Malacology: A Programmable Storage System

Michael A. Sevilla, Noah Watkins, Ivo Jimenez,
Peter Alvaro, Shel Finkelstein, Jeff LeFevre, Carlos Maltzahn
{msevilla, jayhawk, ivo}@soe.ucsc.edu, {palvaro, shel, jlefevre, carlosm}@ucsc.edu

Abstract

Storage systems are caught between the need to evolve data processing applications efficiently and quickly, and the increasing velocity with which storage device technology evolves. This puts tremendous pressure on storage systems to support rapid change both in terms of their interfaces and their performance. But adapting storage systems can be difficult because unprincipled changes might jeopardize years of code-hardening and performance optimization efforts that were necessary for users to entrust their data to the storage system. We introduce the programmable storage approach, which exposes internal services and abstractions of the storage stack as building blocks for higher-level services. We also build a prototype to explore how existing abstractions of common storage system services can be leveraged to adapt to the needs of new data processing systems and the increasing variety of storage devices. We illustrate the advantages and challenges of this approach by composing existing internal abstractions into two new higher-level services: a file system metadata load balancer and a high-performance distributed shared-log. The evaluation demonstrates that our services inherit desirable qualities of the back-end storage system, including the ability to balance load, efficiently propagate service metadata, recover from failure, and to navigate tradeoffs between latency and throughput using leases.

1. Introduction

A storage system implements abstractions designed to persistently store data and must exhibit a high level of correctness to prevent data loss. Storage systems have evolved around storage devices that often were orders of magnitude slower than CPU and memory, and therefore could dominate overall performance if not used carefully. Over the last few decades members of the storage systems community have developed clever strategies to meet correctness requirements while somewhat hiding the latency of traditional storage media [12]. To avoid lock-in by a particular vendor, users of storage systems have preferred systems with highly standardized APIs and lowest common denominator abstract data types such as blocks of bytes and byte stream files [4].

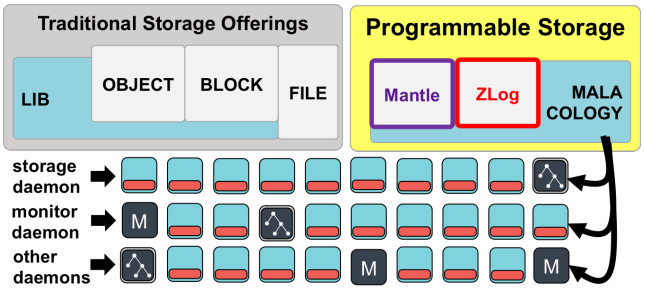

Figure 1: Scalable storage systems have storage daemons which store data, monitor daemons (M) that maintain cluster state, and service-specific daemons (*e.g.*, file system metadata servers). Malacology enables the programmability of internal abstractions (bold arrows) to re-use and compose existing subsystems. With Malacology, we add new services, ZLog and Mantle, that sit alongside traditional user-facing APIs (file, block, object).

A number of recent developments have disrupted traditional storage systems. First, the falling prices of flash storage and the availability of new types of non-volatile memory that are orders of magnitude faster than traditional spinning media are moving overall performance bottlenecks away from storage devices to CPUs and networking, and pressure storage systems to shorten their code paths and incorporate new optimizations [21, 22]. Second, emerging "big data" applications demand interface evolution to support flexible consistency as well as flexible structured data representations. [3]. Finally, production-quality scalable storage systems available as open source software have established and are continuing to establish new, *de-facto* API standards at a faster pace than traditional standards bodies [30, 39].

The evolutionary pressure placed on storage systems by these trends raises the question of whether there are principles that storage systems designers can follow to evolve storage systems efficiently, without jeopardizing years of code-hardening and performance optimization efforts. In this pa-

per we investigate an approach that focuses on identifying and exposing existing storage system resources, services, and abstractions that in a generalized form can be used to *program* new services. By composing higher-level services over existing abstractions one can reuse subsystems and leverage their optimizations, established correctness, robustness, and efficiency.

Contribution 1: We define a programmable storage system to be a storage system that facilitates the re-use and extension of existing storage abstractions provided by the underlying software stack, to enable the creation of new services via composition. A programmable storage system can be realized by exposing existing functionality (such as file system and cluster metadata services and synchronization and monitoring capabilities) as interfaces that can be "glued together" in a variety of ways using a high-level language. Programmable storage differs from *active storage* [34]—the injection and execution of code within a storage system or storage device—in that the former is applicable to any component of the storage system, while the latter focuses at the data access level. Given this contrast, we can say that active storage is an example of how one internal component (the storage layer) is exposed in a programmable storage system.

