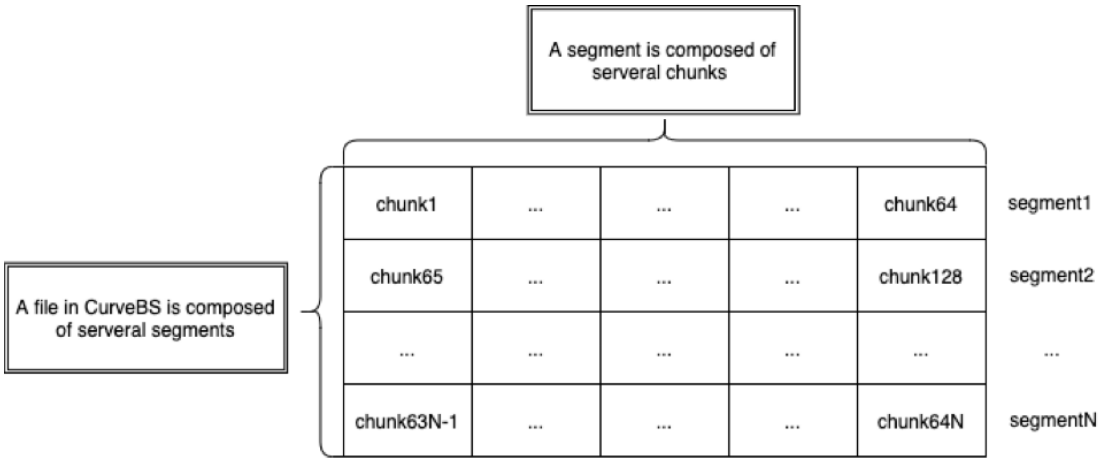


1. CurveBS maps the virtual block device to a file. For example, block device /dev/sda corresponds to file /foo/bar in CurveBS
2. The address space of the block device /dev/sda maps to chunks of file in the system. For example, chunk1 corresponds to the address space of 0 to 4MB in /dev/sda, and chunk2 corresponds to the address space of 4 to 8MB in /dev/sda. The size of chunk can be configured
3. Each file (/foo/bar) contains chunks scattered all over the storage nodes. ChunkServer provides 4KB random read/write capability to support 4KB aligned read/write on block devices.

CurveBS file structure of virtual block device

As mentioned above, CurveBS maps virtual block devices to files.

Let's look at the metadata for a file in CurveBS.

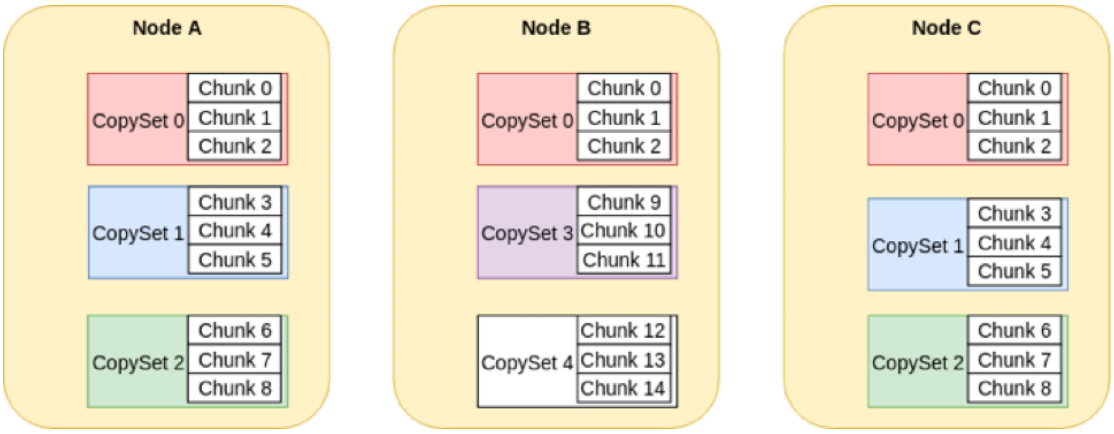


1. A file in CurveBS consists of chunks. The default size of chunk is 16MB. If the file directly maps to chunk, a 4TB file will consist of 256KB chunks when the chunk size is set to 16MB. There will be lots of pressure on KV storage, so we use two-level mapping and introduced segment.

