RELIABLE, SCALABLE, AND HIGH-PERFORMANCE DISTRIBUTED STORAGE: Distributed Object Storage

Authored by: Sage Weil

Abstract

Distributed object storage architecture leverages device intelligence to provide a reliable and scalable storage abstraction with minimal oversight. Described is an efficient, scalable, and low-overhead cluster management protocol that facilitates consistent and coherent data access through the propagation of small cluster maps that specify device membership and data distribution. This allows a dynamic cluster of semi-autonomous OSDs to self-manage consistent data replication, failure detection, and failure recovery while providing the illusion of a single logical object store with direct, high-performance client access to data.

Keywords: RADOS, cluster map, data distribution, object storage devices

Introduction

The work described is a scalable and reliable object storage service termed RADOS (Reliable, Autonomic Distributed Object Store) without compromising performance. RADOS facilitates an evolving, balanced distribution of data and workload across a dynamic and heterogeneous storage cluster while providing applications with the illusion of a single logical object store with well-defined safety semantics and strong consistency guarantees. Metadata bottlenecks associated with data layout and storage allocation are eliminated through the use of a compact cluster map that describes cluster state and data layout in terms of placement groups.

By separating serialization from safety, the architecture provides strong consistency semantics to applications by minimally involving clients in failure recovery. RADOS utilizes a globally replicated cluster map that provides all parties with complete knowledge of the data distribution, typically specified using a function like CRUSH. This avoids the need for object lookup present in conventional architectures, which RADOS leverages to distribute replication, consistency management, and failure recovery among a dynamic cluster of OSDs while still preserving consistent read and update semantics. A scalable failure detection and cluster map distribution strategy enables the creation of extremely large storage clusters, with minimal oversight by the tightly-coupled and highly reliable monitor cluster that manages the master copy of the map. Because clusters at the petabyte scale are necessarily heterogeneous and dynamic, OSDs employ a robust recovery algorithm that copes with any combination of device failures, recoveries, or data reorganizations. Recovery from transient outages is fast and efficient, and parallel re-replication of data in response to node failures limits the risk of data loss.

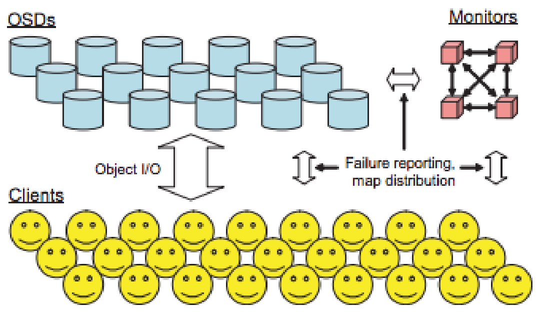

Figure 6.1: A cluster of many thousands of OSDs store all objects in the system. A small, tightly coupled cluster of monitors collectively manage the cluster map that describes the current cluster composition and the distribution of data. Each client instance exposes a simple storage interface to applications.

Description

The main components of the work are:

1. Distributed Object Storage- RADOS achieves scalability by eliminating the controllers and gateway servers present in most storage architectures. Clients are given direct access to storage devices. This is enabled by CRUSH, which provides clients and OSDs (object storage devices) with complete knowledge of the current data distribution. When device failures or cluster expansion require a change in the distribution of data, OSDs communicate amongst themselves to realize that distribution, without any need for controller oversight. The RADOS cluster is managed exclusively through manipulation of the cluster map, a small data structure that describes what OSDs are participating in the storage cluster and how data is mapped onto those devices. A small cluster of highly-reliable monitors are jointly responsible for maintaining the map and seeing that OSDs learn about cluster changes. Because the cluster map is small, well-known, and completely specifies the data distribution, clients are able to treat the entire storage cluster as a single logical object store.

