outline!

Submission Type: Research

Abstract

To meet the needs of a diverse and growing set of cloud-based applications, modern distributed storage frameworks expose a variety of composable subsystems as building blocks. This approach gives infrastructure programmers significant flexibility in implementing application-specific semantics while reusing trusted components. Unfortunately, in current storage systems the composition of subsystems is a low-level task that couples (and hence obscures) a variety of orthogonal concerns, including functional correctness and performance. Building an application by wiring together a collection of components typically requires thousands of lines of carefully-written C++ code, an effort that must be repeated whenever device or subsystem characteristics change.

In this paper, we propose a declarative approach to subservice composition that allows programmers to focus on the high-level functional properties that are required by applications. Choosing an implementation that is consistent with the declarative functional specification then can be posed as a search problem over the space of parameters such as block sizes, storage interfaces (e.g. key/value or block storage) and concurrency control mechanisms. We present experimental evaluation of our prototype, (etc etc)

1 Introduction

Storage systems are increasingly providing features that take advantage of application-specific knowledge to achieve optimizations and provide unique services. However, this trend is leading to the creation of a large number of software extensions that will be difficult to maintain as system software and hardware continue to evolve.

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. The widely

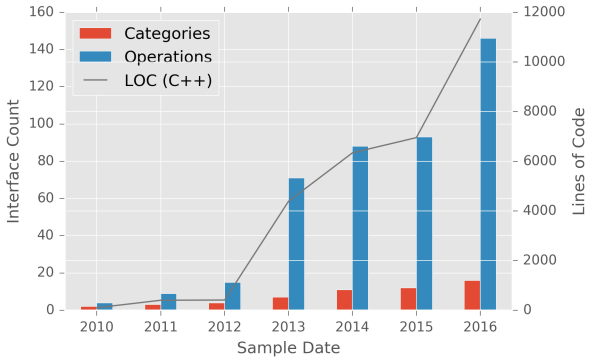
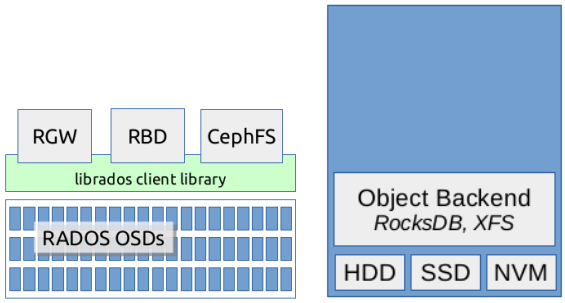


Figure 1: [source] Growth of officially supported, custom object interfaces in RADOS over 6 years. An *operation* is a function executed in the context of an object, and operations are grouped into different *categories* corresponding to applications or utilities, such as reference counting

deployed Ceph distributed storage system is an example of a storage system that supports application-specific extensions in the form of custom I/O interfaces to objects managed by the underlying RADOS object storage system [10, 11]. Organizations are increasingly reliant upon these extensions as is shown in Figure 1 by a marked increase in the number of object operations that are packaged as part of the Ceph distribution and widely used by internal Ceph subsystems and by applications such as OpenStack Swift and Cinder [2].

In addition to the growth in the quantity of operations in use throughout Ceph installations, Figure 1 also depicts the amount of low-level C++ written to implement these operations. Unfortunately, this code is written assuming a performance profile defined by the combination of the hardware and software versions available at the time of development. While the bulk of these interfaces are created by core Ceph developers with a complete view of the performance model, this may be changing as the development community has been receptive to outside contributions with the recent inclusion by CERN developers of an extension for performing limited numeric operations on object data [1]. And while Ceph has not yet reached the point of directly exposing these features to non-administrative users, the recent inclusion of a mechanism for dynamically defining extensions using Lua [3] suggests that aspects of this feature may soon appear. What is needed is support for creating storage interfaces



(a) Ceph **(b)** OSD

Figure 2: Ceph and OSD... gotta cram more stuff in here...

using a method that allows transparent optimization as the system, application, and supporting environment evolve.

Previous work related to storage interface design has largely been in the context of standardization efforts and active storage. While the later has been specifically concerned with the creation of application-specific interfaces and demonstrating short-term benefits, the standardization seeks to develop a last set of interfaces. In neither case have we seen efforts focused on extensible interfaces with portability and longevity a primary concern.