2. Segment is also the granularity of space allocation. Multiple chunks are aggregated into one segment. The default segment size can be 1GB. it only requires 4KB segments for a 4TB file, according to 4KB records in KV.

The metadata is stored on the MDS.

Data storage in CurveBS



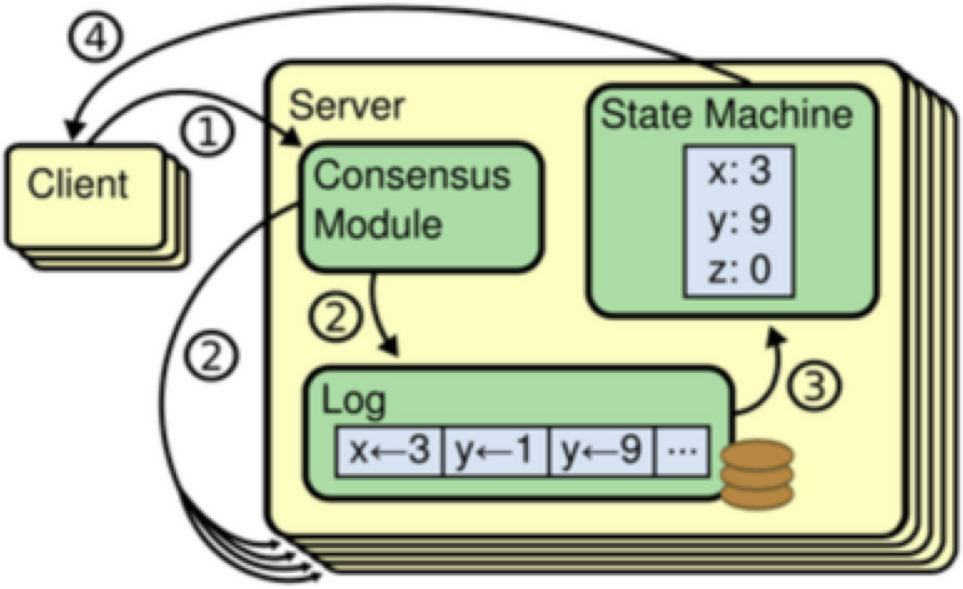
This section mainly describes how data nodes manage chunk. Each node in the figure represents a data node (ChunkServer).

A CopySet is the basic unit of data replication in CurveBS. A CopySet contains multiple chunks. Copysets in CurveBS are created in advance. When users write data, they only need to select an appropriate CopySet and create a new chunk.

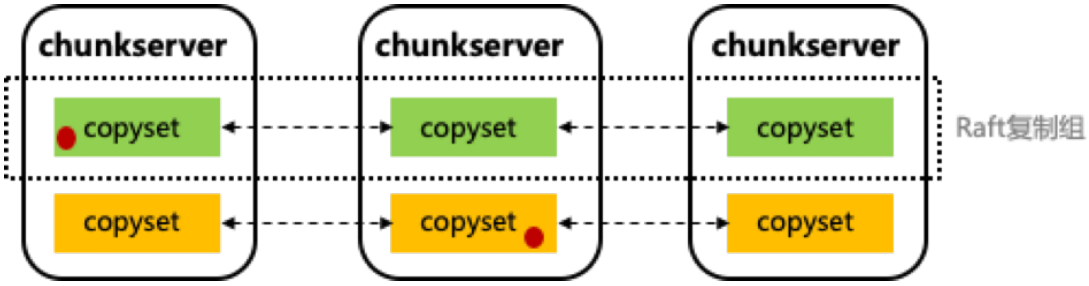
Why use a CopySet to manage data?