To illustrate the benefits and challenges of this approach we have designed and evaluated Malacology, a programmable storage system that facilitates the construction of new services by re-purposing existing subsystem abstractions of the storage stack. We build Malacology in Ceph, a popular open source software storage stack. We choose Ceph to demonstrate the concept of programmable storage because it offers a broad spectrum of existing services, including distributed locking and caching services provided by file system metadata servers, durability and object interfaces provided by the backend object store, and propagation of consistent cluster state provided by the monitoring service (see Figure 1). Malacology is expressive enough to provide the functionality necessary for implementing new services.

Malacology includes a non-exhaustive set of interfaces that can be used as building blocks for constructing novel storage abstractions, including:

1. An interface for managing strongly-consistent time-varying **service metadata**.
2. An interface for installing and evolving domain-specific, cluster-wide **data interfaces**.
3. An interface for managing access to **shared resources** using a variety of optimization strategies.
4. An interface for **load balancing** resources across the cluster.
5. An interface for **durability** that persists policies using the underlying storage stack's object store.

Contribution 2: We implement two distributed services using Malacology to demonstrate the feasibility of the programmable storage approach:

1. A high-performance distributed shared log service called ZLog, that is an implementation of CORFU. [7]
2. An implementation of Mantle, the programmable load balancing service [36]

The remainder of this paper is structured as follows. First, we describe and motivate the need for programmable storage by describing current practices in the open source software community. Next we describe Malacology by presenting the subsystems within the underlying storage system that we re-purpose, and briefly describe how those system are used within Malacology (§4). Then we describe the services that we have constructed within the Malacology framework (§5), and evaluate our ideas within our prototype implementation (§6). We conclude by discussing related and future work.

2. Application-Specific Storage Stacks

Building storage stacks from the ground up for a specific purpose results in the best performance. For example, GFS [18] and HDFS [37] were designed specifically to serve Map/Reduce and Hadoop jobs, and use techniques like exposing data locality and relaxing POSIX constraints to achieve application-specific I/O optimizations. Another example is Boxwood [31], which experimented with B-trees and chunk stores as storage abstractions to simplify application building. Alternatively, general-purpose storage stacks are built with the flexibility to serve many applications by providing a variety of interfaces and tunable parameters. Unfortunately, managing competing forces in these systems is difficult and users want more control from the general-purpose storage stacks without going as far as building their storage system from the ground up.

To demonstrate a recent trend towards more application-specific storage systems we examine the state of programmability in Ceph. Something of a storage Swiss army knife, Ceph simultaneously supports file, block, and object interfaces on a single cluster [1]. Ceph's Reliable Autonomous Distributed Object Storage (RADOS) system is a cluster of object storage devices (OSDs) that provide Ceph with data durability and integrity using replication, erasure-coding, and scrubbing [47]. Ceph already provides some degree of programmability; the OSDs support domain-specific object interfaces that are implemented by composing existing low-level storage interfaces that execute atomically. These interfaces are written in C++ and are statically loaded into the system.

The Ceph community provides empirical evidence that developers are already beginning to embrace programmable storage. Figure 2 shows a dramatic growth in the production use of domain-specific interfaces in the Ceph community since 2010. In that figure, classes are functional group-

ings of methods on storage objects (e.g. remotely computing and caching the checksum of an object extent). What is most remarkable is that this trend contradicts the notion that API changes are a burden for users. Rather it appears that gaps in existing interfaces are being addressed through ad-hoc approaches to programmability. In fact, Table 1 categorizes existing interfaces and we clearly see a trend towards reusable services.

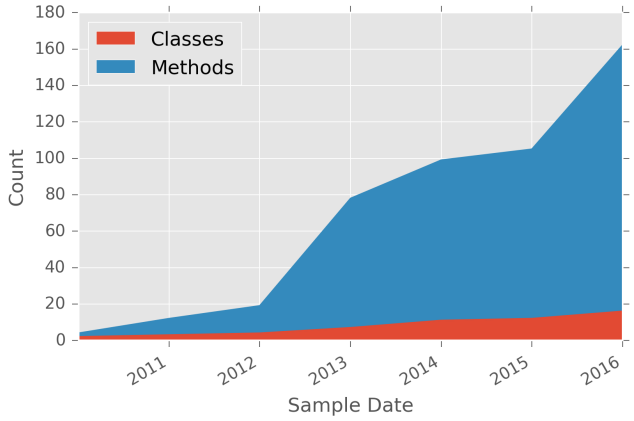

Figure 2: Since 2010, the growth in the number of co-designed object storage interfaces in Ceph has been accelerating. This plot is the number of object classes (a group of interfaces), and the total number of methods (the actual API end-points).