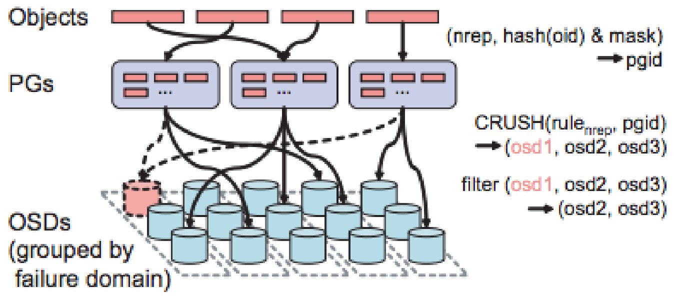
Objects
PGs
OSDs (grouped by failure domain)
(nrep, hash(oid) & mask) → pgid
CRUSH(rule$_{nrep}$, pgid) → (osd1, osd2, osd3)
filter (osd1, osd2, osd3) → (osd2, osd3)

Figure 6.2: Objects are grouped into placement groups (PGs), and distributed to OSDs via CRUSH, a specialized replica placement function. Failed OSDs (e.g. osd1) are filtered out of final mapping

   a. Data Placement- RADOS employs a data distribution policy in which objects are pseudo-randomly assigned to devices. When new storage is added, a random subsample of existing data is migrated to new devices to restore balance. This strategy maintains a probabilistically balanced distribution that, on average, keeps all devices similarly loaded, allowing the system to perform well under any potential workload. Most importantly, data placement takes the form of a pseudo-random function that calculates the proper location of objects; no large or cumbersome centralized allocation table is needed.

      i. Each object stored by the system is first mapped into a placement group (PG). Each object's PG is determined by a hash of the object name o, the desired level of replication r, and a bit mask m that controls the total number of placement groups in the system. That is, $pgid = (r, \text{hash}(o)\&m)$, where & is a bit-wise AND and the mask $m = 2^k - 1$, constraining the number of PGs by a power of two.

      ii. As the cluster scales, it is periodically necessary to adjust the total number of placement groups. During such adjustments, PG contents can be split in two by adding a bit to m. However, to avoid a massive split from simultaneously affecting all placement groups in the system—resulting in a massive reshuffling of half of all data—in practice we replace the m mask with the stablemod(x,n,m) function, where $n\&m = n$ and $n\&m = 0$ (where the bar indicates a bit-wise NOT). That is, $pgid = (r, \text{stablemod}(\text{hash}(o),n,m))$. This similarly constrains the range of $pgid$ while allowing n to be *any* number of PGs—not just a power of two. If $x\&m$ is less than n, we proceed as before. Otherwise, stablemod(x,n,m) returns $x\&(m >> 1)$ (see Algorithm 3). This provides a “smooth” transition between bit masks m, such that PG splits can be spread over time.

Algorithm 3 Function to constrain the number of PGs. Note that $m = 2^k - 1$, $n\&m = n$, and $n\&\bar{m} = 0$.

```
1: procedure STABLEMOD($x,n,m$)          ▷ Choose between mask $m$ and $m \gg 1$
2:     if $x\&m < n$ then
3:         return $x\&m$                   ▷ Use larger mask.
4:     else
5:         return $x\&(m \gg 1)$           ▷ Use smaller mask.
6:     end if
7: end procedure
```

iii. Placement groups are assigned to OSDs using CRUSH (see Data Distribution), a pseudorandom data distribution function that efficiently maps each PG to an ordered list of r OSDs upon which to store object replicas. CRUSH behaves similarly to a hash function: placement groups are deterministically but pseudo-randomly distributed. Unlike a hash function, however, CRUSH is stable: when one (or many) devices join or leave the cluster, most PGs remain where they are; CRUSH shifts just enough data to maintain a balanced distribution. CRUSH also uses weights to control the relative amount of data assigned to each device based on its capacity or performance.

iv. Placement groups provide a means of controlling the level of replication declustering. That is the number of replication peers is related to the number of PGs μ it stores—typically on the order of 100 in the current system. As a cluster grows, the PG mask m can be periodically adjusted to “split” each PG in two. Because distribution is stochastic, μ also affects the variance in device utilizations: more PGs result in a more balanced distribution. More importantly, declustering facilitates distributed, parallel failure recovery by allowing each PG to be independently re-replicated from and to different OSDs. At the same time, the system can limit its exposure to coincident device failures by restricting the number of OSDs with which each device shares common data.

b. Cluster Maps- The cluster map provides a globally known specification of which OSDs are responsible for which data, and which devices are allowed to process reads or updates. Each time the cluster map changes due to an OSD status change, the map *epoch* is incremented. Map epochs allow all parties to agree on what the current distribution of data is, and to determine when their information is out of data. Updates are distributed as *incremental maps*: small messages describing the differences between two successive epochs. In most cases, such updates simply state that one or more OSDs have failed or recovered, although in general they may include status changes for many devices, and multiple updates may be bundled together to describe the difference between distant map epochs.

