The Case for Programmable Object Storage Systems

Noah Watkins Michael Sevilla Carlos Maltzahn
University of California, Santa Cruz
{jayhawk,msevilla,carlosm}@soe.ucsc.edu

Abstract

As applications scale to new levels and migrate into cloud environments, there has been a significant departure from the exclusive reliance on the POSIX file I/O interface. However in doing so, application often discover a lack of services, forcing them to use bolt-on features or take on the responsibility of critical data management tasks. This often results in duplication of complex software with extreme correctness requirements. Instead, wouldn't it be nice if an application could just convey what it wanted out of a storage system, and have the storage system understand?

The central question we address in this paper is whether or not the design delta between two storage systems can be expressed in a form such that one system becomes little more than a configuration of the other. Storage systems should expose their useful services in a way that separates performance from correctness, allowing for their safe reuse. After all, hardened code in storage systems protects countless value, and its correctness is only as good as the stress we place on it. We demonstrate these concepts by synthesizing the CORFU high-performance shared-log abstraction in Ceph through minor modifications of existing sub-systems that are orthogonal to correctness.

1. Introduction

Applications are increasingly opting for non-POSIX file I/O interfaces in order to simplify design, increase scalability, and enhance performance. As a result, complex middleware or entirely new storage systems are being constructed to meet the demand. This trend has resulted in the duplication of mission critical software that underlies virtually all storage systems—software which takes often many years of code hardening to reach an acceptable level of correctness.

Submission to SoCC '15, August, 2015, Hawaii, USA.

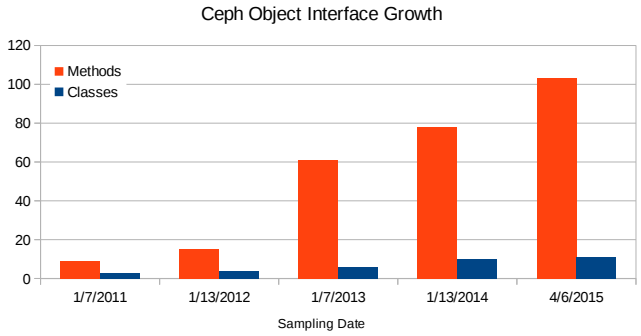

Figure 1. Growth in the use of co-designed object storage interfaces in Ceph.

But this duplication isn't without reason. Often domain-specific features and optimizations are necessary. What is needed is first-class support for storage system extensibility enabling applications and providers alike to exploit system services in order construct domain-specific storage interfaces without duplicating hardened software that protects information of untold value.

Storage sub-systems are typically designed from day-one as internal services, leading to a design that tightly intertwines safety and performance engineering goals. Thus, exposing these reusable services may be ad-hoc at best, and at worst opens up many channels of interference between monolithic system components. Existing approaches to extensibility have primarily focused on code injection (e.g. active storage [14]). For instance, Figure 1 shows a dramatic growth in new object interfaces available in Ceph via code-injection techniques. While this growth serves as an indicator of value, interface development and management doesn't currently match the needs of developers. Tied to sparse system upgrade cycles, interface changes typically require client and server restarts resulting in increased downtime the more frequently the feature is utilized.

In contrast to software-defined storage that focuses on centralized policies controlling fixed sets of abstractions, we argue for the creation of *programmable storage* that virtualizes, and safely exposes, well-trusted system services allowing new interfaces to be packaged, versioned, and dynam-

ically deployed allowing developers to custom tailor their "view" of the system.

The standardization of the POSIX file I/O interface has been a major success, allowing application developers to avoid vendor lock-in. However, large-scale storage systems have been dominated by proprietary products, preventing exploration of alternative interfaces and complicating future migration paths, eliminating the benefits of commodity systems. But the recent availability of high-performance open-source storage systems is changing this because these systems are modifiable, enabling interface change, and reducing the risks of lock-in. While vendor lock-in can largely be avoided through open-source usage, platform lock-in is a concern as applications become tied to a modified system. However, by finding and exposing common, key customization points in a system that are orthogonal to correctness, the cost of porting system extensions can be minimized.

In the remainder of this paper we demonstrate how a non-trivial storage interface can be constructed in an existing open-source storage system with relatively small modification to common storage system services. In particular, we reuse existing sub-systems in the Ceph distributed storage system [16] to synthesize the CORFU high-performance shared-log abstraction [2].

2. Programmable Storage

When application goals are not met by a storage system the most common reaction is to design a workaround. Workarounds roughly fall into one of two categories: so called "bolt-on" services that introduce a 3rd party system (e.g. a metadata service), or expanded application responsibility in the form of data management (e.g. a new data layout).

Extra Services. "Bolt-on" services are designed to improve overall application performance, but come at the expensive of additional sub-systems and dependencies that the application must manage, as well as trust. For example, it is well understood that MapReduce performs poorly for iterative and interactive computation due to its failure model that heavily relies on on-disk storage of intermediate data. Many have added services to Hadoop to keep more data in the runtime (e.g., HaLoop [8], Twister [11], CGL-MapReduce [10], MixApart [13]). While performance improves, it comes at a cost: "bolt-on" services frequently result in overly complex systems that re-implement functionality and re-execute redundant code, unnecessarily increasing the likelihood of bugs.