1. Reduce metadata. If the replication group membership is saved for each chunk, the number of metadata will be large.
2. Example Reduce the number of replication groups. The size of chunk is usually small, such as 16MB by default. If replication groups are created based on chunk. The number of replication groups is large and the communication traffic between replication groups is heavy. Once a CopySet is introduced, it can be explored with CopySet granularity.
3. Improve data reliability. If the replication group is excessively scattered, the probability of replication group failure increases.

Data consistency in CurveBS



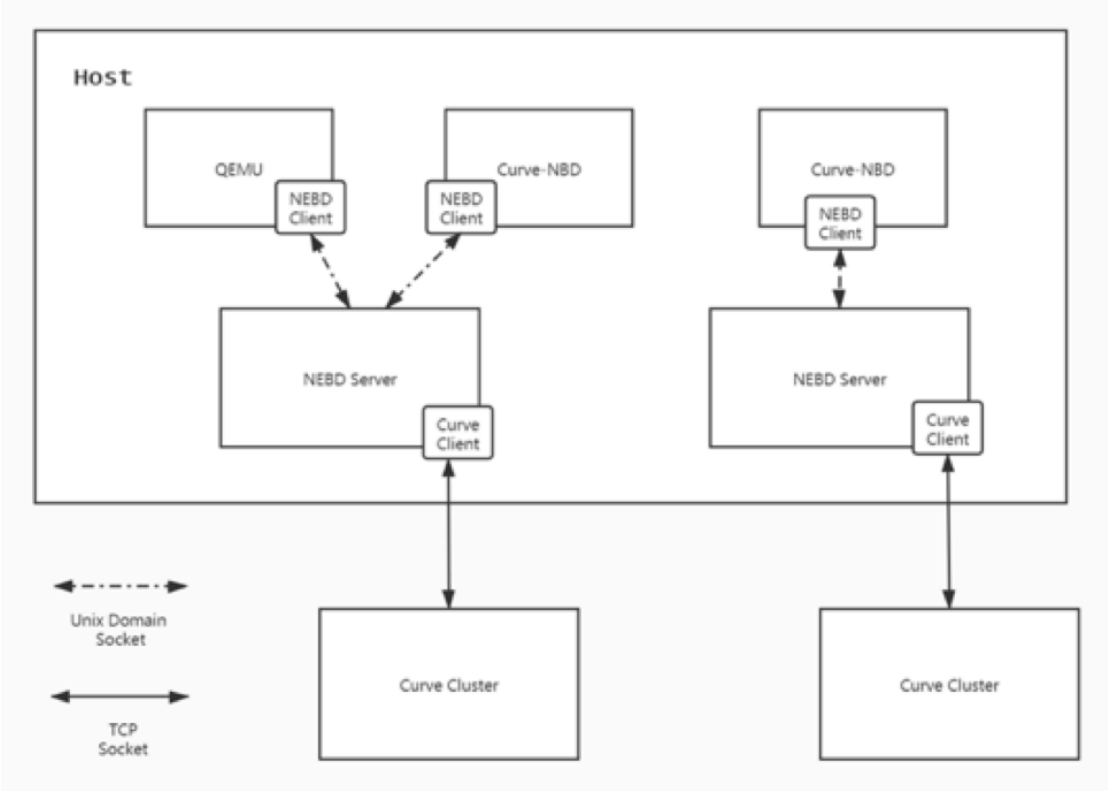
CurveBS uses RAFT as a consistency protocol. The figure above is from the RAFT paper(https://raft.github.io/raft.pdf).



CurveBS uses BRaft to implement the consistency protocol. For CurveBS

1. A CopySet, as the basic unit of a consistent replication group, contains Consensus Module and Log Module.
2. The State Machine corresponds to the chunk in the CurveBS, and it applies the operations on the chunk (write, delete, snapshot) to the corresponding chunk.