epoch:	map revision
m:	number of placement groups -1
up:	OSD $\mapsto$ { network address, *down* }
in:	OSD $\mapsto$ { *in*, *out* }
crush:	CRUSH hierarchy and placement rules

Table 6.1: Data elements present in the OSD cluster map, which describes both cluster state and the distribution of data.

i. Down and Out- The cluster map’s hierarchical specification of storage devices is complemented by the current network address of all OSDs that are currently online and reachable (*up*), and indication of which devices are currently *down*. RADOS considers an additional dimension of OSD liveness: *in* devices are included in the mapping and assigned placement groups, while *out* devices are not. For each PG, CRUSH produces a list of exactly *r* OSDs that are *in* the mapping. RADOS then filters out devices that are *down* to produce the list of active OSDs for the PG. If the active list is currently empty, PG data is temporarily unavailable, and pending I/O is blocked.

c. Communication and Failure Model- RADOS employs an asynchronous, ordered point to point message passing library for communication. For simplicity, the prototype considers a failure on the TCP socket to imply a device failure, and immediately reports it. OSDs exchange periodic heartbeat messages with their peers to ensure that failures are detected. This is somewhat conservative in that an extended ethernet disconnect or a disruption in routing at the IP level will cause an immediate connection drop and failure report. However, it is safe in that any failure of the process, host, or host's network interface will eventually cause a dropped connection. This strategy can be made somewhat more robust by introducing one or more reconnection attempts to better tolerate network intermittency before reporting a failure. OSDs that discover that they have been marked *down* simply sync to disk and kill themselves to ensure consistent behavior

d. Monitors- All OSD failures are reported to a small cluster of *monitors*, which are jointly responsible for maintaining the master copy of the cluster map. OSDs can request the latest cluster map from or report failures to any monitor. When an OSD submits a failure report, it expects to receive an acknowledgement in the form of a map update that marks the failed OSD *down* (or back *up* at a new address). If it does not get a response after a few seconds, it simply tries contacting a different monitor.
    i. All OSD failures are reported to a small cluster of monitors, which are jointly responsible for maintaining the master copy of the cluster map. OSDs can request the latest cluster map from or report failures to any monitor. When an OSD submits a failure report, it expects to receive an acknowledgement in the form of a map update that marks the failed OSD *down*. If it does not get a response after a few seconds, it simply tries contacting a different monitor.
    ii. The monitor cluster allows only a single update to be proposed at a time, simplifying the implementation, while also coordinating updates with a *lease* mechanism to provide a consistent ordering of cluster map read and update operations.
    iii. The cluster initially elects a *leader* to serialize map updates and manage consistency. Once elected, the leader begins by requesting the map epochs stored by each monitor. Monitors have a fixed amount of time *T* to respond to the probe and join the *quorum*. The leader ensures that a majority of the monitors are active and that it has the most recent map epoch, and then begins distributing short-term leases to active monitors.
    iv. Each lease grants active monitors permission to distribute copies of the cluster map to OSDs or clients who request it. If the lease term *T* expires without being renewed, it is assumed the leader has died and a new election is called. Each lease is acknowledged to the leader upon receipt. If the leader does not receive timely acknowledgements when a new lease is distributed, it assumes an active monitor has died and a new election is called. When a monitor first starts up, or finds that a previously called election does not complete after a reasonable interval, an election is called.
    v. When an active monitor receives an update request, it first checks to see if it is a new. If, for example, the OSD in question was already marked *down*, the monitor simply responds with the necessary incremental map updates to bring the reporting OSD up to date. New failures are forwarded to the leader, who serializes updates, increments the map epoch, and an update protocol to distribute the update to other monitors, simultaneously revoking leases. Once the update is acknowledged by a majority of monitors a final commit message issues a new lease.
    vi. The combination of a synchronous two-phase commit and the probe interval *T* ensures that if the

active set of monitors changes, it is guaranteed that all prior leases (which have a matching term *T*) will have expired before any subsequent map updates take place. Consequently, any sequence of map queries and updates will result in a consistent progression of map versions, provided a majority of monitors is available.