Application Changes. The second approach to adapting to a storage system deficiency is to change the application itself by adding more data management intelligence, often into the application itself, or as domain-specific middleware. For instance, an application may change to exploit data locality or I/O parallelism in a distributed storage system. This isn't a bad proposition, but creates a coupling that is highly tied to the underlying physical properties of the system, making it difficult to adapt to future changes at the storage system level.

Category	Specialization	Methods
Locking	Shared Exclusive	6
Logging	Replica State Timestamped	3 4 4
Garbage Collection	Reference Counting	4
Metadata Management	RBD RGW User Version	37 27 5 5

Table 1. A variety of RADOS object storage classes exist that expose reusable interfaces to applications.

Storage Changes. When these two approaches fail to meet an application's needs, developers turn their attention to the storage system itself. For example, HDFS has been the focus of scalability concerns, especially for metadata-intensive workloads [15]. This has lead to modifications to its architecture or API [4] to improve performance. Yet another approach is to "modify" a storage system using auto-tuning techniques that attempt to find a good solution among a huge space of available system configurations. However, in practice auto-tuning is limited to only the configuration "knobs" that the storage system exposes (e.g. block size). For instance, auto-tuning may be capable of identifying instances in which new data layouts would benefit a workload, but unless the system can provide such a transformation, the option is left off the table.

2.1 Programmability

We advocate a new approach that we refer to as *storage programmability* which is a method by which an application communicates its requirements to the storage system in a way that allows the application to realize a new behavior without sacrificing the correctness of the underlying system. The general concept is not new—active storage research has advocated pushing computation closer to the data. For instance, active storage techniques are used extensively in production Ceph environments. Table 1 shows the wide-variety of object interfaces that have been co-designed with applications that run on top of Ceph, and Figure 1 shows a dramatic growth in the use of co-designed interfaces since 2011. While we consider active storage to be an excellent example of programmability, what separates our proposal from previous work is the observation that so much *more* of the storage system can be reused to construct advanced, domain-specific interfaces. Designing more programmability into storage systems has the following benefits:

1. Separation of storage performance engineering from reliability engineering, enabling unique optimizations without risking years of code hardening efforts.
2. Encourages developers to create a new stack of storage systems abstractions, both domain-specific and across domains (e.g. RBD and locking in Table 1).
3. Encourages developers to generalize sophisticated techniques in other fields, such as database query optimizers, for re-use in storage systems.
4. Provides the opportunity for a well-defined environment for evolving optimizations which cannot affect correctness.
5. Improves collaboration by informing commercial parallel file system vendors on the design of low-level APIs for their products so that they match the versatility of open-source storage systems.
6. Eases tension between versatile open source storage systems and reliable proprietary systems so they can work together to lead the community of storage designers.

Our notion of programmable storage differs from "software-defined storage" (SDS) in terms of goals and scope, although definitions of SDS are still in flux. According to a recent SNIA working draft [9] the primary goal of SDS is to control and facilitate flexible and dynamic provisioning of storage resources of different kinds, including flash memory and disk drives, to create a virtualized mapping between common storage abstractions (e.g. files, objects, and blocks) and storage devices taking data service objectives in terms of protection, availability, performance, and security into account. Programmable storage on the other hand facilitates the customization of storage system software to create new storage system services and abstractions.

In the remainder of this paper we will demonstrate the power of these ideas by synthesizing an entirely new storage system service in an existing system through configuration and small changes. Next we discuss programmable storage in more depth. Then we describe the CORFU system and show how its components can be mapped onto components found in existing systems. Finally we introduce new concepts to handle an optimization found in CORFU and finish by performing an evaluation.

3. A Distributed Shared Log Service

In this section we describe an existing storage system called CORFU [2] that exposes the abstraction of a high-performance shared log, and use the architecture throughout the remainder of this paper as a driving example for dynamically constructing new services in existing storage systems. The value of the shared log abstraction as a storage service is highlighted by its role as a fundamental building block in many distributed systems, as well as in several recent research efforts focused on cloud-based metadata management [3] and elastic database storage engines [5–7]. Next we'll provide an overview of the CORFU architecture and then extract analogies between its design and existing storage systems.

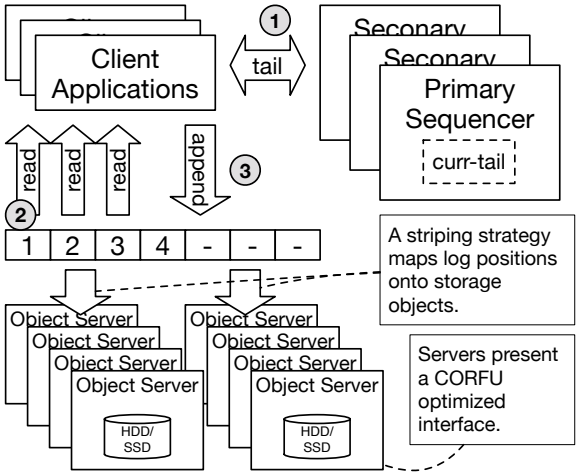

Figure 2. High-level view of the CORFU architecture on top of Ceph. Clients request positions from a sequencer service and interact directly with the log striped across a set of object servers.

