Proceedings of the 2nd international Petascale Data Storage Workshop (PDSW '07)
held during Supercomputing '07. November 11, 2007, Reno, NV.

RADOS: A Scalable, Reliable Storage Service for Petabyte-scale Storage Clusters

Sage A. Weil Andrew W. Leung Scott A. Brandt Carlos Maltzahn
University of California, Santa Cruz
{sage,aleung,scott,carlosm}@cs.ucsc.edu

ABSTRACT

Brick and object-based storage architectures have emerged as a means of improving the scalability of storage clusters. However, existing systems continue to treat storage nodes as passive devices, despite their ability to exhibit significant intelligence and autonomy. We present the design and implementation of RADOS, a reliable object storage service that can scales to many thousands of devices by leveraging the intelligence present in individual storage nodes. RADOS preserves consistent data access and strong safety semantics while allowing nodes to act semi-autonomously to self-manage replication, failure detection, and failure recovery through the use of a small cluster map. Our implementation offers excellent performance, reliability, and scalability while providing clients with the illusion of a single logical object store.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Reliability, availability, and serviceability; D.4.3 [**File Systems Management**]: Distributed file systems; D.4.7 [**Organization and Design**]: Distributed systems

General Terms

design, performance, reliability

Keywords

clustered storage, petabyte-scale storage, object-based storage

1. INTRODUCTION

Providing reliable, high-performance storage that scales has been an ongoing challenge for system designers. High-throughput and low-latency storage for file systems, databases, and related abstractions are critical to the performance of a broad range of applications. Emerging clustered storage architectures constructed from storage bricks or object storage devices (OSDs) seek to distribute low-level block allocation decisions and security enforcement to intelligent storage devices, simplifying data layout and eliminating I/O bottlenecks by facilitating direct client access to data [1, 6, 7, 8, 21, 27, 29, 31]. OSDs constructed from commodity components combine a CPU, network interface, and local cache with an underlying disk or RAID, and replace the convention block-based storage interface with one based on named, variable-length objects.

However, systems adopting this architecture largely fail to exploit device intelligence. As in conventional storage systems based on local or network-attached (SAN) disk drives or those embracing the proposed T10 OSD standard, devices passively respond to read and write commands, despite their potential to encapsulate significant intelligence. As storage clusters grow to thousands of devices or more, consistent management of data placement, failure detection, and failure recovery places an increasingly large burden on client, controller, or metadata directory nodes, limiting scalability.

We have designed and implemented RADOS, a Reliable, Autonomic Distributed Object Store that seeks to leverage device intelligence to distribute the complexity surrounding consistent data access, redundant storage, failure detection, and failure recovery in clusters consisting of many thousands of storage devices. Built as part of the Ceph distributed file system [27], 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.

At the petabyte scale, storage systems are necessarily dynamic: they are built incrementally, they grow and contract with the deployment of new storage and decommissioning of old devices, devices fail and recover on a continuous basis, and large amounts of data are created and destroyed. RADOS ensures a consistent view of the data distribution and consistent read and write access to data objects through the use of a versioned *cluster map*. The map is replicated by all parties (storage and client nodes alike), and updated by lazily propagating small incremental updates.

By providing storage nodes with complete knowledge of the distribution of data in the systems, devices can act semi-autonomously using peer-to-peer like protocols to self-manage data replication, consistently and safely process updates, participate in failure detection, and respond to device fail-

Supercomputing'07 Nov. 10-16, 2007, Reno, NV

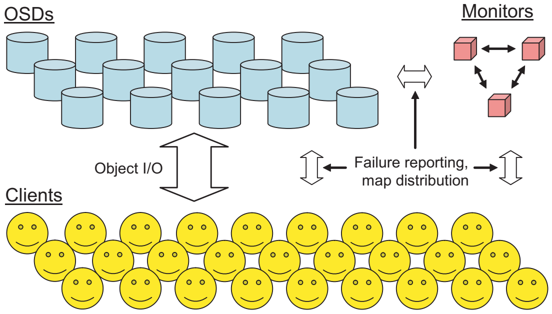

Figure 1: A cluster of many thousands of OSDs store all objects in the system. A small, tightly coupled cluster of monitors collectively manages the cluster map that specifies cluster membership and the distribution of data. Each client exposes a simple storage interface to applications.

ures and the resulting changes in the distribution of data by re-replicating or migrating data objects. This eases the burden on the small *monitor cluster* that manages the master copy of the cluster map and, through it, the rest of the storage cluster, enabling the system to seamlessly scale from a few dozen to many thousands of devices.