e. Map Propagation- Differences in map epochs are significant only when they vary between two communicating OSDs (or between a client and OSD), which must agree on their proper roles with respect to a particular PG. This property allows RADOS to distribute map updates by combining them with existing inter-OSD messages, shifting the distribution burden to OSDs. Each OSD maintains a history of past map incrementals, tags all messages with its latest epoch, and makes note of its peers' epochs. If an OSD receives a message from a peer with an older map, it shares the necessary incremental(s) to bring that peer in sync. Similarly, when contacting a peer thought to have an older epoch, incremental updates are preemptively shared. The heartbeat messages periodically exchanged for failure detection ensure that updates spread quickly—in $O(logn)$ time for a cluster of n OSDs.

2. Reliable Autonomic Storage- RADOS replicates each data object on two or more devices for reliability and availability. Replication and failure recovery are managed entirely by OSDs through a version-based consistency scheme utilizing short-term update logs. A peer to peer recovery protocol avoids any need for controller-driven recovery, facilitating a flat cluster architecture with excellent scalability.
   a. Replication- Storage devices are responsible for update serialization and write replication, shifting the network overhead associated with replication from the client network or WAN to the OSD cluster's internal network, where greater bandwidth and lower latencies are expected. RADOS implements primary-copy replication, chain replication, and *splay* replication that combines elements of the two. All three strategies provide strong consistency guarantees, such that read and write operations occur in some sequential order, and completed writes are reflected by subsequent reads.

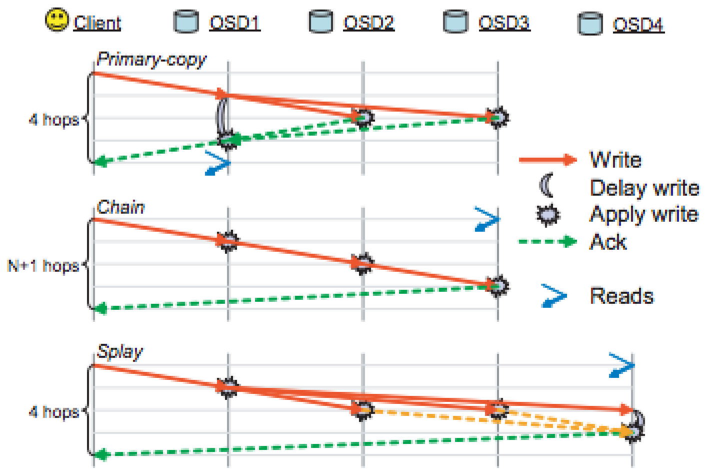

Figure 6.3: Replication strategies implemented by RADOS. Primary- copy processes both reads and writes on the first OSD and updates replicas in parallel, while chain forwards writes sequentially and processes reads at the tail. Splay replication combines parallel updates with reads at the tail to minimize update latency.

      i. Primary-copy replication, the first OSD in a PG's list of active devices is the primary, while additional OSDs are called replicas. Clients submit both reads and writes to the primary, which serializes updates within each PG. The write is forwarded to the replicas, which apply the update to their local object store and reply to the primary. Once all replicas are updated, the primary applies the update and replies to the client, as shown in Figure 6.3.

ii. Chain replication separates update serialization from read processing. Writes are directed at the first OSD (the *head*), which applies the update locally and forwards it to the next OSD in the list. The last OSD (the *tail*) responds to the client. Reads are directed at the tail, whose responses will always reflect fully replicated updates. For 2x replication, this offers a clear advantage: only three messages and network hops are necessary, versus four for primary-copy replication. However, latency is dependent on the length of the chain, making the strategy problematic for high levels of replication.