3.1 The CORFU Shared-log Storage System

While the shared log abstraction is a very important building block in many distributed systems, typical implementations are based on consensus protocols such as Paxos [12] which serialize requests through a primary server, making throughput scalability a difficult property to achieve. We now provide an overview of the novel CORFU [2] log service that eliminates this bottleneck allowing log operations to scale to very high rates.

The high-level architecture of CORFU is shown in Figure 2. In the middle the abstract log is shown to which clients submit reads (2) and appends (3). These I/Os are directed directly at the storage system. In the original CORFU design a set of raw flash devices were used. We've tailored the diagram for our discussion and instead shown that the log is mapped onto a set of object storage servers according to a striping policy. Importantly, the sequencer service shown in the upper right side is responsible for assigning a global ordering to the log by responding to *tail* queries (3) made by clients when appending.

The key to high-performance in CORFU is two fold. First, clients perform all bulk I/O directly to storage targets enabling horizontal bandwidth scalability. However, this alone is not sufficient as append operations are serialized through the sequencer service that maintains a global ordering on log entries. The second feature of CORFU that enables high-performance is its novel protocol that allows the sequencer to be implemented as a volatile, in-memory

network counter, eliminating all I/O from the fast path, while still remaining robust to sequencer failures. For instance, the sequencer used in the CORFU paper was able to achieve 200K requests per second [2]. A complete description of the CORFU system is beyond the scope of this paper, and includes additional log operations such as trimming and position invalidation. Next we describe the salient features of the system that are used to achieve high-performance while maintaining correctness.

3.2 System Design

Maintaining correctness in CORFU is achieved through a combination of a co-designed storage interface and sequencer recovery protocol.

Sequencer. The sequencer component of CORFU is a network server that enforces log serializability by assigning sequential log positions to clients appending to a log. High-performance is achieved by storing the log tail exclusively in main memory, avoiding all I/O to persistent storage during normal operation. The challenge faced by the sequencer recovery process following a failure is to repopulate the cached tail value in a way that maintains correctness. To achieve this CORFU depends on a mechanism that tags all operations with an epoch value, and efficiently invalidates outstanding client requests between sequencer instances through the use of smart storage devices.

Storage Interface. The storage interface is a critical component in the CORFU design. Storage devices provide a relatively simple write-once, random read interface for reading and writing log entries. The key to correctness in CORFU lies with the enforcement of up-to-date epoch tags on client requests; requests tagged with out-of-date epoch values are rejected, and clients are expected to request a new tail from the sequencer. This mechanism forms the basis for sequencer recovery.

Sequencer Recovery. In order to repopulate the cached tail value during recovery of a sequencer, the maximum position in the log must be obtained. To do this, the storage interface exposes an additional *seal* method that returns the maximum log position that has been written to that device. In order to maintain correctness while racing with in-flight client requests, the seal method also takes a new epoch value that is registered with the device, rejecting all old requests and guaranteeing the validity of the maximum position until the sequencer has determined a global maximum, and resumes responding to client operations. Recovery of the sequencer server itself can be handled using an existing protocol such as leader election, or master-slave recovery as long as exactly one sequencer per logical log is active at a time.

Metadata Management. There are several pieces of metadata in CORFU that must be saved durably. This metadata includes the current epoch value, the current sequencer instance, and metadata related to log configuration such as the set of devices the log is being striped across. An auxiliary service implemented using Paxos is one way to manage this data as update performance is not critical; it is only used during sequencer recovery or system reconfiguration.

Fault-tolerance. Client-driven chained replication is used to achieve fault-tolerance and availability for log data in CORFU by organizing raw devices into replication groups. The state of storage devices (e.g. active, failed) is recorded in a cluster map and stored durably along side other metadata previously described. Clients update this map as they interact with the cluster of storage devices, and reconfigurations are propagated through the invalidation mechanism based on out-of-date epoch values.

3.3 Mapping CORFU to Common Storage Services

We now begin the process of mapping the components of CORFU onto commonly available storage sub-systems in an attempt to derive CORFU from customized versions of existing storage systems. First let's consider fault-tolerance. Virtually all storage systems today contain some form of fault-tolerance and availability, typically in some form of replication or erasure coding. Of course, for CORFU we are only interested in strong consistency so this eliminates some systems, but by and large, this is a first class sub-system found in distributed storage systems.

