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 widely 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 [8, 9]. 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 subystems and by applications such as OpenStack Swift and Cinder [1]. In addition to the growth in the quantity of operations, 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. As Ceph continues to evolve at a fast pace, the cost of adapting application-specific codes to changing assumptions regarding performance may become significant.

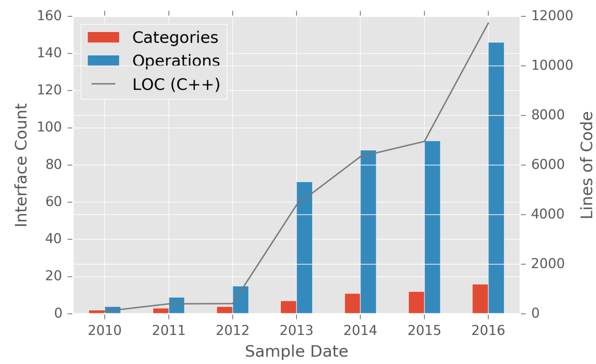


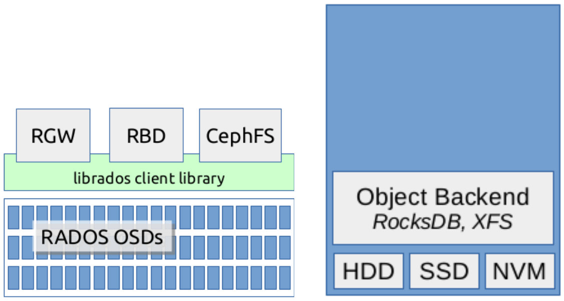
Figure 1: [source] Growth of officially supported, custom object interfaces in RADOS over 6 years. An *operation* is a function executed in the context of an object, and operations are grouped into different *categories* corresponding to applications or utilities, such as reference counting

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 de-



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

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

ployed 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 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. 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 [6, 5]. Several backend implementations exist for storing data in targets such as a 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.

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.

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.

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 [3]. 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 the throughput of log operations [7]. The CORFU protocol addresses this limitation by de-coupling log entry storage from log metadata management, making use of a centeralized, volatile, in-memory *sequencer service* that assigns log positions to clients that are appending to the log. Since the sequencer is centeralized 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 [2, 4].

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 in RADOS, allowing users to freely choose between media types such as SSD, spinning disks, or emerging NVRAM technologies.

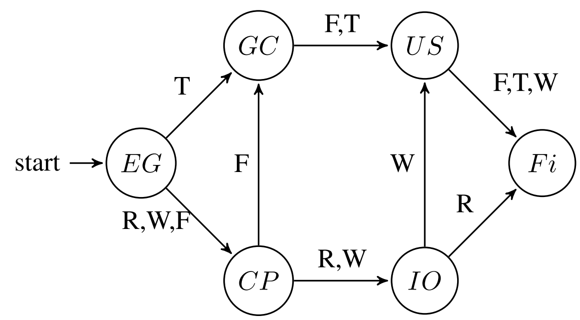


Figure 3: State transition diagram for read (R), write (W), fill (F), and trim (T) CORFU operaitons. The states epoch guard (EG), check position (CP), and update state (US) access metadata. The I/O performs a log entry read or write, and garbage collection (GC) marks entries for reclamation.

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 contian 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 marks 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 using both host-based and FPGA-based implementations, 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 satisifies 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. A common approach is to first define the set of requirements for the desired interface, construct a design space parameterized on the set of native interfaces, and winnow this space down through a process of elimination.

Towards a CORFU Object Interface The state-machine shown 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 flexibilty. 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. However, as we will see in the next section this choice of a physical design can result in poor performance compared to other designs.

2.3 Physical Design

We categorize the challenges of physical design into log entry management, or metadata management.

Our strategy is simply to construct the design space, and winnow it down using a set of targetted experiments.

We'll start by forming a baseline performance profile without any protocol overhead.

All designs store a log entry. We can benchmark the system without any overhead associated with indexing or metadata management.

Baseline Performance Figure **??** illustrates the two primary dimensions of the physical design space. The first dimension concerns the strategy by which a log entry is addressed within RADOS. In a one-to-one strategy each

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.

log entry is stored in a distinct object with a name based on the log entry position. This is an attractive option because it is trivial for a client to address any log entry, and an object interface does not require any complexity associated with multiplexing multiple entries—addressing relies on native indexing provided by the OSDs themselves. In contrast to a 1:1 strategy, an N:1 strategy *stripes* log entries across a smaller set of objects, introducing added complexity into both the client and object interface implementations. The second design dimension selects the primary storage interface for log entries, namely the key-value or bytestream interfaces, described in Section 2.1.

These two dimensions are represented by the first two columns of Table 2 which describes the entire design space, where *KV* corresponds to the key-value interface, and the bytestream interface is represented by *AP* and *EX* for an append or explicit i/o straetgy.

. Note that we decompose the bytestream I/O interface into two sub-strategies: I/O using explicit offsets (EX), and appends (AP).

CORFU device interface requirements in Ceph as an object interface. So far we have

append is more attractive, all things being equal, as it supports flexible mapping size.

There is a finite set of strategies for mapping the CORFU interfaces onto the current set of native I/O interfaces used to construct object interfaces. Table 2 shows the design space for mapping the CORFU interfaces onto Ceph. In order to select the best mapping we performed a parameter sweet over the design space.

Figure 4a shows the result of a parameter sweep which clearly shows that an N-1 mapping based on the bytestream interface provides superiour performance. However, the other two graphs show the same interfaces running on an old version of Ceph that show the same decision would not have been the optimal choice ***note: for the outline these are not the real graphs we'll be using***.

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

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

[3] 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.

[4] 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.

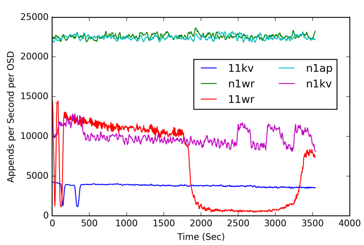
[5] Facebook. RocksDB. http://rocksdb.org, 2013.

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

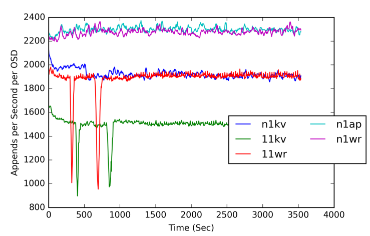
[7] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[8] 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.

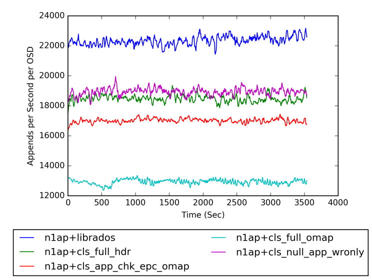
[9] Sage A. Weil, Andrew Leung, Scott A. Brandt, and Carlos Maltzahn. Rados: A fast, scalable, and reliable storage service for petabyte-scale storage clusters. Reno, NV, November 2007.



(a) a



(b) b



(c) c

Figure 4: Shown here are the graphs and such that demonstrate that the same physical design choices are not the same between differing version of Ceph even on the same hardware.