Client

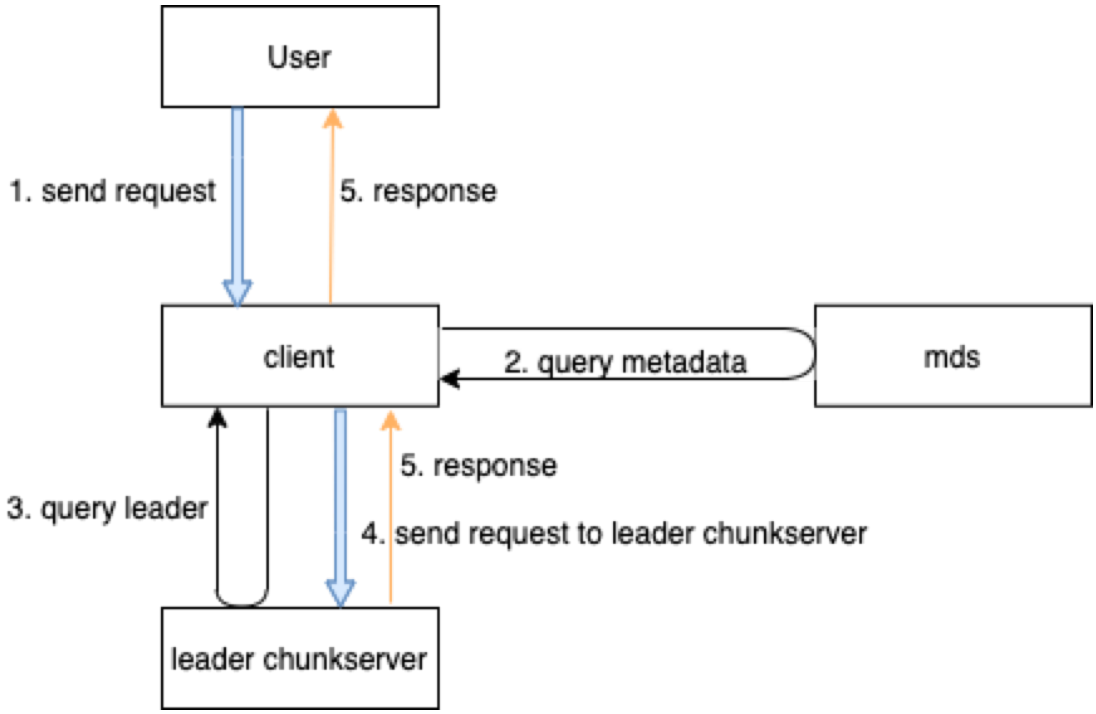


1. CurveBS client provides services by libCurve. At first, external processes such as QEMU, Cinder, and NBD use it to access storage services provided by Curve. The client provids posix-like interface, such as read/write/aioread/aiowrite in data plane and open/create/rename/extend, etc in control plane.
2. Depending on libCurve, the application must be restarted every time the library is updated. To solve this problem, the CurveBS Client has been optimized to decouple the application from libcurve and add a NEBD module to join the two.
   - Nebdclient: Forward the requests from QEMU and curve-NBD to the specified NEBDServer through Unix Domain sockets.
   - Nebdserver: Accepts requests from NEBDClient and calls Curve Client for corresponding processing. it can receive requests from different NEBDClients.

3. Through the above splitting, NebdClient replaces Curve Client and directly interfaces with upper services. There is no logical processing in NEBDClient, it just proxy requests and has limited retries, which ensuring that NEBDClient needs no change. To upgrade curve-client, you only need to restart the NEBDServer, which affects services in few seconds.