Our prototype implementation exposes an object interface in which byte extents can be read or written (much like a file), as that was our initial requirement for Ceph. Data objects are replicated n ways across multiple OSDs to protect against node failures. However, the scalability of RADOS is in no way dependent on the specific object interface or redundancy strategy; objects that store key/value pairs and parity-based (RAID) redundancy are both planned.

2. SCALABLE CLUSTER MANAGEMENT

A RADOS system consists of a large collection of OSDs and a small group of monitors responsible for managing OSD cluster membership (Figure 1). Each OSD includes a CPU, some volatile RAM, a network interface, and a locally attached disk drive or RAID. Monitors are stand-alone processes and require a small amount of local storage.

2.1 Cluster Map

The storage cluster is managed exclusively through the manipulation of the *cluster map* by the monitor cluster. The map specifies which OSDs are included in the cluster and compactly specifies the distribution of all data in the system across those devices. It is replicated by every storage node as well as clients interacting with the RADOS cluster. Because the cluster map completely specifies the data distribution, clients expose a simple interface that treats the entire storage cluster (potentially tens of thousands of nodes) as a single logical object store.

Each time the cluster map changes due to an OSD status change (*e. g.*, device failure) or other event effecting data layout, the map *epoch* is incremented. Map epochs allow communicating parties to agree on what the current distribution of data is, and to determine when their information is (relatively) out of data. Because cluster map changes may be frequent, as in a very large system where OSDs failures and recoveries are the norm, 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
up:	OSD $\mapsto$ { network address, *down* }
in:	OSD $\mapsto$ { *in*, *out* }
m:	number of placement groups ($2^k - 1$)
crush:	CRUSH hierarchy and placement rules

Table 1: The cluster map specifies cluster membership, device state, and the mapping of data objects to devices. The data distribution is specified first by mapping objects to placement groups (controlled by *m*) and then mapping each PG onto a set of devices (CRUSH).

2.2 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 is robust in that it maintains a probabilistically balanced distribution that, on average, keeps all devices similarly loaded, allowing the system to perform well under any potential workload [22]. Most importantly, data placement is a two stage process that *calculates* the proper location of objects; no large or cumbersome centralized allocation table is needed.

Each object stored by the system is first mapped into a *placement group* (PG), a logical collection of objects that are replicated by the same set of devices. 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. As the cluster scales, it is periodically necessary to adjust the total number of placement groups by changing m; this is done gradually to throttle the resulting migration of PGs between devices.

Placement groups are assigned to OSDs based on the cluster map, which maps each PG to an ordered list of r OSDs upon which to store object replicas. Our implementation utilizes CRUSH, a robust replica distribution algorithm that calculates a stable, pseudo-random mapping [28]. (Other placement strategies are possible; even an explicit table mapping each PG to a set of devices is still relatively small (megabytes) even for extremely large clusters.) From a high level, 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. In contrast, hashing approaches typically force a reshuffle of all prior mappings. CRUSH also uses weights to control the relative amount of data assigned to each device based on its capacity or performance.

Placement groups provide a means of controlling the level of replication declustering. That is, instead of an OSD sharing all of its replicas with one or more devices (mirroring), or with sharing each object with different device(s) (complete declustering), the number of replication peers is related to the number of PGs μ it stores—typically on the order of 100 PGs per OSD. Because distribution is stochastic, μ also affects the variance in device utilizations: more PGs per OSD 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.

2.3 Device State

The cluster map includes a description and current state of devices over which data is distributed. This includes the current network address of all OSDs that are currently online and reachable (*up*), and an 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.

OSDs are normally both *up* and *in* the mapping to actively service I/O, or both *down* and *out* if they have failed, producing an active list of exactly r OSDs. OSDs may also be *down* but still *in* the mapping, meaning that they are currently unreachable but PG data has not yet been remapped to another OSD (similar to the "degraded mode" in RAID systems). Likewise, they may be *up* and *out*, meaning they are online but idle. This facilitates a variety of scenarios, including tolerance of intermittent periods of unavailability (*e. g.*, an OSD reboot or network hiccup) without initiating any data migration, the ability to bring newly deployed storage online without using it immediately (*e. g.*, to allow the network to be tested), and the ability to safely migrate data off old devices before they are decommissioned.

2.4 Map Propagation

Because the RADOS cluster may include many thousands of devices or more, it is not practical to simply broadcast map updates to all parties. Fortunately, 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 the particular PG the I/O references. This property allows RADOS to distribute map updates lazily by combining them with existing inter-OSD messages, effectively shifting the distribution burden to OSDs.