Enforcement of the CORFU protocol at the storage device layer is more challenging. While CORFU purposes a custom device-level interface, storage virtualization technologies allow new interfaces to be constructed at higher levels in systems software (e.g. a server daemon). Typically these features are only available to developers and privileged users such administrators, and closely resemble the ideas found in a large body of work related to active storage (e.g. [1, 14]).

Third, management of configuration information (e.g. cluster membership) in distributed storage systems closely mirrors the data management requirements of CORFU. For instance, all clients must agree on the striping strategy that maps log positions onto storage devices, and configuration changes must be propagated in a consistent way to avoid data loss.

The final component of CORFU, the sequencer, poses the largest challenge when identifying candidate sub-systems to customize. In CORFU a sequencer is effectively nothing more than a named service with domain-specific optimizations and service fail-over. Comparing this description to that of object extensibility highlighted above, it is easy to see the similarities. Specifically, objects with an extended interface behave as a named service, and when the system provides fault-tolerance and availability, service fail-over becomes a side effect. Unfortunately it isn't as easy to achieve the domain-specific optimizations used by the CORFU sequencer. The durability of object state achieved through the use of non-volatile media ensures less than optimal performance. Thus, in Section 5 we describe *programmable persistence* that allows us to realize a similar optimization.

There are additional benefits that arise when constructing interfaces within an existing system beyond taking advantage of institutional stability. The design of CORFU assumes a cluster of raw flash devices because log-centric systems tend to have a larger percentage of random reads making it difficult to achieve high-performance with spinning disks. However, the speed of the underlying storage does not affect correctness. Thus, in a software-defined storage system such as Ceph a single implementation can transparently take advantage of any software or hardware upgrades, and make use of data management features such as tiered storage, allowing users to freely choose between media types such as SSD, spinning disks, or future NVRAMs.

4. Programmable Object Storage Interface

We've chosen to implement the CORFU storage interface using extensibility features found in the RADOS object store [17] that underlies the Ceph storage system. Briefly, RADOS consists of a cluster of object storage servers (OSDs) to which clients direct object operations. Each OSD contains local resources such as CPU, memory, and various types of non-volatile media that store object data.

The object interface provided by RADOS is very rich, consisting of both bulk data I/O as well as key-value data stored internally in LevelDB or RocksDB. In addition to supporting multiple data models, many other features exist such as cloning, trimming, server-side caching hints, and tiered placement. One of the most powerful features in RADOS is the ability to extend object interfaces by grouping a set object operations along with arbitrary logic (via a C++ module) into an atomic transaction applied locally at an OSD. This allows powerful interfaces to be constructed such as atomic compare and swap. Figure 3 shows a high-level view of the internal OSD structure. A request first enters a complex request processing pipeline. A request is processed inside of a transaction by sequentially applying a set of user-defined methods to the object state which in turn access state in the storage layer.

One of the major challenges of providing extensibility as a service is to decide on how it is exposed. The extensibility features in Ceph are accessible exclusively through heavy-weight C++ modules that require distribution of architecture specific binaries, and servers and clients must be restarted when upgrades are made. This also has the side effect of effectively restricting the feature to system developers and administrators. In order to provide extensibility as a service, both to system providers, as well as developers seeking to construct application-specific features, a different mechanism is necessary.

In this paper we stop short of purposing a specific, comprehensive language for describing new interfaces. Rather, we acknowledge this as the primary goal of our on going research, and utilize a feature we have added to Ceph that allows us to use the Lua embedded language for dynamically

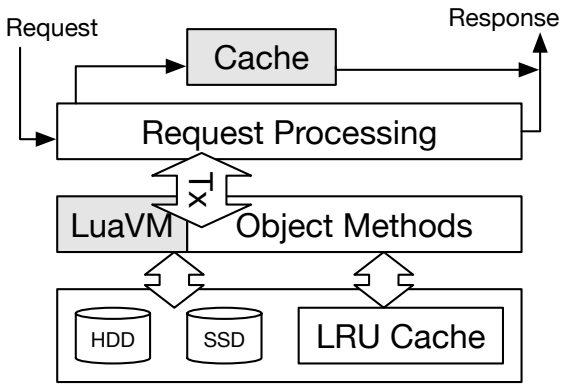

Figure 3. High-level architecture of RADOS OSD. The shaded components are what we have added or proposed.

altering the behavior of different components in Ceph. The Lua language provides a good baseline as it can achieve very high-performance which we use as a proxy for later language development. The shaded box labeled *LuaVM* in Figure 3 represents extensions we have made that expose the object methods in a transactional context through the Lua virtual machine enabling dynamic object methods to be executed.

4.1 The CORFU Storage Interface in RADOS

The transaction context available in RADOS enables the CORFU storage interface semantics to be built in a very compact way. In this particular case we have chosen to store all log entries in the underlying key/value store (e.g. LevelDB or RocksDB) that is provided as a basic service within Ceph as alternative data model for storing data in objects along side bulk byte-stream like data. Note that this decision to store log entries in the key/value store is related to the striping strategy. Other strategies exist such as mapping the entries onto the bulk binary data model and using the key/-value store as an index.