Category	Example	#
Logging	Geographically distribute replicas	11
Metadata Management	Snapshots in the block device OR Scan extents for file system repair	74
Locking	Grants clients exclusive access	6
Other	Garbage collection, reference counting	4

Table 1: A variety of object storage classes exist to expose interfaces to applications. # is the number of methods that implement these categories.

The takeaway from Figure 2 is that programmers are already trying to use programmability because their needs, whether they be related to performance, availability, consistency, convenience, etc., are not satisfied by the existing default set of interfaces. The popularity of the custom object interface facility of Ceph could be due to a number of reasons, such as the default algorithms/tunables of the storage system being insufficient for the application's performance goals, programmers wanting to exploit application-specific semantics, and/or programmers knowing how to manage resources to improve performance. A solution based on application-specific object interfaces is a way to work around the traditionally rigid storage APIs because custom object interfaces give programmers the ability to tell the storage system about their application: if the application is CPU or IO bound, if it has locality, if its size has the potential to overload a single node, etc. Programmers often know what the problem is and how to solve it, but until the ability to modify object interfaces, they had no way to express to the storage system how to handle their data.

Our approach is to expose more of the commonly used, code-hardened subsystems of the underlying storage system as interfaces. The intent is that these interfaces, which can be as simple as a redirection to the persistent data store or as complicated as a strongly consistent directory service, should be used and re-used in many contexts to implement a wide range of services. By making programmability a 'feature', rather than a 'hack' or 'workaround', we help standardize a development process that now is largely ad-hoc.

3. Challenges

Implementing the infrastructure for programmability into existing services and abstractions of distributed storage systems is challenging, even if one assumes that the source code of the storage system and the necessary expertise for understanding it is available. Some challenges include:

- Storage systems are generally required to be highly available so that any complete restarts of the storage system to reprogram them is usually unacceptable.
- Policies and optimizations are usually hard-wired into the services and one has to be careful when factoring them to avoid introducing additional bugs. These policies and optimizations are usually cross-cutting solutions to concerns or trade-offs that cannot be fully explored at the time the code is written (as they relate to workload or hardware). Given these policies and optimizations, decomposition of otherwise orthogonal internal abstractions can be difficult or dangerous.
- Mechanisms that are often only exercised according to hard-wired policies and not in their full generality have hidden bugs that are revealed as soon as those mechanisms are governed by different policies. In our experience introducing programmability into a storage system proved to be a great debugging tool.
- Programmability, especially in live systems, implies changes that need to be carefully managed by the system itself, including versioning and propagation of those changes without affecting correctness.

To address these challenges we present Malacology , our prototype programmable storage system. It uses the programmable storage design approach to evolve storage systems efficiently and without jeopardizing years of code-hardening and performance optimization efforts. Although Malacology uses the internal abstractions of the underlying storage system, including its subsystems, components, and implementations, we emphasize that our system still addresses the general challenges outlined above.

The main challenge of designing a programmable storage system is choosing the right internal abstractions and picking the correct layers for exposing them. A programmable storage is not defined by what abstractions are exposed, rather

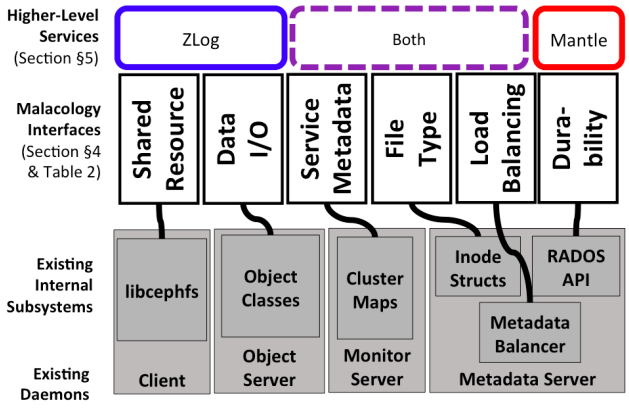

Figure 3: Malacology is implemented on the daemons and clients that run in a Ceph cluster. Interfaces expose internal subsystems and are used as building blocks for higher-level services.

a programmable storage system adheres to the design approach of choosing interfaces so administrators can have better control of the storage system. The interfaces presented in this paper are abstractions that we found useful for building our prototype services ZLog and Mantle, yet they may not provide the best trade-offs for all higher-level services. For example, if consensus is correctly exposed one could implement high-level features like versioning, serialization, or various flavors of strongly consistent data management on top; but perhaps a low-level consensus interface is suited well for a particular set of applications. These questions are not answered in this paper and instead we focus on showing the feasibility of building such a system, given advances in the quality and robustness of today's storage stacks.