Each OSD maintains a history of past incremental map updates, tags all messages with its latest epoch, and keeps track of the most recent epoch observed to be present at each peer. 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 (see Section 3.3) ensure that updates spread quickly—in $O(\log n)$ time for a cluster of n OSDs.

For example, when an OSD first boots, it begins by informing a monitor (see Section 4) that is has come online with a *OSDBoot* message that includes its most recent map epoch. The monitor cluster changes the OSD's status to *up*, and replies with the incremental updates necessary to bring the OSD fully up to date. When the new OSD begins contacting OSDs with whom it shares data (see Section 3.4.1), the exact set of devices who are affected by its status change learn about the appropriate map updates. Because a booting OSD does not yet know exactly which epochs its peers have, it shares a safe recent history (at least 30 seconds) of incremental updates.

This preemptive map sharing strategy is conservative: an OSD will always share an update when contacting a peer unless it is certain the peer has already seen it, resulting in OSDs receiving duplicates of the same update. However, the number of duplicates an OSD receives is bounded by the number of peers it has, which is in turn determined by the number of PGs μ it manages. In practice, we find that the actual level of update duplication is much lower than this (see Section 5.1).

3. INTELLIGENT STORAGE DEVICES

The knowledge of the data distribution encapsulated in the cluster map allows RADOS to distribute management of data redundancy, failure detection, and failure recovery to the OSDs that comprise the storage cluster. This exploits the intelligence present in OSDs by utilizing peer to peer-like protocols in a high-performance cluster environment.

RADOS currently implements n-way replication combined with per-object versions and short-term logs for each PG. Replication is performed by the OSDs themselves: clients submit a single write operation to the first *primary* OSD, who is then responsible for consistently and safely updating all replicas. This shifts replication-related bandwidth to the storage cluster's internal network and simplifies client design. Object versions and short-term logs facilitate fast recovery in the event of intermittent node failure (*e. g.*, a network disconnect or node crash/reboot).

We will briefly describe how the RADOS cluster architecture—in particular, the cluster map—enables distributed replication and recovery operations, and how these capabilities can be generalized to include other redundancy mechanisms (such as parity-based RAID codes).

3.1 Replication

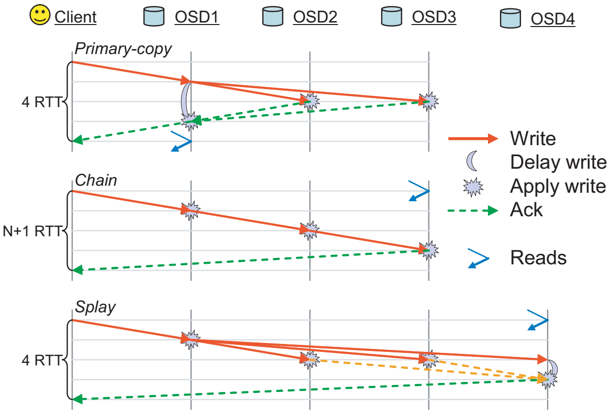

Figure 2: Replication strategies implemented by RADOS. Primary-copy processes both reads and writes on the first OSD and updates replicas in parallel, while chain forwards writes sequentially and processes reads at the tail. Splay replication combines parallel updates with reads at the tail to minimize update latency.

RADOS implements three replication schemes: primary-copy [3], chain [26], and a hybrid we call *splay* replication. The messages exchanged during an update operation are shown in Figure 2. In all cases, clients send I/O operations to a single (though possibly different) OSD, and the cluster ensures that replicas are safely updated and consistent read/write semantics (*i. e.*, serializability) are preserved. Once all replicas are updated, a single acknowledgement is returned to the client.

Primary-copy replication updates all replicas in parallel, and processes both reads and writes at the primary OSD. Chain replication instead updates replicas in series: writes are sent to the primary (head), and reads to the tail, ensuring that reads always reflect fully replicated updates. Splay replication simply combines the parallel updates of primary-copy replication with the read/write role separation of chain replication. The primary advantage is a lower number of message hops for 2-way mirroring.

3.2 Strong Consistency

All RADOS messages—both those originating from clients and from other OSDs—are tagged with the sender's map epoch to ensure that all update operations are applied in a fully consistent fashion. If a client sends an I/O to the wrong OSD due to an out of data map, the OSD will respond with the appropriate incrementals so that the client can redirect the request. This avoids the need proactively share map updates with clients: they will learn about them as they interact with the storage cluster. In most cases, they will learn about updates that do not affect the current operation, but allow future I/Os to be directed accurately.