CurveBS IO processing flow

Overall process



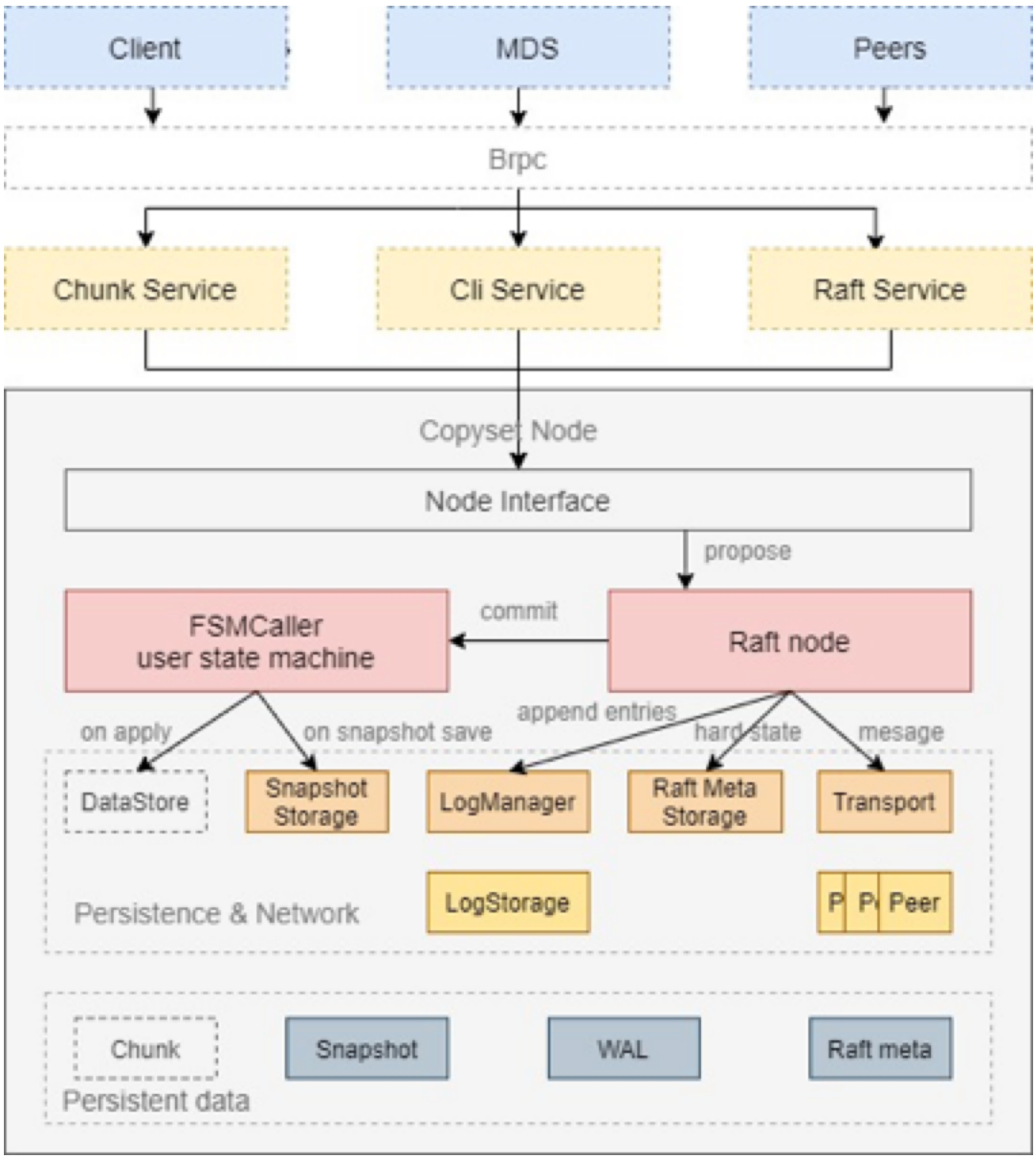
The flow of user's read and write request to curve:

1. The user initiates a request (fd=1, offset=0, length=4096).
2. The Client queries metadata from the MDS and stores the metadata locally. The request is converted to a chunk request (CopysetId=1, chunkId=5678, offset in chunk = 0, length in Chunk = 4096).
   If the currently segment written is not allocated, the client request the MDS to allocate segment. As mentioned earlier, segments are allocated with a default size of 1GB (configurable >=

1GB), so client requests MDS for new segment infrequently. For random write scenarios, it is possible to allocate space frequently at the beginning, but after the allocation is complete, performance recovers.

3. The Client queries the ChunkServer for the leader ChunkServer node of the copyset where chunk resides.
4. The Client sends read/write requests to the leader (IP=127.0.0.1, port=8200, CopysetId=1, chunkId=5678, offset in chunk=0, length in Chunk =4096).
5. The Chunkserver completes the request and notifies the client that the request is completed.
6. At last, the Client notifies the user that the request is completed.