We propose the introduction of a declarative language for building object based interfaces that allows a storage system to meet the needs of an application throughout the development process without requiring rewrites.

2 Background

In this section we highlight the salient components of Ceph, especially its *object class* feature that offers users the ability to load and execute application-specific codes. We provide a description of our motivating example, a high-performance distributed shared-log built upon Ceph that makes extensive use of the object class facility, and then we present the challenges that application developers face when using this extensibility feature offered by the storage system.

2.1 Ceph and Storage Programmability

Figure 2a illustrates the collection of components commonly referred to as Ceph. At the bottom, a cluster of 10s–10,000s *object storage devices* compose the distributed object storage system called RADOS. Widely deployed applications such as the S3/Swift–compliant RADOS Gateway (RGW), RADOS Block Device (RBD), and the POSIX Ceph File System are built upon the *librados* client layer that presents a fault-tolerant always-on view of the RADOS cluster.

The object storage device (OSD), illustrated in Figure 2b, is the building block of the RADOS cluster and is responsible for managing and providing access to a set of named objects. The configuration of an OSD is flexible, and commonly contains a mix of commodity hardware such as HDD and SSD bulk storage, a multi-core CPU, GBs of RAM, and one or two 10 Gb Ethernet links. Clients access object data managed by an OSD by invoking native object operations exposed by the OSD such as reading or writing bytes, as well as more complex operations like taking snapshots or composing one or more native operations into compound procedures that execute in a transactional context.

The native object operations in RADOS roughly fall into two categories based on the type of data being accessed: key-value items, or bulk bytestream data. The key-value interface operates as a dedicated database associated with each object, and the bytestream interface supports random byte-level access similar to a file. At a low-level each of these abstract I/O interfaces map to hardware storage devices through a pluggable object backend storage service. For instance, LevelDB or RocksDB may be used to store key-value data, while the *FileStore* implementation maps the bytestream interface onto a local POSIX file system [8, 7]. Several backend implementations exist for storing data in targets such as the Kinetic Drive or in NVMe devices, among others.

Object Classes While Ceph provides a wide variety of native object operations, it also includes a facility referred to as *object classes* that allow developers to create application-specific object operations in the form of C++ shared libraries dynamically loaded into the OSD process at runtime. Object classes can be used to implement basic data management tasks such as indexing metadata, or used to perform complex operations such as data transformations or filtering. Table 1 summarizes the range of object classes maintained in the upstream Ceph project which support internal Ceph subsystems as well as applications and services that run on top of Ceph.

A critical step in the development of application-specific object interfaces is deciding how to best make use of the native object interfaces. For instance if an application stores an image in an object, it may also extract and store EXIF metadata as key-value pairs in the object key-value database. However, depending on the application needs it may be sufficient or offer a performance advantage to store this metadata as a header within the bytestream. In the remainder of this section we will explore the challenges associated with these design questions.

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

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

2.2 Motivating Application: CORFU

The primary motivating example we will use in this paper is the CORFU distributed shared-log designed to provide high-performance serialization across a set of flash storage devices [5]. The shared-log is a powerful abstraction useful when building distributed systems and applications, but common implementations such as Paxos or Raft funnel I/O through a single node limiting total throughput [9]. The CORFU protocol addresses this limitation by de-coupling log entry storage from log metadata management, making use of a centralized, volatile, in-memory *sequencer service* that assigns positions to clients that are appending to the log. Since the sequencer is centralized serialization is trivial, and the use of non-durable state allows the sequencer service to operate at very high rates. The CORFU system has been used to demonstrate a number of interesting services such as transactional key-value and metadata services, replicated state machines, and an elastic cloud-based database management system [4, 6].

Two aspects of CORFU make its design attractive in the context of the Ceph storage system. First, CORFU assumes a cluster of 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 existing and future data management features such as tiering, allowing users to freely choose between media types such as SSD, spinning disks for archival storage, or emerging NVRAM technologies.