Recall from previous section, that every operation is tagged with an epoch to support client request invalidation during sequencer recovery. Listing 1 shows a helper function used to enforce up-to-date epoch tags on client requests. The Lua function takes as input an epoch value extracted from the request, and compares it to the stored epoch value contained in the object's key-value store, returning an error if the epoch is too old. Note that all unrecoverable I/O errors are unchecked; these are handled transparently by the Lua runtime and are returned to the client.

Next we examine the interface that clients use to write the contents of a log entry, shown in Listing 2. First the epoch guard shown in Listing 1 enforces that clients have an up-to-date view of the log. Next the write-once semantics of the CORFU storage interface are implemented by returning a read-only error if the log position has already been written. If the position is free then the entry is written, and the stored maximum log position is conditionally updated. The

```
def check_epoch(epoch)
  curr_epoch = omap_read("epoch.key")
  if epoch < curr_epoch
    return STALE_EPOCH
  else
    return OK
```

Listing 1: Helper function used to enforce up-to-date epoch tags on client requests.

important thing to take note of is that this entire function (including the check epoch helper) is executed atomically, enabling clients to simplify their design.

```
def write(op)
  ret = check_epoch(op.epoch)
  if ret != OK
    return ret

  if omap.exists(op.position)
    return READ_ONLY

  omap.set(op.position, op.entry_data)

  maxpos = omap.get("max.pos")
  if op.position > maxpos
    omap.set("max.pos", op.position)

  return OK
```

Listing 2: Object interface for writing to a log position.

As shown in Listing 3 reading from the log is very similar. First the epoch of the request is checked. If the entry at the requested position does not exist a not-written error is returned. This state is very important to clients that require strict ordering such as state machine replication, which cannot process later entries. Finally the entry is read and the entry data is returned to the client. Note that the invalidated state is used to by the client protocol to mark positions as junk if writers are too slow.

Other interfaces exist which we don't show here. For instance, the seal command updates the stored epoch value and returns the stored maximum position. In Section 6 we discuss how interfaces and metadata are consistently versioned and dynamically deployed and installed using existing services found in Ceph.

5. Virtual Object Persistence

One of the more tricky challenges in mapping the components of CORFU onto an existing storage system is finding a mapping target for the sequencer service. As mentioned in Section 3.3, one option is to use the object interface extensibility feature in RADOS to construct an object interface

```
def read(op)
  ret = check_epoch(op.epoch)
  if ret != OK
    return ret

  if !omap.exists(op.position)
    return NOT_WRITTEN

  entry = omap.read(op.position)
  if entry.invalidated
    return INVALIDATED

  return entry.data
```

Listing 3: Object interface for reading a log entry.

that implements the sequencer protocol. This would be extremely convenient as new objects of a sequencer type can be created at will. Listing 4 shows a candidate implementation of the sequencer as a custom object type which reads the tail value, optionally increments, and then returns the value. In addition to on demand sequencer service, it would provide automatic service fail-over via existing object fault-tolerance mechanisms as depicted in Figure 4. Unfortunately, existing RADOS semantics force all object state to be stored durably, a property that introduces unnecessary overhead for the sequencer. Next we describe programmable persistence that allows us to use a cache for object state, and then describe orthogonal policies that can be used to optimize performance for cached data access.

```
def tail(op)
  tail = omap.get("tail")
  if op.increment
    tail++
    omap.set("tail", tail)
  return tail
```

Listing 4: Example implementation of sequencer as a custom object type.

5.1 Programmable Persistence

In order to handle fail-over for non-durable object state, we make the following observation about CORFU that guides us to a solution. During sequencer recovery the log is inspected to determine the correct value with which to repopulate the cached tail position. Where replication and erasure coding have very clear recovery mechanisms following failure (copy and reconstruction, respectively), recovery can also be expressed as a domain-specific procedure. In the case of the CORFU sequencer, the cached value is implicitly contained in the log. Thus, recovery of the sequencer cache

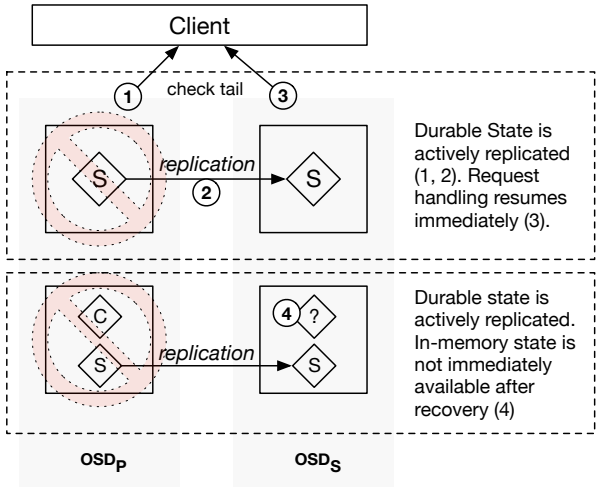

Figure 4. Availability of data following fail-over. Durable state is already handled through replication or erasure coding (top). Cached state is not replicated and must be computed (4).

value can be expressed as a function of the log that is stored using traditional durability mechanisms.