The Malacology prototype we present has been implemented on Ceph. While there are other systems on top of which Malacology could be implemented (see Table 2), we choose Ceph because it is a production quality system and because it is open source. The large developer community ensures that code is robust and the visibility of the code lets us expose any interface we want. In the next section we describe the Ceph components that we expose as Malacology interfaces.

4. Malacology: A Programmable Storage System

The guiding principle is to re-use existing services and extend them so that these services can be *programmed*. We accomplish programmability of a service by exporting bindings (or "hooks") for an interpreted programming language so that programming can occur without having to restart the storage system (see also below, §4.4). Table 2 shows the internal services from Ceph that we expose in the Malacology prototype via Lua [26] bindings and Figure 3 compares what was already present in Ceph (gray boxes) to the Malacology interfaces we added. Section 5 will describe the higher-level services we built with these interfaces.

Lua is a portable embedded scripting language and we choose it as the interpreted language for Malacology because it offers superior performance and productivity trade-offs, including a JIT-based implementation that is well known for near native performance. Additionally, Lua has been used extensively in game engines, and systems research [43], including storage systems where it has been effectively used both on [17, 20, 46] and off [36] the performance critical path. Finally, the flexibility of the runtime allows execution sandboxing in order to address security and performance concerns. We will now discuss the common subsystems used to manage storage system and how Malacology makes them programmable.

4.1 Service Metadata Interface

Keeping track of state in a distributed system is an essential part of any successful service and a necessary component in order to diagnose and detect failures, when they occur. This is further complicated by variable propagation delays and heterogeneous hardware in dynamic environments. Service metadata is information about the daemons in the system and includes membership details, hardware layout (*e.g.*, racks, power supplies, etc.), data layout, and daemon state/-configuration. It differs from traditional file system metadata which is information about files. For the rest of the paper when we use the phrase "metadata server" or "metadata service", we are referring to the daemon(s) that manages file system metadata.

Existing Ceph Implementation: In the case of Ceph, a consistent view of cluster state among server daemons and clients is critical to provide strong consistency guarantees to clients. Ceph maintains cluster state information in per-subsystem data structures called "maps" that record membership and status information. A Paxos [29] monitoring service is responsible for integrating state changes into cluster maps, responding to requests from out-of-date clients and synchronizing members of the cluster whenever there is a change in a map so that they all observe the same system state. As a fundamental building block of many system designs, consensus abstractions such as Paxos are a common technique for maintaining consistent data versions, and are a useful system to expose.

The default behavior of the monitor can be seen as a Paxos-based notification system, similar to the one introduced in [13], allowing clients to identify when new values (termed epochs in Ceph) are associated to given maps. Since Ceph does not expose this service directly, as part of our Malacology implementation, we expose a key-value service designed for managing service metadata that is built on top of the consensus engine. Since the monitor is intended to be out of the high-performance I/O path, a general guideline is to make use of this functionality infrequently and to assign small values to maps.

Interface	Section	Example in Production Systems	Example in Ceph	Provided Functionality
Service Metadata	§4.1	Zookeeper/Chubby coordination [13, 25]	cluster state management [28]	consensus/consistency
Data I/O	§4.2	Swift in situ storage/compute [33]	object interface classes [44]	transaction/atomicity
Shared Resource	§4.3.1	MPI collective IO, burst	POSIX metadata protocols	serialization/batching
File Type	§4.3.2	?????	file striping strategy	data/metadata access
Load Balancing	§4.3.3	VMWare's VM migration [16, 23]	migrate POSIX metadata [48]	migration/sampling
Durability	§4.4	S3/Swift interfaces (RESTful API)	object store library [47]	persistence/safety

Table 2: Common internal abstractions. The same "internal abstractions" common across large-scale systems because they provide primitives that solve general distributed systems problems. Here we list examples of what these internal abstractions are used for in "production systems" and in Ceph. Malacology provides these internal abstractions as interfaces (Section 4) that higher level services (Section 5) can use.

Malacology: Malacology exposes a strongly-consistent view of time-varying service metadata as a service rather than a hidden internal component. Malacology provides a generic API for adding arbitrary values to existing subsystem cluster maps. As a consequence of this, applications can define simple but useful service-specific logic to the strongly-consistent interface, such as authorization control (just specific clients can write new values) or triggering actions based on specific values (e.g. sanitize values). The higher-level services we implement in §5 make use of this functionality to register, version and propagate dynamic code (Lua scripts) for new object interfaces defined in storage daemons (§4.2) and policies in the load balancer §4.3. Using this service guarantees that interface definitions are not only made durable, but are transparently and consistently propagated throughout the cluster so that clients are properly synchronized with the latest interfaces.