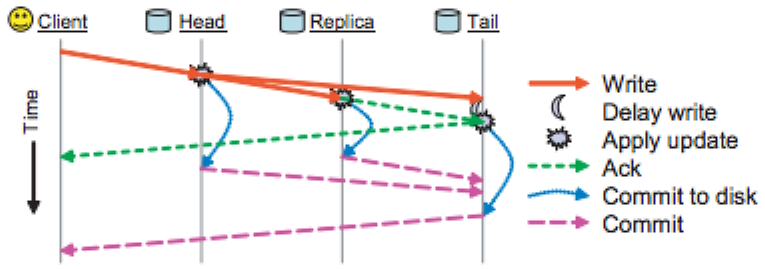

Figure 6.4: RADOS responds with an *ack* after the write has been applied to the buffer caches on all OSDs replicating the object (shown here with splay replication). Only after is has been safely committed to disk is a second *commit* notification sent to the client.

iii. Splay replication combines elements of the two. As with chain replication, updates are directed at the head and reads at the tail. For high levels of replication, however, updates to the middle OSDs occur in parallel, lowering the latency seen by the client. Both primary-copy andsplay replication delay the local write in order to maintain strong consistency in the presence of an OSD failure, although splay must do so for less time, lowering OSD memory requirements.

b. Serialization vs Safety- RADOS disassociates write acknowledgement from safety at all levels in order to provide both efficient update serialization and strong data safety. During a replicated write, each replica OSD sends an *ack* to the tail immediately after applying the update to the in-memory cache of the local EBOFS object store, and the tail responds to the client with an *ack* only after all replicas have applied the update. Later, the EBOFS (Extent and B- tree based Object File System) provides each OSD with asynchronous notification that the update is safely committed to disk, they send a second message to the tail, and only after all replicas have done so is the client sent a final *commit*. The strategy is similar for the other schemes: with primary-copy replication, *ack*s and *commits* go to the primary instead of the tail, and with chain replication, only *commits* go to the tail (the replicated update itself is an implicit *ack*). Once clients receive an *ack*, they can be sure their writes are visible to others, and synchronous application level calls can typically unblock. Clients buffer all updates until a *commit* is received, however, allowing clients to participate in recovery if all OSDs replicating the update fail, losing their in-memory (uncommitted) state.

c. Maps and Consistency- All RADOS messages are tagged with the map epoch to ensure that all update operations are applied in a fully consistent fashion. All replicas are involved in any given update operation and any relevant map updates will be discovered. Because a given set of OSDs who are newly responsible for a PG cannot become active without consulting prior members or determining they are failed, no updates can be lost, and consistency is maintained.

i. In the event of a partial network failure that results in an OSD becoming only partially unreachable, the OSD servicing reads for a PG could be declared "failed" but still be reachable by clients with an old map. Meanwhile, the updated map may specify a new OSD in its place. In order to prevent any read operations from being processed by the old OSD after new updates are be processed by the new one, timely heartbeat messages are required between OSDs in each PG in order for the PG to remain available (readable). If the OSD servicing reads have not heard from other replicas in *H* seconds, reads will block. Then, in order for a new OSD to take over that role from another OSD, it must either obtain positive acknowledgement from the old OSD, ensuring they are aware of their role change, or delay for the same time interval.

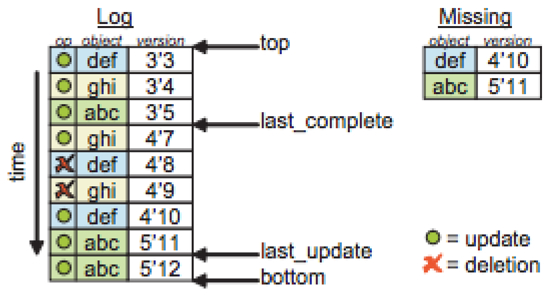

Figure 6.5: Each PG has a log summarizing recent object updates and deletions. The most recently applied operation is indicated by *last_update*. All updates above *last_complete* are known to have been applied, while any missing objects in the interval between *last_complete* and *last_update* are summarized in the missing list.

d. Versions and Logs- RADOS uses versioning to identify individual updates and to serialize them within each placement group. Each version consists of an *(epoch,v)* pair, where *epoch* reflects the map epoch at the time of the update, and *v* increases monotonically. Each PG has a *last update* attribute that reflects the most recently applied modification to one its objects, and each object has a similar version attribute to reflect the last time it was modified.
   i. OSDs maintain a short-term log of recent updates (illustrated in Figure 6.5) for each PG, stored both on disk and in RAM or NVRAM. Each log entry includes the object name, the type of operation (update, delete), a version number identifying the update, and a unique request identifier consisting of the client name and a client-assigned identifier. Unique identifiers allow OSDs to detect and ignore duplicate requests, rendering all operations idempotent.
   ii. The first OSD in the PG serializes writes by assigning a new version number and appending a new entry to its log. The request is then forwarded along with the version stamp to all other replica OSDs (or just to the next replica for chain replication). An OSD processing an update always writes to the log immediately, even if it delays the write for consistency. For this reason, the log may extend below the last update pointer (*i. e.* write-ahead).
   iii. Log appends or pointer changes are written to disk wrapped in atomic EBOFS transactions with the updates they describe, such that the log provides a perfect record of which updates were committed before any crash. The log also forms the basis for recovery when a PG is being brought up to date or replicated to an entirely new OSD.