In contrast with recovery using replication shown at the top of Figure 4 in which the secondary OSD has immediate access to the replicated value following fail-over recovery, programmable persistence interposes on the recovery process to install domain-specific actions to restore non-replicated state. The bottom of Figure 4 illustrates the concept. As a secondary recovers durable state has been replicated, but cached state is lost (label 4). Normally the OSD resumes operation once durable state is consistent. By interposing on the basic state machine we delay this transition until the domain-specific recovery process that repopulates the cache has completed (e.g. reading the log from the system to determine the maximum log position and installing a new epoch value).

5.2 Cache Optimization

The final component of synthesizing the sequencer service within Ceph is to address the optimization that places the cached log tail in volatile memory and optimizes request performance. The bottom of Figure 3 depicts an LRU cache that maintains common object information (e.g. size, modification time). When an operation such as a *stat* is able to read from this cache, requests are satisfied without I/O. However, in order to maintain strong consistency, LRU cache hits are still subjected to nearly the entire request processing pipeline, and that can add a lot of unnecessary latency.

Rather than utilize this strongly consistent cache, we instead propose the addition of a new cache, shown at the top of Figure 3 that can be accessed at an optimal stage during request processing, given the durability and consistency requirements of the object state being accessed.

In Section 7 we evaluate the performance improvements that can be achieved using such a technique. We don't propose yet any specific method for specifying the consistency and durability properties of object state, though our main focus has been on declarative methods that separate configuration from correctness.

6. Interface and Metadata Management

Enabling programmability in storage systems implies that the extensibility features must be dynamic. Practically speaking this makes sense because in multi-tenant environments, or any system in which applications frequently modify system interfaces and services, restarts that are typically required by traditional extensibility methods are too costly.

Throughout this paper we have shown how various components of a storage system can be modified slightly to expose an entirely new service. All of these modifications work in tandem to implement the desired behavior, and are typically co-designed together such that each depends on the other to behave as expected. In highly dynamic environments such as a distributed storage system, changes to the system propagate at variable rates, requiring all components of the system to be able to adapt. Thus, system services realized through extensibility should be versioned together as a group, exposing the version to allow introspection and domain-specific treatment of version mismatch scenarios.

And finally, it is important to not underestimate the importance of the preservation of the modifications that enable a new system service. In many (if not all) cases, interfaces defining access to data are just as important as the data itself by virtue of inherently providing structural context. Thus, all components of a system service must be kept to the same standards of protection as data in the system itself.

The Ceph storage system has a large collection of metadata management and cluster monitoring services that are easily used to fulfill the requirements outlined above. For instance, metadata describing the set of nodes in the system is managed via a Paxos cluster, and automatically distributed to clients and servers through a scalable gossip protocol, and additionally the Ceph file system has a capabilities sub-system used to control client caching behavior.

Figure 5 illustrates the use of Ceph's cluster management service. All clients and servers eventually converge to the latest version of metadata that has been proposed, but intermediate states exist in which clients and servers may have mismatched versions of system modifications. The solution we currently provide is to expose the current version to the execution environment via the extensibility API. This allows scripts to be written that introspect their environment and thus policies for handling mismatch can be handled on a case-by-case basis. We envision that later work in formalizing an extensibility language will reveal a fixed set of scenarios that can be codified into configuration mechanisms.

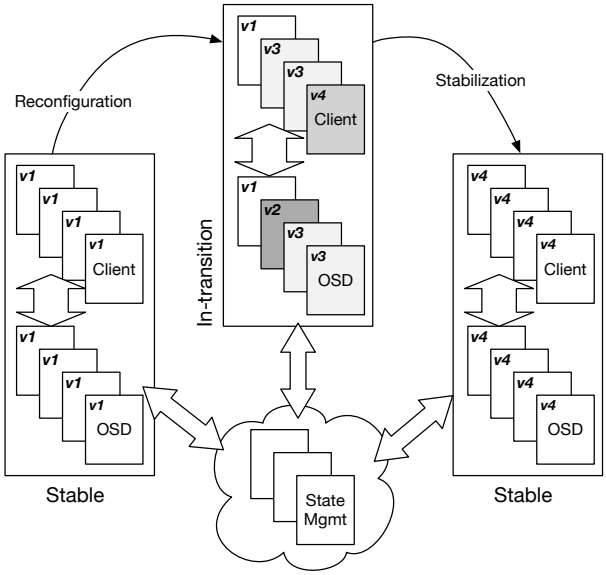

Figure 5. Distribution of system modifications converge on the latest version but must be robust to version mismatch.

Note that we assume in this example that a service may be used by any client, and data may be stored on any server. Thus, the global distribution and versioning of metadata by the monitoring service is sufficient. However, in a system where provisioning is important, it may be worthwhile to add a partitioning feature to the management of interface data to match the provisioning policies at the cluster management level.