Impact: provides core functionality because it lets daemons come to consensus on system critical state. Bugs in the internal subsystems or ommitting this from services that need this type of consistency affects correctness.

4.2 Data I/O Interface

Briefly described in Section 2, Ceph supports application-specific object interfaces [47]. The ability to offload computation can reduce data movement, and transactional interfaces can significantly simplify construction of complex storage interfaces that require uncoordinated parallel access.

Existing Ceph Implementation: An object interface is a plugin structured similarly to that of an RPC in which a developer creates a named function within the cluster that clients may invoke. In the case of Ceph each function is implicitly run within the context of an object specified when the function is called. Developers of object interfaces express behavior by creating a composition of native interfaces or other custom object interfaces, and handle serialization of function input and output. A wide range of native interfaces are available to developers such as reading and writing to a byte stream, controlling object snapshots and clones, and accessing a sorted key-value database. These native interfaces may be transactionally composed along with application specific logic to create semantically rich interfaces. An example would be an interface that atomically updates a matrix stored in the bytestream and an index of the matrix stored in the key-value database.

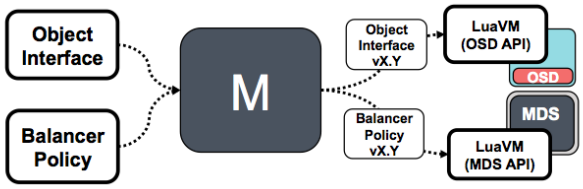

Figure 4: Malacology allows users to dynamically define object/file system metadata interfaces by extending the object storage daemon (OSD) and metadata server (MDS) subsystems with an embedded Lua VM. It uses the service metadata interface (M) to propagate interfaces/versions across the cluster

Malacology: The implementation of Ceph's object abstraction, although powerful, does not readily support programmability. Supporting only C/C++ for object interface developers, Ceph requires distribution of compiled binaries for the correct architecture, adding a large barrier of entry for developers and system administrators. Second, having no way to dynamically unload modules, any changes require a full restart of a storage daemon which may have serious performance impacts due to loss of cached data. And finally, the security limitations of the framework limit the use of object interfaces to all but those with administrative level access and deep technical expertise.

To address these concerns, Malacology takes advantage of Lua extensions contributed by the Ceph community. This allows new object interfaces to be dynamically loaded into the system and modified at runtime, resulting in a object storage API with economy of expression, which at the same time provides the full set of features of the base object class. New object interfaces that are expressed in thousands of lines of code can be implemented in approximately an order of magnitude less code [17]. While the use of Lua does not prevent deployment of malicious code, certain types of coding mistakes can be handled gracefully, and access policies are used to limit access to trusted users [26].

Impact: helps applications optimize performance by push-

ing behavior to lower parts of the storage stack, thereby minimizing hops and distributing computation.

4.3 Distributed Metadata Interfaces

File systems provide clients with the familiar POSIX file abstraction. While this guarantees strong consistency it comes at the cost of scalability, increased complexity, and lower performance. In general, distributed file systems protect resources by providing hierarchical indexing and distributed locking services.

4.3.1 Shared Resource Interface

In general, file system metadata servers manages client sessions, allowing clients to obtain locks (e.g. file byte ranges), and capabilities (e.g. to cache file data). Clients and metadata servers use a cooperative protocol in which clients voluntarily release resources back to the file system metadata service in order to implement sharing policies.

Existing Ceph Implementation: In Ceph, the locking service implements a capability-based system that expresses what data and file system metadata clients are allowed to access as well as what state they may cache and modify locally. While designed for the file abstraction, indexing, locking, and caching are all common services that are useful to a broad spectrum of applications. Distributed applications that share centralized resources (e.g. a database or directory) face similar challenges which are often solved using application-specific sharding.

Malacology: While the current policy for sharing access and voluntarily releasing resources is largely best-effort, Malacology supports generalized policies between metadata servers and clients that can be used to implement fairness or priority.

Impact: provides core functionality to protect and provide exclusive access for any shared resource. May hurt performance if the resource in question does not require strong consistency.

4.3.2 Malacology: File Type Interface

Applications that manage large amounts of file system metadata (e.g. users or database snapshots) often require a naming service. The metadata service manages a POSIX file system hierarchy where files and directories are represented as inode data structures and expose a POSIX file interface.

Existing Ceph Implementation: CephFS is the POSIX compliant file system that uses Ceph. Inodes are quite large (1KB for an inode, 400 bytes for a directory entry, and 700 bytes for a directory) and contain CephFS-specific policies like how to stripe data across RADOS.