If the master copy of the cluster map has been updated to change a particular PGs membership, updates may still be processed by the old members, provided they have not yet heard of the change. If the change is learned by a PG replica first, it will be discovered when the primary OSD forwards updates to replicas and they respond with the new incremental map updates. This is completely safe because any set of OSDs who are newly responsible for a PG are required to contact all previously responsible (non-failed) nodes in order to determine the PGs correct contents; this ensures that prior OSDs learn of the change and stop performing I/O before newly responsible OSDs start.

Achieving similar consistency for read operations is slightly less natural than for updates. In the event of a 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 processed by the new one, we require timely heartbeat messages between OSDs in each PG in order for the PG to remain readable. That is, if the OSD servicing reads hasn't heard from other replicas in H seconds, reads will block. Before another OSD to take over the primary role for a PG, 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. In the current implementation, we use a relatively short heartbeat interval of two seconds. This ensures both timely failure detection and a short interval of PG data unavailability in the event of a primary OSD failure.

3.3 Failure Detection

RADOS employs an asynchronous, ordered point to point message passing library for communication. A failure on the TCP socket results in a limited number of reconnect attempts before a failure is reported to the monitor cluster. Storage nodes exchange periodic heartbeat messages with their peers (those OSDs with whom they share PG data) to ensure that device failures are detected. OSDs that discover that they have been marked *down* simply sync to disk and kill themselves to ensure consistent behavior.

3.4 Data Migration and Failure Recovery

RADOS data migration and failure recovery are driven entirely by cluster map updates and subsequent changes in the mapping placement groups to OSDs. 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—device failure is simply one of many possible causes of the generalized problem of establishing a new distribution of data across the storage cluster.

RADOS makes no continuity assumptions about data distribution between one map and the next. In all cases, RADOS employs a robust *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 a PG log and its record of what the current contents of a PG *should* be (*i. e.*, what object versions it contains), even when object replicas may be missing locally. Thus, even 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.

3.4.1 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 must re-peer. Considering all map epochs (not just the most recent) 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. As with replication, peering (and any subsequent recovery) proceeds independently for every PG in the system.

Peering 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 (*i. e.*, it is a *replica*, or a *stray* which is longer in the active set), a *Notify* message is sent to the current primary. This message includes basic state information about the locally stored PG, including the most recent update, bounds of the PG log, and the most recent known epoch during which the PG successfully peered. Notify messages ensure that a new primary for a PG discovers its new role without having to consider all possible PGs (of which there may be millions) for every map change. Once aware, the primary generates a *prior set*, which includes all OSDs that may have participated in the PG since it was last successfully peered. 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 (*e. g.*, if peering never completed for an intermediate PG mapping).

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. If available PG logs are insufficient (*e. g.*, if one or more OSDs has no data for the PG), a list of the complete PG contents is generated. For node reboots or other short outages, however, this is not necessary—the recent PG logs are sufficient to quickly resynchronize replicas.

Finally, the primary OSD shares missing log fragments with replica OSDs, such that all replicas know what objects the PG *should* contain (even if they are still missing locally), and begins processing I/O while recovery proceeds in the background.

3.4.2 Recovery

A critical advantage of declustered replication is the ability to parallelize failure recovery. Replicas shared with any single failed device are spread across many other OSDs, and each PG will independently choose a replacement, allowing re-replication to just as many more OSDs. On average, in a large system, any OSD involved in recovery for a single failure will be either pushing or pulling content for only a single PG, making recovery very fast.

Recovery in RADOS is motivated by the observation that I/O is most often limited by read (and not write) throughput. Although each individual OSD, armed with all PG metadata, could independently fetch any missing objects, this strategy has two limitations. First, multiple OSDs independently recovering objects in the same PG they will probably not pull the same objects from the same OSDs at the same time, resulting in duplication of the most expensive aspect of recovery: seeking and reading. Second, the update replication protocols (described in Section 3.1) become increasingly complex if replica OSDs are missing the objects being modified.

For these reasons, PG recovery in RADOS is coordinated by the primary. As before, 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 surviving copy of the object is only read once. If the primary is pushing an object (*e. g.*, in response to a pull request), 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. Thus, in the aggregate, every re-replicated object is read only once.

4. MONITORS

A small cluster of *monitors* are collectively responsible for managing the storage system by storing the master copy of the cluster map and making periodic updates in response to configuration changes or changes in OSD state (*e. g.*, device failure or recovery). The cluster, which is based in part on the Paxos part-time parliament algorithm [14], is designed to favor consistency and durability over availability and update latency. Notably, a majority of monitors must be available in order to read or update the cluster map, and changes are guaranteed to be durable.