7. Evaluation

We perform our evaluation on a slice of the Wisconsin CloudLab cluster. Each node in the cluster contains a 16 core Haswell processor, 128 GB of RAM, 2x 1.2 TB SAS HDD and 1x 480 GB SAS SSD. While sporting dual 10Gb networking ports, as of writing the system is in an alpha release with network support limited to a 1Gb control network.

7.1 Programmable Persistence

As discussed in Section 5 the ability to achieve high-performance for a CORFU sequencer depends on an optimization that stores server state in volatile memory. The important components to realizing this in Ceph are a programmatic way to describe the recovery of cached data, and optimizations that take advantage of serving cached data. The first experiment we highlight is the performance of a sequencer-like workload interacting with cached state in the OSD, and observing the ability of the system to recover from failures. We use an 8 OSD RADOS cluster and configure the system to make four replicas of an object representing the sequencer service. Figure 6 shows the number of IOPS over time experienced by clients interacting with the sequencer before and after three failure events. The OSD with rela-

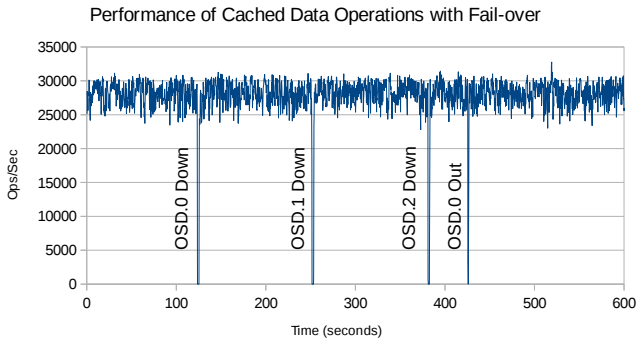

Figure 6. Request rate to cached data before, during, and after fail-over. The OSDs are failed by shutting down the server daemons.

tively little tuning is able to achieve between 25K and 30K IOPS. After each OSD failure the client experiences a short period where the service in unavailable, after which point Ceph automatically migrates the service and the client resumes. The final drop in performance doesn't correspond to a failed OSD, but appears to correspond to the 300 second delay Ceph uses before marking an OSD that is down as being out of the cluster.

In this experiment failure is simulated by forcing the current primary OSD to shutdown. However, there are other ways in which OSDs can become unavailable, such as a network partition, an administrative action, or hardware or software failures. A classic challenge is tuning a system to respond to failure without costly thrashing due to transient problems. Figure 7 shows the same experiment as Figure 6 where the method of removing the OSD from the system is an administrative action. Notice the degradation of performance after each failure; this is not due to a lack of I/O parallelism as it may appear at first. Rather, Ceph is in a mode where data is temporarily remapped, likely in anticipation of a quick resolution. Removing the mappings by activating the failed OSDs causes performance to resume.

This raises an important point regarding system configuration and reuse of system components. Services such as the CORFU sequencer implemented as an object in RADOS are more likely to benefit not only from fast failover, but also fast detection of failure. This stands in stark contrast to the policies that are currently used in RADOS where larger grace periods are used to account for transient outages because initiating failure is very expensive for large storage arrays. Thus, extensibility features in a storage system will require careful exposure of fail-over policies as well, at a granularity fine enough to allow such distinctly different behaviors to co-exist.

While the performance of the cache optimization allows the sequencer to achieve roughly 25K IOPS, this is an order of magnitude slower than what was achieved in the original CORFU paper. We note that a significant execution cost associated with maintaining consistency and transaction pro-

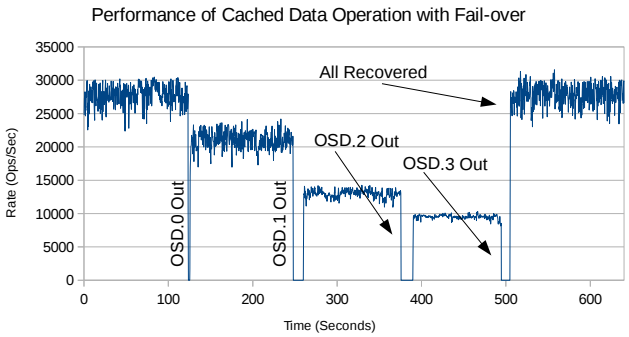

Figure 7. Request rate to cached data before, during, and after fail-over. The OSDs are failed using an administrative function that removes them from operation.

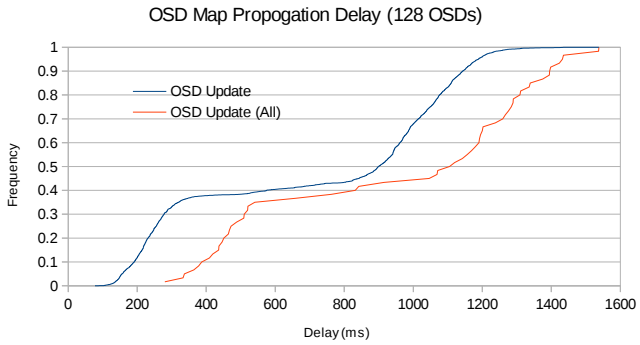

Figure 8. Object server request rate before, during, and after fail-over.

cessing in the OSD is represented in the performance of our implementation. With relaxed consistency policies on object state we can move cached data access closer to the network device avoiding the overhead of this consistency maintenance.