Malacology: We allow new inode types to be defined such that applications can create domain-specific interfaces to inodes that may modify locking and capability policies. We will show how this is used in Section 5.2.1 when we discuss a distributed shared-log built on Malacology.

Impact: this interface is both a feature and a performance optimization. It is a feature because it allows developers to add support for different storage types, such as how to read new file formats or what consistency semantics to use for a specific subtree in the hierarchical namespace. It is also a performance optimization because future programmers can add optimizations for processing specific types of files into the inode itself.

4.3.3 Load Balancing Interface

Many large scale storage systems separate file system metadata and data IO so that the corresponding services can scale independently. Metadata requests transfer small amounts of data and they happen relatively frequently so many systems employ separate file system metadata clusters.

Existing Ceph Implementation: Ceph also addresses the challenge of balancing file system metadata load with a separate metadata cluster. This cluster uses load balancing policies to migrate directory inodes around the cluster to alleviate overload at single nodes [48]. The policies use metrics based on system state (e.g. CPU and memory utilization) and statistics collected by the cluster (e.g. the popularity of an inode). Ceph uses dynamic subtree partitioning to move variable sized namespace subtrees. These units can be shipped anywhere (i.e., to any metadata server of any capacity) at any time for any reason. The original balancer was designed with hard-coded policies and tunables.

Malacology: the existing load balancing mechanisms are exposed through an API and programmers can customize the behavior through a domain specific language. These mechanisms include the ability to migrate, partition, and measure load. Using the Service Metadata and Durability interfaces, this Load Balancing interface can safely version balancer policies, save balancer policies in the back-end object store and centralize warnings/errors. When combined with the File Type interface, load balancing policies can express a variety of rules for handling multi-tenant and workloads.

Impact: helps applications optimize performance by allowing them to specify how to partition, replicate, and distribute metadata in response to overloaded servers.

4.4 Durability Interface

Object stores protect data using techniques like erasure coding, replication, and data scrubbing. For scalability, many of these features are implemented using a peer-to-peer protocol that allows object storage devices to operate autonomously

without a centralized coordinator.

Existing Ceph Implementation: Ceph provides storage by striping and replicating data across RADOS [47], the reliable distributed object store. RADOS protects data using common techniques such as erasure coding, replication, and scrubbing. example, when the number of placement groups change, the OSDs re-balance and re-shard data in the background in a process called placement group splitting in which OSDs communicate directly with each other to cover a new state. In order to reduce load on the monitoring service, Ceph OSDs use a gossip protocol to efficiently propagate changes to cluster maps throughout the system, and autonomously initiate recovery mechanisms when failures are discovered.

Malacology: Metadata service policies and object storage interfaces are stored durability within RADOS and managed by storing references with the object maps. Since the cluster already propagates a consistent view of these data structures, we use this service to automatically install interfaces in OSDs, and install policies within the metadata server daemon (MDS) such that clients and daemons are synchronized on correct implementations without restarting.

Impact: this is a feature because it adds data safety and persistence to system metadata; while nice to have it does not necessarily effect correctness.

5. Services Built on Malacology

In this section we describe two services built on top of Malacology. The first is Mantle [36], a framework for dynamically specifying metadata load balancing policies. The second system, ZLog, is a high-performance distributed shared-log. In addition to these services, we'll demonstrate how we combine ZLog and Mantle to implement service-aware metadata load balancing policies.

5.1 Mantle: Programmable Load Balancer

Mantle [36] is a programmable load balancer that separates the metadata balancing policies from their mechanisms. Administrators inject code to change how the metadata cluster distributes metadata. Our previous work showed how to use Mantle to implement a single node metadata service, a distributed metadata service with hashing, and a distributed metadata service with dynamic subtree partitioning. We have re-implmented Mantle on Malacology so that Mantle can enjoy the properties of existing internal abstractions.

The original implementation was "hard-coded" into Ceph and lacked robustness (no versioning, durability, or policy distribution). Re-implemented using Malacology, Mantle now enjoys (1) the versioning provided by the metadata service interface provided by the monitor daemons and (2) the durability and distribution provided by Ceph's reliable object store. Re-using the internal abstractions with Malacology resulted in a $2\times$ reduction in source code compared to the original implementation.

5.1.1 Versioning Balancer Policies

Ensuring that the version of the current load balancer is consistent across the physical servers in the metadata cluster was not addressed in the original implementation. The user had to set the version on each individual server and it was trivial to make the versions inconsistent. Maintaining consistent versions is important for balancing policies that are cooperative in which local decisions are made assuming properties about other instances of the same policy.

With Malacology, Mantle stores the version of the current load balancer in the Ceph service metadata interface. The version of the load balancer corresponds to an object name in the balancing policy. Using the service metadata interface ensures that all metadata servers use the same version of the load balancer. As a result, the policy version inherits the consistency of Ceph's internal monitor daemons. The user changes the version of the load balancer using a new CLI command.