4.1 Paxos Service

The cluster is based on a distributed state machine service, based on the Paxos, in which the cluster map is the current machine *state* and each successful update results in a new map *epoch*. The implementation simplifies standard Paxos slightly by allowing only a single concurrent map mutation at a time (as in Boxwood [17]), while combining the basic algorithm with a lease mechanism that allows requests to be directed at any monitor while ensuring a consistent ordering of read and update operations. [1]

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 (currently two seconds) to respond to the probe and join the *quorum*. If a majority of the monitors are active, the first phase of the Paxos algorithm ensures that each monitor has the most recent committed map epoch (requesting incremental updates from other monitors as necessary), and then begins distributing short-term leases to active monitors.

Each lease grants active monitors permission to distribute copies of the cluster map to OSDs or clients who request

[1] This is implemented as a generic service and used to manage a variety of other global data structures in Ceph, including the MDS cluster map and state for coordinating client access to the system.

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 (to establish a new quorum). When a monitor first boots up, or finds that a previously called election does not complete after a reasonable interval, an election is called.

When an active monitor receives an update request (*e. g.*, a failure report), it first checks to see if it is 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 aggregates updates. Periodically the leader will initiate a map update by incrementing the map epoch and using the Paxos update protocol to distribute the update proposal to other monitors, simultaneously revoking leases. If the update is acknowledged by a majority of monitors, a final commit message issues a new lease.

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—significantly, map versions will never "go backwards"—regardless of which monitor messages are sent to and despite any intervening monitor failures, provided a majority of monitors are available.

4.2 Workload and Scalability

In the general case, monitors do very little work: most map distribution is handled by storage nodes (see Section 2.4), and device state changes (*e. g.*, due to a device failure) are normally infrequent.

The leasing mechanism used internally by the monitor cluster allows any monitor to service reads from OSDs or clients requesting the latest copy of the cluster map. Such requests rarely originate from OSDs due to the preemptive map sharing, and clients request updates only when OSD operations time out and failure is suspected. The monitor cluster can be expanded to distribute this workload (beyond what is necessary purely for reliability) for large clusters.

Requests that require a map update (*e. g.*, OSD boot notifications or reports of previously unreported failures) are forwarded to the current leader. The leader aggregates multiple changes into a single map update, such that the frequency of map updates is tunable and independent of the size of the cluster. Nonetheless, a worst case load occurs when large numbers of OSDs appear to fail in a short period. If each OSD stores μ PGs and f OSDs fail, then an upper bound on the number of failure reports generated is on the order of μf, which could be very large if a large OSD cluster experiences a network partition. To prevent such a deluge of messages, OSDs send heartbeats at semi-random intervals to stagger detection of failures, and then throttle and batch failure reports, imposing an upper bound on monitor cluster load proportional to the cluster size. Non-leader monitors then forward reports of any given failure only once, such that the request workload on the leader is proportional to fm for a cluster of m monitors.

5. PARTIAL EVALUATION

Performance of the object storage layer (EBOFS) utilized by on each OSD has been previously measured in conjunction with Ceph [27]. Similarly, the data distribution properties of CRUSH and their effect on aggregate cluster throughput are evaluated elsewhere [27, 28]. In this short paper we focus only on map distribution, as that directly impacts the clusters' ability to scale. We have not yet experimentally evaluated monitor cluster performance, although we have confidence in the architecture's scalability.

5.1 Map Propagation

The RADOS map distribution algorithm (Section 2.4) ensures that updates reach all OSDs after only $\log n$ hops. However, as the size of the storage cluster scales, the frequency of device failures and related cluster updates increases. Because map updates are only exchanged between OSDs who share PGs, the hard upper bound on the number of copies of a single update an OSD can receive is proportional to μ.

In simulations under near-worst case propagation circumstances with regular map updates, we found that update duplicates approach a steady state even with exponential cluster scaling. In this experiment, the monitors share each map update with a single random OSD, who then shares it with its peers. In Figure 3 we vary the cluster size x and the number of PGs on each OSD (which corresponds to the number of peers it has) and measure the number of duplicate map updates received for every new one (y). Update duplication approaches a constant level—less than 20% of μ—even as the cluster size scales exponentially, implying a fixed map distribution overhead. We consider a worst case scenario in which the only OSD chatter are pings for failure detection, which means that, generally speaking, OSDs learn about map updates (and the changes known by their peers) as slowly as possible. Limiting map distribution overhead thus relies only on throttling the map update frequency, which the monitor cluster already does as a matter of course.