CORFU and Storage Programmability The second property of CORFU relevant in the context of Ceph is the dependency CORFU places on custom storage device interfaces used to guarantee serialization during failure and reconfiguration. Each flash device in a CORFU cluster exposes a 64-bit write-once address space consisting of the primary I/O interfaces *write(pos, data)* and *read(pos)* for accessing log entries, as well as *fill(pos)* and *trim(pos)* that invalidate and reclaim log entries, respectively. All I/O operations in CORFU initiated by clients are tagged with an *epoch* value, and flash devices are expected to reject client requests that contain an old epoch value. To facilitate recovery or handle system reconfiguration in CORFU, the storage devices are also required to support a *seal(epoch)* command that stores the latest epoch and returns the maximum position written to that device. The seal interface is used following the failure of a sequencer to calculate the tail of the log that the sequencer should use to repopulate its in-memory state.

While the authors of the CORFU paper describe prototype device interfaces implemented as both host-based and FPGA-based solutions, RADOS *directly* supports the creation of logical storage devices through its object class feature described previously in Section 2.1. Thus, by using software-based object interfaces offered by RADOS flash devices in CORFU can be replaced by software-defined storage offering significant flexibility and a simplified design.

The implementation of a custom object class that satisfies the needs of an application such as CORFU is often straightforward. However, as described in Section 2.1 there are a variety of native object I/O interfaces available, and it is not always immediately clear how best to utilize these interfaces.

Towards a CORFU Object Interface The state-machine diagram in Figure 3 shows the composition of actions for each component of the CORFU interface. For instance, all operations begin by applying an *epoch guard* that ensures the request is tagged with an up-to-date epoch value. The *read* (R) and *write* (W) operations both proceed by (1) examining metadata associated with the target position, (2) performing I/O to read or write the log entry, and in the case of a write, (3) updates metadata for the target log position.

The primary concern of an application developer when implementing an object interface in Ceph is deciding how native interfaces are composed into a compound operation. These types of decisions are commonly referred to as physical design, and can affect performance and application flexibility. For instance, one valid design option is to store each log position in an object with a name using a one-to-one mapping with the log entry position. This would simplify the design of the *write* interface because a small amount of metadata stored as a header in the object could describe the state of the log entry. However, as we will see in the next section this choice of a physical design can result in poor performance compared to other designs.

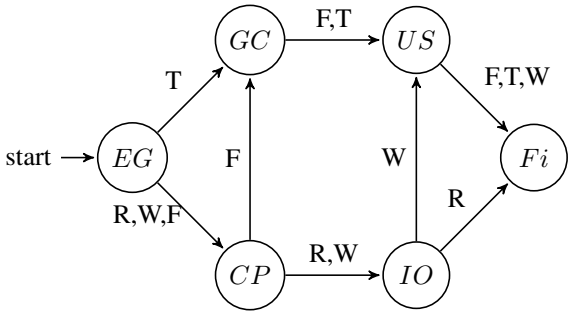


Figure 3: State transition diagram for read (R), write (W), fill (F), and trim (T) CORFU operations. The states epoch guard (EG), check position (CP), and update state (US) access metadata. The I/O performs a log entry read or write, and garbage collection (GC) marks entries for reclamation.

2.3 Physical Design

As we have seen, Ceph provides a rich storage API and places few restrictions on the structure of applications. So how should we go about implementing CORFU on Ceph? Our strategy is straightforward: first we will define the entire design space, and then winnow down this space using a set of targeted benchmarks. The design space can be divided into three challenges: selecting a strategy for log entry addressing, choosing a native I/O interface for storing log entry content, and implementing efficient metadata management.

1. **Entry addressing.** We refer to the method by which a client locates a log entry in Ceph as entry addressing, and we consider two strategies. In a one-to-one (1:1) strategy each log entry is stored in a distinct object with a name derived from the associated log entry position. This is an attractive option because it is trivial for clients to locate a log entry given its position. In contrast, an N:1 strategy *stripes* log entires across a smaller set of objects, but this adds complexity to both the client and the object interface which must multiplex a set of entries.

2. **Log entry storage.** Clients read and write binary data associated with each log entry, and these entries can be stored in the bytestream or in the key-value database associated with an object. Retrieval of log entry payloads should perform well for both large (e.g. database checkpoint) and small log entries.

3. **Metadata management.** The CORFU protocol defines the storage interface semantics, such as enforcing up-to-date epoch values and a write-once address space. The object interface constructed in Ceph must implement these semantics in software by storing metadata (e.g. the current epoch) and validating requests against this metadata (e.g. has the target position been written?). A key-value store is a natural location for this type of data, but metadata management adds overhead to each request and must be carefully designed.