We omit the evaluation of the actual cost of recovering the sequencer cache as this can largely be considered a constant added onto the baseline fail-over costs introduced by Ceph. Sequencer fail-over in our version of the system is a single read to a fixed number of objects.

7.2 Interface and Configuration Propagation Delay

In Section 6 we discussed the challenges related to versioning and distribution of interface configuration and metadata. We simulate this in Ceph by measuring the cost of propagating changes to the OSD map which is a strongly consistent data structure used to direct I/O operations in the cluster to the correct servers. Changes to the OSD map are automatically propagated to clients and servers, making them an excellent proxy to evaluate the performance method for interface management.

We conduct a set of experiments as follows. Using a cluster of 128 OSDs we trigger the construction of a new version of the OSD map and insert a unique ID into the map. On each daemon in the system we log a message when the new map is received and note the unique ID contained in it. We repeat the experiment 60 times with a 5 second delay and record the results. Using this data we generated two curves shown in Figure 8.

The lower curve, labeled *OSD Update (All)*, measures the cost of for all daemons to receive a specific update. That is, all daemons have converged on a single copy of the OSD map. For instance, 50% of all updates were fully propagated in the system in less than 1.1 seconds. For some interfaces this is an important metric because they may not tolerate a mixed set of versions.

Other applications, however, may tolerate mixed versions using domain-specific semantics. For this case we look at the upper curve, labeled *OSD Update*. This curve measures the cost for any OSD to receive an update. As an example, for any system daemon, there is a 90% likelihood that that daemon will receive an update in less than 1.2 seconds.

8. Conclusion and Future Work

Programmable storage is a viable method for eliminating duplication of complex error prone software that are used as workarounds for storage system deficiencies. However, this duplication has real-world problems related to reliability. We propose that system expose their services in a safe way allowing application developers to customize system behavior to meet their needs while not sacrificing correctness.

We are intend to pursue this work towards the goal of constructing a set of customization points that allow a wide variety of storage system services to be configured on-the-fly in existing systems. This work is one point along that path in which we have looked an a target special-purpose storage system. Ultimately we want to utilize declarative methods for expressing new services.

References

[1] A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms and evaluations. 1998.

[2] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. D. Davis. Corfu: A shared log design for flash clusters. In *NSDI'12*, San Jose, CA, April 2012.

[3] M. Balakrishnan et al. Tango: Distributed data structures over a shared log. In *SOSP '13*, Farmington, PA, November 3-6 2013.

[4] A. Balmin, T. Kaldewey, and S. Tata. Clydesdale: Structured Data Processing on Hadoop. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 705–708, 2012. ISBN 978-1-4503-1247-9. . URL `http://doi.acm.org/10.1145/2213836.2213938`.

[5] P. A. Bernstein, C. W. Reid, and S. Das. Hyder – a transactional record manager for shared flash. In *CIDR '11*, Asilomar, CA, January 9-12 2011.

[6] P. A. Bernstein, C. W. Reid, M. Wu, and X. Yuan. Optimistic concurrency control by melding trees. In *VLDB '11*, 2011.

[7] P. A. Bernstein, S. Das, B. Ding, and M. Pilman. Optimizing optimistic concurrency control for tree-structured, log-structured databases. In *SIGMOD '15*, 2015.

[8] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. *Proceedings of the VLDB Endowment*, 3(1-2), 2010.

[9] M. Carlson, A. Yoder, L. Schoeb, D. Deel, and C. Pratt. Software defined storage. Working draft, SNIA, April 2014.

[10] J. Ekanayake, S. Pallickara, and G. Fox. MapReduce for Data Intensive Scientific Analyses. In *Proceedings of the 4th IEEE International Conference on eScience*, ESCIENCE'08, 2008.

[11] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: A Runtime for Iterative MapReduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC'10, 2010.

[12] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[13] M. Mihailescu, G. Soundararajan, and C. Amza. MixApart: Decoupled Analytics for Shared Storage Systems. In *Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'12, 2012.

[14] E. Riedel, G. A. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia. In *24th international Conference on Very Large Databases (VLDB '98)*, New York, NY, 1998.

[15] K. o. V. Shvachko. HDFS Scalability: The Limits to Growth. *login; The Magazine of USENIX*, 2010.

[16] 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 *OSDI'06*, Seattle, WA, Nov. 2006.

[17] S. A. Weil, A. Leung, S. A. Brandt, and C. Maltzahn. Rados: A fast, scalable, and reliable storage service for petabyte-scale storage clusters. Reno, NV, November 2007.