6. FUTURE WORK

Our current implementation has worked well as a basis for the Ceph distributed file system. However, the scalable cluster management services it provides are much more general than Ceph's requirements, and there are a number of additional research directions we are pursuing.

6.1 Key-value Storage

The reliable and scalable object storage service that RADOS provides is well-suited for a variety of non-file 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 or resemble any number of fundamental data structures. We are currently working on abstracting the specific object interface to allow key/value

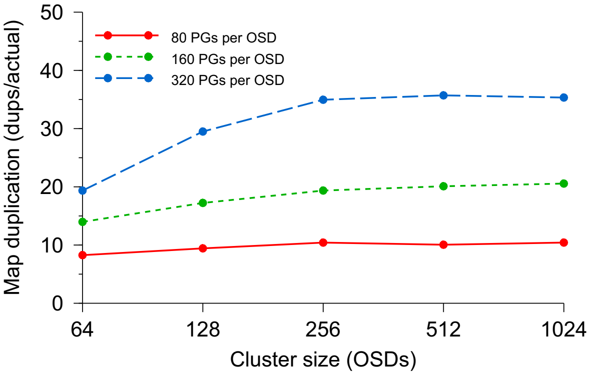

Figure 3: Duplication of map updates received by individual OSDs as the size of the cluster grows. The number of placement groups on each OSD effects number of peers it has who may share map updates.

storage, as with a dictionary or index data structure. This would facilitate distributed and scalable B-link trees that map ordered keys to data values (much like Boxwood [17]), as well as high-performance distributed hash tables [24]. The primary research challenge is to preserve a low-level extent interface that will allow recovery and replication code to remain simple and generic, and to facilitate alternative redundancy strategies (such as parity-based RAID codes) that are defined in terms of byte ranges.

6.2 Scalable FIFO Queues

Another storage data structure that is often required at scale is a FIFO queue, like that provided by GFS [7]. Unlike GFS, however, we hope to create a distributed and scalable FIFO data structure in RADOS with reliable queue insertion.

6.3 Object-granularity Snapshot

Many storage systems provide snapshot functionality on a volume-wide basis, allowing a logical point-in-time copy of the system to be created efficiently. As systems grow to petabytes and beyond, however, it becomes increasingly doubtful that a global snapshot schedule or policy will be appropriate for all data. We are currently implementing an object-granularity *clone* operation to create object copies with copy-on-write behavior for efficient storage utilization, and are extending the RADOS client interface to allow transparent versioning for logical point-in-time copies across sets of objects (*i. e.*, files, volumes, etc.). Although this research is being driven by our efforts to incorporate flexible snapshot-like functionality in Ceph, we expect it to generalize to other applications of RADOS.

6.4 Load Balancing

Although RADOS manages scalability in terms of total aggregate storage and capacity, we have not yet addressed the issue of many clients accessing a single popular object. We have done some preliminary experimentation with a *read shedding* mechanism which allows a busy OSD to shed reads to object replicas for servicing, when the replica's OSD has a lower load and when consistency allows (*i. e.*, there are no conflicting in-progress updates). Heartbeat messages exchange information about current load in terms of recent average read latency, such that OSDs can determine if a read is likely to be service more quickly by a peer. This facilitates fine-grained balancing in the presence of transient load imbalance, much like D-SPTF [16]. Although preliminary experiments are promising, a comprehensive evaluation has not yet been conducted.

More generally, the distribution of workload in RADOS is currently dependent on the quality of the data distribution generated by object layout into PGs and the mapping of PGs to OSDs by CRUSH. Although we have considered the statistical properties of such a distribution and demonstrated the effect of load variance on performance for certain workloads, the interaction of workload, PG distribution, and replication can be complex. For example, write access to a PG will generally be limited by the slowest device storing replicas, while workloads may be highly skewed toward possibly disjoint sets of heavily read or written objects. To date we have conducted only minimal analysis of the effects of such workloads on efficiency in a cluster utilizing declustered replication, or the potential for techniques like read shedding to improve performance in such scenarios.

6.5 Quality of Service

The integration of intelligent disk scheduling, including the prioritization of replication versus workload and quality of service guarantees, is an ongoing area of investigation within the research group [32].

6.6 Parity-based Redundancy

In addition to n-way replication, we would also like to support parity-based redundancy for improved storage efficiently. In particular, intelligent storage devices introduce the possibility of seamlessly and dynamically adjusting the redundancy strategy used for individual objects based on their temperature and workload, much like AutoRAID [30] or Ursa Minor [1].