In the remainder of this section we will explore the full design space defined by the cross product of these design challenges to arrive at a final design.

Baseline Entry Storage Performance We begin the process of exploring the design space by exploiting the fact that metadata management is a complexity and performance overhead that a full implementation must incur beyond the costs of storing log entry data. Therefore we first explore the design space by restricting the space to log entry addressing and log entry storage (the *I/O* action shown in Figure 3). These two dimensions are represented by the first two columns of Table 2 which describes the entire design space. In the I/O column *KV* corresponds to the key-value interface, and the bytestream interface is represented by *AP* for an append strategy, and *EX* for a strategy that writes to an explicit bytestream offset (both described shortly).

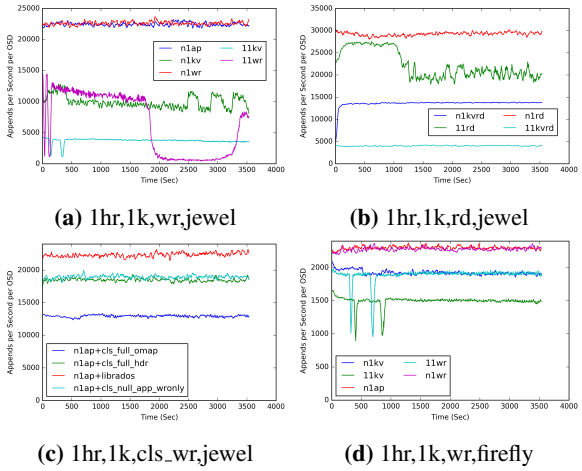
Map	I/O	Entry Size	Addressing	Metadata
1:1	KV	Flex	Ceph	KV/BS
	AP	Flex	Ceph/VFS	KV/BS
N:1	KV	Flex	KV/BS	
	EX	Fixed	VFS	KV/BS
	AP	Flex	KV/BS	KV/BS

Table 2: The high-level design space of mapping CORFU log entry storage onto the RADOS object storage system.

Figure 4a shows the expected performance of 1K log appends without metadata management overhead using each of the five strategies defined in Table 2. The first thing to notice in this figure is that both one-to-one strategies have relatively poor performance. The large period of reduced throughput for *11wr* corresponds to the OSD splitting file system directories in order to maintain a maximum directory size, and will occur in *11kv* although the threshold number of objects is not reached in this example due to reduced overall throughput of the *11kv* strategy.

The second lesson that we can learn from Figure 4a is that even when using an N:1 addressing strategy, the key-value interface imposes a large overhead. This is unfortunate because the key-value interface can provide a direct solution to addressing log entries within an object. Instead, what we find is that an N:1 addressing strategy that stores log entries in the bytestream using either object appends or writes to explicit offsets outperform all other strategies by a factor of over 2x.

The apparent performance tie between the strategies of appending to an object and writing to explicit offsets can



(a) 1hr,1k,wr,jewel **(b)** 1hr,1k,rd,jewel

(c) 1hr,1k,cls_wr,jewel **(d)** 1hr,1k,wr,firefly

Figure 4: placeholder.

be broken by considering the flexibility offered by each approach. The third column *entry size* in Table 2 shows if a particular strategy supports storage of entries with dynamic sizes, or if entries must be restricted to a fixed size. Notably the strategy that stores log entries at explicit object offsets is limited in this regard because each log entry is effectively pre-mapped into the storage system. This leaves the clear winner: an N:1 strategy that appends log entries to objects provides flexibility and the best write performance in this particular configuration. This result is corroborated by considering the read performance for each strategy as well. Figure 4b shows the expected random read performance from a log containing 1K entries in which an addressing strategy that stores log entries in the bytestream has the best performance.

Metadata Management The previous results show that storing log entries in the bytestream has the potential to provide the best overall performance, but by design, those results do not contain the real-world overheads introduced by metadata management such as validating that requests are tagged with an up-to-date epoch value. Having only focused on entry I/O costs, in this section we consider the overhead of the remaining actions shown in Figure 3.

1. **Epoch guard.** Each client request must be validated against the current epoch. This singleton value in infrequently updated, but the cost of accessing the value must be incurred for every request.