e. Failure Recovery- RADOS failure recovery is driven entirely by cluster map updates and subsequent changes in each PG's list of active devices. Such changes may be due to device failures, recoveries, cluster expansion or contraction, or even complete data reshuffling from a totally new CRUSH replica distribution policy. When an OSD crashes and recovers, EBOFS object store will be warped back in time to the most recent snapshot committed to disk. In all cases, RADOS employs at *peering* algorithm to establish a consistent view of PG contents and to restore the proper distribution and replication of data. This strategy relies on the basic design premise that OSDs aggressively replicate the PG log and its record of what the current state of a PG even when some object replicas may be missing locally. If recovery is slow and object safety is degraded for some time, PG metadata is carefully guarded, simplifying the recovery algorithm and allowing the system to reliably detect data loss.

i. Peering- When an OSD receives a cluster map update, it walks through all new map incrementals up through the most recent to examine and possibly adjust PG state values. Any locally stored PGs whose active list of OSDs changes are marked *inactive*, indicating that they must re-peer. Considering all map epochs ensures that intermediate data distributions are taken into consideration: if an OSD is removed from a PG and then added again, it is important to realize that intervening updates to PG contents may have occurred. Peering and any subsequent recovery proceeds independently for every PG in the system.

    1. The process is driven by the first OSD in the PG (the *primary*). For each PG an OSD stores for which it is not the current primary, a *Notify* message is sent to the current primary. This message includes basic state information about the locally stored PG, including *last update*, *last complete*, the bounds of the PG log, and *last epoch started*, which indicates the most recent known epoch during which the PG successfully peered.
    2. Notify messages ensure that an OSD that is the new primary for a PG discovers its new role without having to consider all possible PGs for every map change. Once aware, the primary generates a prior set, which includes all OSDs that may have participated in the PG since last epoch started. Because this is a lower bound, as additional notifies are received, its value may be adjusted forward in time. The prior set is explicitly queried to solicit a notify to avoid waiting indefinitely for a prior OSD that does not actually store the PG.
    3. Armed with PG metadata for the entire prior set, the primary can determine the most recent update applied on any replica, and request whatever log fragments are necessary from prior OSDs in order to bring the PG logs up to date on active replicas. That is, the primary must assemble a log that stretches from the oldest log bottom on active replicas to the newest log bottom (most recent update) on any prior OSD. Because the log only reflects recent history, this may not be possible (e. g. if the primary is new to the PG and does not have any PG contents at all), making it necessary for the primary to generate or request a backlog. A backlog is an extended form of the log that includes entries above the top pointer to reflect any other objects that exist in the PG (i. e. on disk) but have not been modified recently. The backlog is generated by simply scanning locally stored PG contents and creating entries for objects with versions prior to the log top. Because it does not reflect prior deletions, the backlog is only a partial record of the PG's modification history.
    4. Once the primary has assembled a sufficient log, it has a complete picture of the most recent PG contents: they are either summarized entirely by the log, or the recent log in combination with locally stored objects. From this, the primary updates its missing list by scanning the log for objects it does not have. All OSDs maintain a missing list for active PGs, and include it when logs are requested by the primary. The primary can infer where objects can be found by looking at which OSDs include the object in their log but don't list it as missing.
    5. Once the log and missing list are complete, the PG is ready to be activated. The primary first sends a message to all OSDs in the prior set (but not in the active set) to update last epoch started. Once this is acknowledged, the primary sets its own PG to active, and sends a log fragment to each OSD in the active set to bring them up to date and mark them active as well. Updating last epoch started on residual OSDs implicitly renders them obsolete in that they know the PG became active in an epoch after their last update and

their information is likely out of date. In the future, a primary left with only obsolete information from its prior set can opt to either consider itself crashed or, if an administrator is desperate, bring the PG online with potentially stale data.