5.1.2 Making Balancer Policies Durable

The load balancer version described above corresponds to the name of an object in RADOS that holds the actual Lua balancing code. When MDS nodes start balancing load, they first check the latest version from the MDS map and compare it to the balancer they have loaded. If the version has changed, they dereference the pointer to the balancer version by reading the corresponding object in RADOS. This is in contrast to the original balancer which stored load balancer code on the local file system – a technique which is unreliable and may result in silent corruption.

The balancer pulls the Lua code from RADOS synchronously; asynchronous reads are not possible because of the architecture of the MDS. The synchronous behavior is not the default behavior for RADOS operations, so we achieve this with a timeout: if the asynchronous read does not come back within half the balancing tick interval the operation is canceled and a Connection Timeout error is returned. By default, the balancing tick interval is 10 seconds, so Mantle will use a 5 second second timeout.

This design allows Mantle to immediately return an error if anything RADOS-related goes wrong. We use this implementation because we do not want to do a blocking OSD read from inside the global MDS lock. Doing so would bring down the MDS cluster if any of the OSDs are not responsive.

Storing the balancers in RADOS is simplified by the use of an interpreted language for writing balancer code. If we used a language that needs to be compiled, like the C++ object classes in the OSD, we would need to ensure binary compatibility, which is complicated by different operating systems, distributions, and compilers.

5.1.3 Logging, Debugging, and Warnings

In the original implementation, Mantle would log all errors, warnings, and debug messages to a log stored locally on each MDS server. To get the simplest status messages or to debug problems, the user would have to log into each MDS individually, look at the logs, and reason about causality and ordering.

With Malacology, Mantle re-uses the centralized logging features of the monitoring service. Important errors, warnings, and info messages are collected by the monitoring subsystem and appear in the monitor cluster log so instead of users going to each node, they can watch messages appear at the monitor. Messages are logged sparingly, so as not to spam the monitor with frivolous debugging but important events, like balancer version changes or failed subsystems show up in the centralized log.

5.2 ZLog: A Fast Distributed Shared Log

The second service implemented on Malacology is ZLog, a high-performance distributed shared-log that is based on the CORFU protocol [7]. The shared-log is a powerful abstraction used to construct distributed systems, such as metadata management [6] and elastic database systems [8–10]. However, existing implementations that rely on consensus algorithms such as Paxos funnel I/O through a single point introducing a bottleneck that restricts throughput. In contrast, the CORFU protocol is able to achieve high throughput using a network counter called a *sequencer*, that decouples log position assignment from log I/O.

While a full description of the CORFU system is beyond the scope of this paper, we briefly describe the custom storage device interface, sequencer service, and recovery protocol, and how these services are instantiated in the Malacology framework.

5.2.1 Sequencer

High-performance in CORFU is achieved using a sequencer service that assigns log positions to clients by reading from a volatile, in-memory counter which can run at a very high throughput and at low latency. Since the sequencer is centralized, ensuring serializability in the common case is trivial. The primary challenge in CORFU is handling the failure of the sequencer in a way that preserves correctness. Failure of the sequencer service in CORFU is handled by a recovery algorithm that recomputes the new sequencer state using a CORFU-specific custom storage interface to discover the tail of the log, while simultaneously invalidating stale client requests using an epoch-based protocol.

Sequencer interface. The sequencer resource supports the ability to *read()* the current tail value and get the *next()* position in the log which also atomically increments the tail position. We implement the sequencer service in Malacology as an application-specific file type that associates a small amount of metadata with the inode state managed by the Ceph metadata service. This approach has the added benefit of allowing the metadata service to handle naming, by representing each sequencer instance in the standard POSIX hierarchical namespace. The primary challenge in mapping the sequencer resource to the metadata service is handling serialization correctly to maintain the global ordering provided by the CORFU protocol.

Initially we sought to directly model the sequencer service in Ceph as a non-exclusive, non-cacheable resource, forcing clients to perform a round-trip access to the resource at the authoritative metadata server for the sequencer inode. Interestingly, we found that the capability system in Ceph uses a strategy to reduce metadata service load by allowing clients that open a shared file to temporarily obtain an exclusive cached copy of the resource, resulting in a round-robin, best-effort batching behavior. When a single client is accessing the sequencer resource it is able to increment the sequencer locally. Any competing client cannot query the sequencer until the metadata service has granted it access.