2. **Per-entry metadata.** Unlike the singleton epoch value, metadata associated with each log position must be read, and optionally updated for every request. For instance the *fill* operation must ensure that it is not applied to a log position that has already been written. When an operation that changes a log position is successful (i.e. *write*, *fill*, *trim*) the per-entry metadata must also be updated to reflect the change. While the size of the per-entry metadata is small, it is accessed for every operation and a single object may manage a large number of entries.

The design space for managing metadata can be large depending on the application. In many cases this space is significantly reduced when the key-value interface is used because it handles a large portion of common data management challenges such as indexing. On the other hand, we have seen that the bytestream can be used to achieve much higher performance when used in lieu of the key-value database, but leaves the design space wide open.

We consider two high-level design prototypes in this paper based on both the key-value and bytestream interfaces. The summary of performance of these designs is shown in Figure 4c where we begin by highlighting two baseline throughputs labeled *librados* and *cls_null* that correspond to the baseline append throughput described in the previous section, and the append throughput achieved with *object class* overhead included. We will discuss the performance of our two prototype designs relative to the base cost of using *object class* facility.

The first design is based on the key-value interface. In this design the epoch value is stored under a fixed key, and the metadata associated with each log entry is stored under a key derived from the log position. The expected throughput of this design is shown in Figure 4c and labeled as *cls_full_omap*. This result shows us that even for very small values the cost of managing data in the key-value store introduces significant overhead. In contrast, the unstructured bytestream interface imposes little to no restriction on how it is used.

An obvious choice for encoding the singleton epoch value in the bytestream is to position it at a fixed offset such as in a header. A method for indexing entry metadata is far less obvious. Solutions span a wide design spectrum and include approaches such as encoding a search tree into the bytestream, or exploiting the regularity log addressing and storing a dense array that can be efficiently indexed. While there is clearly a wide variety of approaches, we have chosen to examine the overhead incurred by an hypothetical best-case design in which per-entry metadata can be accessed with a single I/O. The I/O overhead of the test includes (1) read epoch value from a fixed header location, (2) read the per-entry metadata, and in the case of an update (3) write the updated per-entry metadata. The expected throughput from such a design is shown in Figure 4c labeled as *cls_full_hdr* and performs with roughly a 6% overhead.

Hardware and Software In this section we have demonstrated that a design based on an N:1 addressing

scheme that stores both log entries and metadata in the bytestream can provide the best overall performance, and it does so by a large margin. However, this process of design we have outlined may not yield such clear cut results when applied in other contexts that may differ by network and hardware capabilities as well as configuration options and software versions.

To illustrate this we ran the same benchmark that measures expected append throughput shown in Figure 4a on identical hardware but with a version of long-term support version of Ceph released two years ago. The result of this benchmark is shown in Figure 4d in which the bytestream interface outperforms the key-value interface by 12%, compared to over 100% in the newer versino of Ceph. Qualatatively, given the reduced overall throughput achieved in the older version, some developers may find that incurring the overhead of using the key-value interface is an easy decision given the reduced complexity of the design space for metadata management.

3 Programming Model

In this section we are going to be describing our way of creating interfaces.

4 Other Interfaces

Here we show the derivation of two other interfaces that are in production in Ceph today to demonstrate the generality of our interface.

5 Evaluation

6 Related Work

7 Conclusion

References

[1] Merged pull request for cls_numops. https://github.com/ceph/ceph/pull/4869. Accessed: 2016-05-12.

[2] OpenStack Open Source Cloud Computing Software. http://www.openstack.org. Accessed: 2016-05-12.

[3] Pull request for cls_lua. https://github.com/ceph/ceph/pull/7338. Accessed: 2016-05-12.

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

[5] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. Corfu: A shared log design for flash clusters. In *NSDI'12*, San Jose, CA, April 2012.

[6] Philip A. Bernstein, Colin W. Reid, and Sudipto Das. Hyder – a transactional record manager for shared flash. In *CIDR '11*, Asilomar, CA, January 9-12 2011.

[7] Facebook. RocksDB. http://rocksdb.org, 2013.

[8] Sanjay Ghemawat and Jeff Dean. LevelDB. http://code.google.com/p/leveldb, 2011.

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

[10] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *OSDI'06*, Seattle, WA, November 2006.

[11] Sage A. Weil, Andrew Leung, Scott A. Brandt, and Carlos Maltzahn. Rados: A fast, scalable, and reliable storage service for petabyte-scale storage clusters. Reno, NV, November 2007.