Chunkserver processing flow



CopysetNode is essentially a wrapper around the RAFT algorithm. The Copyset module encapsulates the RAFT Node of BRaft and implements the user state machine of BRaft. Take a write request as an example

1. The Client sends the request to the ChunkServer where the replication group Leader resides
2. When the user request arrives at the ChunkServer, the CopysetNode will encapsulate the request into a task (the task is mainly the user's data) and submit it to the BRaft node.

3. The BRaft node persists the log entry locally and replicates it to the followers so that the followers can also persist the log entry.

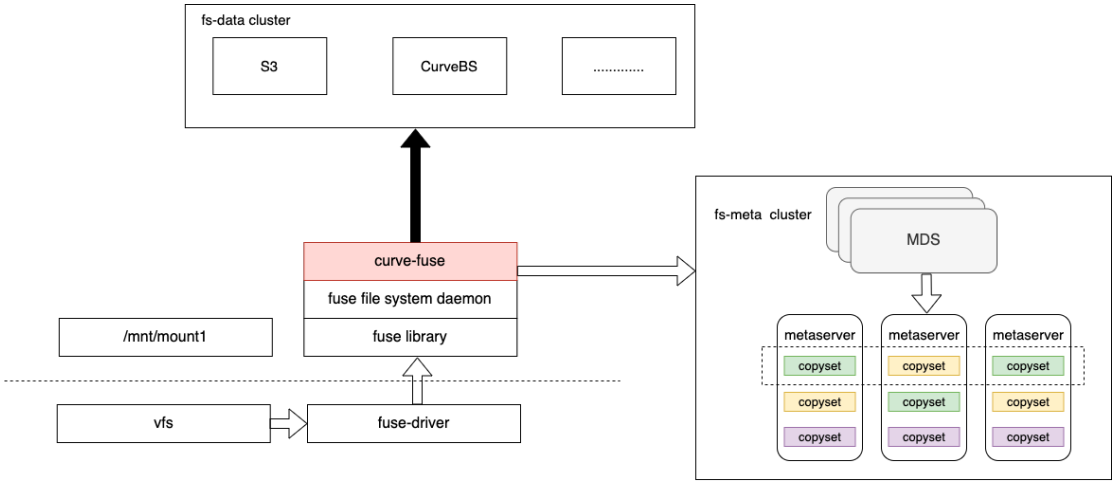
BRaft Log Format: EntryHeader + Entry.

```
// Format of Header, all fields are in network order
// | -------------------- term (64bits) ------------------------ |
// | entry-type (8bits) | checksum_type (8bits) | reserved(16bits) |
// | ------------------ data len (32bits) ---------------------- |
// | data_checksum (32bits) | header checksum (32bits) ----------- |

const static size_t ENTRY_HEADER_SIZE = 24;

struct Segment::EntryHeader {
    int64_t term;
    int type;
    int checksum_type;
    uint32_t data_len;
    uint32_t data_checksum;
};

struct LogEntry : public butil::RefCountedThreadSafe<LogEntry> {
public:
    EntryType type; // log type
    LogId id;
    std::vector<PeerId>* peers; // peers
    std::vector<PeerId>* old_peers; // peers
    butil::IOBuf data;

    LogEntry();

private:
    DISALLOW_COPY_AND_ASSIGN(LogEntry);
    friend class butil::RefCountedThreadSafe<LogEntry>;
    virtual ~LogEntry();
};
```

4. When majority nodes, including themselves, successfully persist the log entry, we call it commit

5. After commit, apply can be applied. During apply, BRaft will call the function that we pass in through the user state machine to complete the disk write operation.

CurveFS architecture



Curvefs implements the file system interface based on FUSE.