ii. Recovery- Recovery in RADOS is coordinated by the primary. Operations on missing objects are delayed until the primary has a local copy. Since the primary already knows which objects all replicas are missing from the peering process, it can preemptively "push" any missing objects that are about to be modified to replica OSDs, simplifying replication logic while also ensuring that the object is only read once. If a replica is handling reads, as in splay replication, requests for missing objects are delayed until the object can be pulled from the primary. If the primary is pushing an object, or if it has just pulled an object for itself, it will always push to all replicas that need a copy while it has the object in memory. Every re-replicated object is read only once.

iii. Client Participation- If a RADOS client has an outstanding request submitted for a PG that experiences a failure, it will simply resubmit the request to the new PG primary. This ensures that if the request was not completely replicated or otherwise did not survive the failure, it will still be processed. If the OSD discovers the request was already applied by the request's presence in the log, it will consider the operation a *no-op*, but will otherwise process it normally so that the client still receives an *ack* and *commit* with the same associated promises.

iv. Concurrent Failures- OSDs include the version associated with each update in the client *ack*, and the RADOS client buffers all updates it submits until a final *commit* is received. If a PG with which it has uncommitted updates crashes, the client includes the previously assigned version with the resubmitted request. When a crashed PG is recovering, OSDs enter a *replay* period for a fixed amount of times after peering but before becoming active. The primary OSD buffers requests such that when the replay period ends, it can reorder any requests that include versions to reestablish the original order of updates. This preserves consistency.

1. This system can also delay reads to uncommitted data, while taking steps to expedite their commit to disk. This approach maintains a low latency for writes, and only increases latency for reads if two operations are actually dependent. Alternatively, a small amount of NVRAM can be employed on the OSD for PG log storage, allowing serialization to be preserved across power failures such that resubmitted client operations can be correctly reordered, similarly preserving fully consistent semantics.

f. Client Locking and Caching- RADOS locks are issued and enforced by the OSDs that store objects. Read (shared) and write (exclusive) locks are implemented as object attributes, and lock acquisition and release behave like any other object update: they are serialized and replicated across all OSDs in the PG for consistency and safety. Locks can either time out or applications can empower a third party to revoke on behalf of failed clients. Also described is module layers on top of the RADOS client to manage client lock state and provide basic object caching services and multi-object updates. This transparently acquires the necessary locks to achieve proper cache consistency. Write locks can also be used to mask latency associated with large updates: ordering is established when the lock is acquired, and is released asynchronously as the data is written back to the OSD. Operations on multiple objects practice deadlock avoidance during lock acquisition.

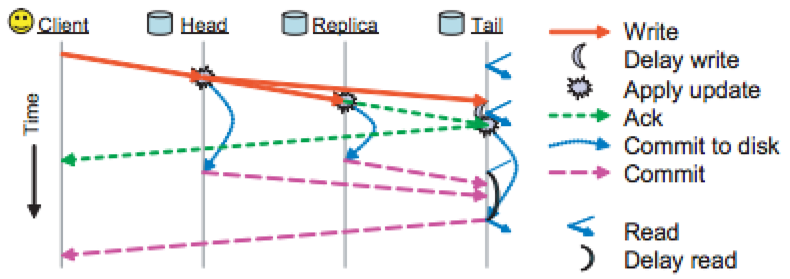

Figure 6.6: Reads in shared read/write workloads are usually unaffected by write operations. However, reads of uncommitted data can be delayed until the update commits. This increases read latency in certain cases, but maintains a fully consistent behavior for concurrent read and write operations in the event that all OSDs in the placement group simultaneously fail and the write *ack* is not delivered.

Future Work

RADOS is well-suited for a variety of other storage abstractions. In particular, the current interface based on reading and writing byte ranges is primarily an artifact of the intended usage for file data storage. Objects might have any query or update interface of resemble any number of fundamental data structures. Potential services include distributed B- link tree that map ordered keys to data value, high-performance distributed hash tables, or FIFO queries.

References: Weil, Sage A. CEPH: Reliable, Scalable, and High- Performance Distributed Storage http://ceph.com/papers/weil-thesis.pdf