In order to facilitate a broad range of parity-based schemes, we would like to incorporate a generic engine such as REO [12]. Preserving the existing client protocol currently used for replication—in which client reads and writes are serialized and/or replicated by the primary OSD—would facilitate flexibility in the choice of encoding and allow the client to remain ignorant of the redundancy strategy (replication or parity-based) utilized for a particular object. Although data flow may be non-ideal in certain cases—a client could write each parity fragments directly to each OSD—aggregate network utilization is only slightly greater than the optimum [11], and often better than straight replication.

7. RELATED WORK

Most distributed storage systems utilize centralized metadata servers [1, 4, 7] or collaborative allocation tables [23] to manage data distribution, ultimately limiting system scalability. For example, like RADOS, Ursa Minor [1] provides a distributed object storage service (and, like Ceph, layers a file system service on top of that abstraction). In contrast to RADOS, however, Ursa Minor relies on an object manager to maintain a directory of object locations and storage strategies (replication, erasure coding, etc.), limiting scalability and placing a lookup in the data path. Although the architecture could allow it, our implementation does not currently provide the same versatility as Ursa Minor's dynamic choice of timing and failure models, or support for online changes to object encoding (although encoding changes are planned for the future); instead, we have focused on scalable performance in a relatively controlled (non-Byzantine) environment.

The Sorrento [25] file system's use of collaborative hashing [10] bears the strongest resemblance to RADOS's application of CRUSH. Many distributed hash tables (DHTs) use similar hashing schemes [5, 19, 24], but these systems do not provide the same combination of strong consistency and performance that RADOS does. For example, DHTs like PAST [19] rely on an overlay network [20, 24, 34] in order for nodes to communicate or to locate data, limiting I/O performance. More significantly, objects in PAST are immutable, facilitating cryptographic protection and simplifying consistency and caching, but limiting the systems usefulness as a general storage service. CFS [5] utilizes the DHash DHT to provide a distributed peer-to-peer file service with cryptographic data protection and good scalability, but performance is limited by the use of the Chord [24] overlay network. In contrast to these systems, RADOS targets a high-performance cluster or data center environment; a compact cluster map describes the data distribution, avoiding the need for an overlay network for object location or message routing.

Most existing object-based storage systems rely on controllers or metadata servers to perform recovery [4, 18], or centrally micro-manage re-replication [7], failing to leverage intelligent storage devices. Other systems have adopted declustered replication strategies to distribute failure recovery, including OceanStore [13], Farsite [2], and Glacier [9]. These systems focus primarily on data safety and secrecy (using erasure codes or, in the case of Farsite, encrypted replicas) and wide-area scalability (like CFS and PAST), but not performance. FAB (Federated Array of Bricks) [21] provides high performance by utilizing two-phase writes and voting among bricks to ensure linearizability and to respond to failure and recovery. Although this improves tolerance to intermittent failures, multiple bricks are required to ensure consistent read access, while the lack of complete knowledge of the data distribution further requires coordinator bricks to help conduct I/O. FAB can utilize both replication and erasure codes for efficient storage utilization, but relies on the use of NVRAM for good performance. In contrast, RADOS's cluster maps drive consensus and ensure consistent access despite a simple and direct data access protocol.

Xin *et al.* [33] conduct a quantitative analysis of reliability with FaRM, a declustered replication model in which—like RADOS—data objects are pseudo-randomly distributed among placement groups and then replicated by multiple OSDs, facilitating fast parallel recovery. They find that declustered replication improves reliability at scale, particularly in the presence of relatively high failure rates for new disks ("infant mortality"). Lian *et al.* [15] find that reliability further depends on the number of placement groups per device, and that the optimal choice is related to the amount of bandwidth available for data recovery versus device bandwidth. Although both consider only independent failures, RADOS leverages CRUSH to mitigate correlated failure risk with failure domains.

8. CONCLUSION

RADOS provides a scalable and reliable object storage service without compromising performance. The architecture 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.

Acknowledgments

This work was supported in part by the Department of Energy under award DE-FC02-06ER25768, the National Science Foundation under award CCF-0621463, the Institute for Scalable Scientific Data Management, and by the industrial sponsors of the Storage Systems Research Center, including Agami Systems, Data Domain, DigiSense, Hewlett Packard Laboratories, LSI Logic, Network Appliance, Seagate Technology, Symantec, and Yahoo. We thank the members of the SSRC, whose advice helped guide this research and the anonymous reviewers for their insightful feedback.