1. Fs-meta Cluster is used to manage the inode and dentry metadata of files. The architecture is like CurveBS, so metadata scalability is very good in this way.
2. Fs-data cluster is used to store file data. Curve-fuse Supports Object storage by S3 apis and CurveBS

CurveBS performance considerations

1. CurveBS uses raft as a consistency protocol which requires majority replicas to be successful. Compare with strong consistency protocol which needs all replicas successful, it provides lower write latencies.

   Ceph community have compared majority replicas case with all replicas case on write latency difference:

   https://github.com/ceph/ceph/pull/15027

In the test, the average latency was reduced by 11.5% and the IOPS increased by 13% in the case of two replicas. The gap is even greater under high stress.

yanjiangxu commented on 10 May 2017 • edited

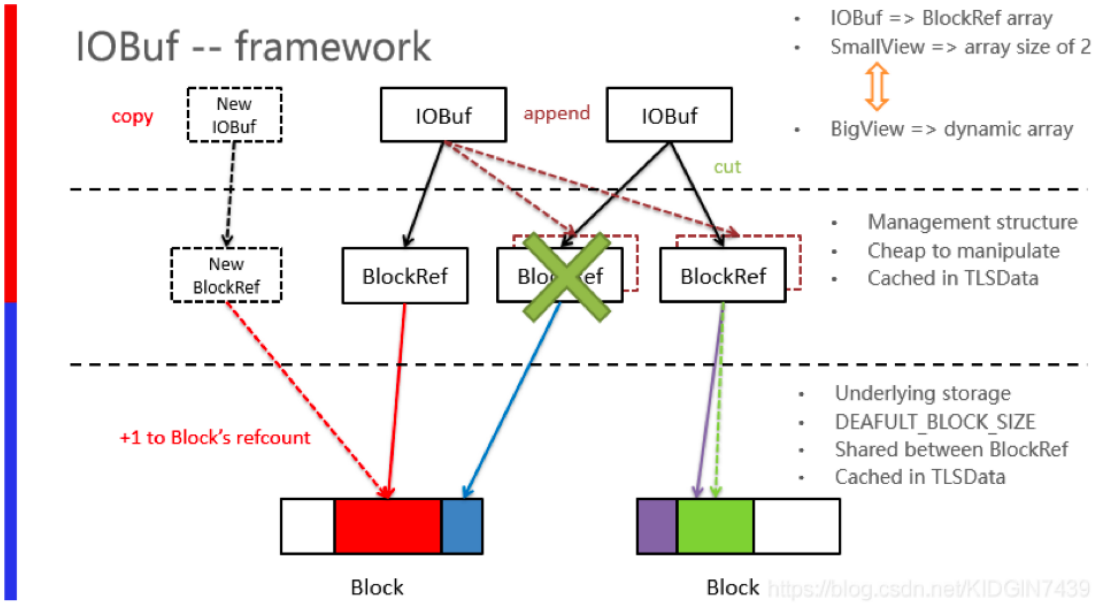
The behavior In original flow:
1) In the case of 3 copies, the client sends the write to the primary, and then the primary forwards to the other two replicas. The primary must wait for the commit until all the replicas from completion. The real write IO latency is the longest latency among three copies.
2) When the data distribution is not uniform, the osd node which owns hottest data will become the bottleneck of total latency.

The behavior In my enhancement:
commit_majority function is not required to wait for all copies of commit to complete: as long as the majority of the commit is completed, the cluster would notify the client to complete.

As in my implementation:
We also follow the scenario: When the number of copies is less than min_size, ceph does not provide read and write capabilities. In order to ensure security, you must ensure that the number of commit to majority.
And I defined two important variables:
`allow_uncommitted_replicas = pool_min_size - 1,`
`majority = pool_default_size - allow_uncommitted_replicas.`

Test case:
1) 3 copies, pool_default_size = 3, pool_min_size = 2, majority = 2,allow_uncommitted_replicas=1
2) The average latency decreased by **11.5%**;
3) Long tail (99.9%) latency decreased by **71.59%**;
4) IOPS increased by **13%**.

2. User space Zero copy of user data



Iobuf structure is used to store user data, and user space data is transferred through reference of data address, avoiding user space data copy.