9. REFERENCES

[1] M. Abd-El-Malek, W. V. C. II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa minor: versatile cluster-based storage. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST)*, pages 59–72, San Francisco, CA, Dec. 2005.

[2] A. Adya, W. J. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, Dec. 2002. USENIX.

[3] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 562–570. IEEE Computer Society Press, 1976.

[4] P. J. Braam. The Lustre storage architecture. http://www.lustre.org/documentation.html, Cluster File Systems, Inc., Aug. 2004.

[5] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 202–215, Banff, Canada, Oct. 2001. ACM.

[6] G. R. Ganger, J. D. Strunk, and A. J. Klosterman. Self-* storage: Brick-based storage with automated administration. Technical Report CMU-CS-03-178, Carnegie Mellon University, 2003.

[7] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, NY, Oct. 2003. ACM.

[8] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 92–103, San Jose, CA, Oct. 1998.

[9] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, May 2005. USENIX.

[10] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *ACM Symposium on Theory of Computing*, pages 654–663, May 1997.

[11] D. R. Kenchammana-Hosekote, R. A. Golding, C. Fleiner, and O. A. Zaki. The design and evaluation of network raid protocols. Technical Report RJ 10316 (A0403-006), IBM Research, Almaden Center, Mar. 2004.

[12] D. R. Kenchammana-Hosekote, D. He, and J. L. Hafner. Reo: A generic raid engine and optimizer. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, pages 261–276, San Jose, CA, Feb. 2007. Usenix.

[13] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Cambridge, MA, Nov. 2000. ACM.

[14] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[15] Q. Lian, W. Chen, and Z. Zhang. On the impact of replica placement to the reliability of distributed brick storage systems. In *Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS '05)*, pages 187–196, Los Alamitos, CA, 2005. IEEE Computer Society.

[16] C. R. Lumb, G. R. Ganger, and R. Golding. D-SPTF: Decentralized request distribution in brick-based storage systems. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 37–47, Boston, MA, 2004.

[17] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, Dec. 2004.

[18] D. Nagle, D. Serenyi, and A. Matthews. The Panasas ActiveScale storage cluster—delivering scalable high bandwidth storage. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC '04)*, Nov. 2004.

[19] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 188–201, Banff, Canada, Oct. 2001. ACM.

[20] A. Rowstrong and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, 2001.

[21] Y. Saito, S. Frølund, A. Veitch, A. Merchant, and S. Spence. FAB: Building distributed enterprise disk arrays from commodity components. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 48–58, 2004.

[22] J. R. Santos, R. R. Muntz, and B. Ribeiro-Neto. Comparing random data allocation and data striping in multimedia servers. In *Proceedings of the 2000 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 44–55, Santa Clara, CA, June 2000. ACM Press.

[23] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, pages 231–244. USENIX, Jan. 2002.

[24] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '01)*, pages 149–160, San Diego, CA, 2001.

[25] H. Tang, A. Gulbeden, J. Zhou, W. Strathearn, T. Yang, and L. Chu. A self-organizing storage cluster for parallel data-intensive applications. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC '04)*, Pittsburgh, PA, Nov. 2004.

[26] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 91–104, San Francisco, CA, Dec. 2004.

[27] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long,

and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, Nov. 2006. USENIX.

[28] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '06)*, Tampa, FL, Nov. 2006. ACM.

[29] B. Welch and G. Gibson. Managing scalability in object storage systems for HPC Linux clusters. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 433–445, Apr. 2004.

[30] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 96–108, Copper Mountain, CO, 1995. ACM Press.

[31] T. M. Wong, R. A. Golding, J. S. Glider, E. Borowsky, R. A. Becker-Szendy, C. Fleiner, D. R. Kenchammana-Hosekote, and O. A. Zaki. Kybos: self-management for distributed brick-based storage. Research Report RJ 10356, IBM Almaden Research Center, Aug. 2005.

[32] J. C. Wu and S. A. Brandt. The design and implementation of AQuA: an adaptive quality of service aware object-based storage device. In *Proceedings of the 23rd IEEE / 14th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 209–218, College Park, MD, May 2006.

[33] Q. Xin, E. L. Miller, T. J. Schwarz, D. D. E. Long, S. A. Brandt, and W. Litwin. Reliability mechanisms for very large storage systems. In *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 146–156, Apr. 2003.

[34] B. Y. Zhao, L. gHuang, J. Stribling, S. C. Rhea, and A. D. J. nad John D. Kubiatowicz. Tapestry: A global-scale overlay for rapid service deployment. *IEEE Journal on Selected Areas in Communications*, 22:41–53, Jan. 2003.