While unexpected, this discovery allowed us to explore an implementation strategy that we had not previously considered. In particular, for bursty clients, and clients that can tolerate increased latency, this mode of operation may allow a system to achieve much higher throughput than a system with a centralized sequencer service. We utilize the programmability of the metadata service to define a new policy for handling capabilities that controls the amount of time that clients are able to cache the sequencer resource. This allows an administrator or application to control the trade-off between latency and throughput beyond the standard best-effort policy that is present in Ceph by default.

In Section 6 we quantify the trade-offs of throughput and latency for an approach based on a round-robin batching mode, and compare this mode to one in which the metadata server mediates access to the sequencer state when it is being shared among multiple clients. Quantifying these trade-offs should provide administrators with guidelines for setting the tunables for different “caching” modes of the sequencer.

Balancing policies. As opposed to the batching mode for controlling access to the sequencer resource, more predictable latency can be achieved by treating the sequencer inode as a shared non-cacheable resource, forcing clients to make a round-trip to the metadata service. However, the shared nature of the metadata service may prevent the sequencer from achieving maximum throughput. To provide a solution to this issue we have taken advantage of the programmability of the metadata service load balancing to construct a service-specific load balancing policy. As opposed to a balancing policy that strives for uniform load distribution, a ZLog-specific policy may utilize knowledge of inode types to migrate the sequencer service to provisioned hardware during periods of contention or high demand.

5.2.2 Storage Interface

The storage interface is a critical component in the CORFU protocol. Clients independently map log positions that they have obtained from the sequencer service (described in detail in the next section) onto storage devices, while storage devices provide an intelligent *write*-once, random *read* interface for accessing 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 after refreshing state from an auxiliary service. This mechanism forms the basis for sequencer recovery.

In order to repopulate the sequencer state (i.e. the cached, current tail of the log) 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 atomically installs a new epoch value and returns the maximum log position that has been written.

Since the sequencer service does not resume until the recovery process has completed, there cannot be a race with clients appending to the log, and the immutability of the log allow reads to never block during a sequencer failure. Recovery of the sequencer process itself may be handled in many ways, such as leader election using an auxiliary service such as Paxos. In our implementation, the recovery is the same as (and is inherited from) the Ceph metadata service. Handling the failure of a client that holds the sequencer state is similar, although a timeout is used to determine when a client should be considered unavailable.

6. Evaluation

Our evaluation demonstrates the feasibility of building new service abstractions atop programmable storage, focusing on the performance of the internal abstractions exposed by Malacology and used to construct the Mantle and ZLog services. We also discuss latent capabilities we discovered in this process that let us navigate different trade-offs within the services themselves. First, we benchmark scenarios with high sequencer contention by examining the interfaces used to map ZLog onto Malacology; specifically, we describe the sequencer implementation and the propagation of object and data interfaces interfaces. Next, we benchmark scenarios in which the storage system manages multiple logs by using Mantle to balance sequencers across a cluster.

Since this work focuses on the programmability of Malacology, the goal of this section is to show that the components and subsystems that support the Malacology interfaces provide reasonable relative performance, as well as to give examples of the flexibility that Malacology provides to programmers.

6.1 Mapping ZLog onto Malacology

We evaluate Malacology by exploring one possible mapping of the ZLog implementation of CORFU onto Ceph in which we re-use (1) the metadata service to manage naming and synchronization of the sequencer resource by treating the resource as an inode, and (2) the monitoring subsubsystem to distribute and install application-specific I/O interface required of the CORFU protocol. In Section 6.2 we then demonstrate how the re-use of the inode abstraction for implementing the sequencer resource enables load balancing policies to migrate the sequencer resource in heavy-load situation.

6.1.1 Sequencer Implementation

We evaluate the feasibility of using the metadata service to implement a sequencer resource that is responsible for maintaining a total ordering of the log. Clients contact the sequencer to obtain the tail of the log and then independently initiate I/O, thus we measure both the throughput and latency of obtaining new tail positions which bounds client append performance.

The sequencer is implemented as a custom file type in which the sequencer state (a 64-bit integer) is embedded in the inode of a file representing the sequencer. A total ordering of the log is imposed by the re-use of the capability service that can be used to grant exclusive access of inode state to clients. The metadata service is responsible for maintaining exclusivity and granting access. Figure 5 (a) shows the behavior of the system in which a best-effort policy is used. The two colors represent points in time that the clients were able to access the resource. The best-effort policy shows a high degree of interleaving between clients but the system spends a large portion of time re-distributing the capability, reducing overall throughput.

In order to control the performance of the system we implement a policy that (1) restricts the length of time that a client may maintain exclusive access and (2) limits the number of log positions that a client may generate without yielding to other clients waiting for access. The behavior of these two modes is illustrated in Figures 5 (b) and (c), respectively.

Figure 6 demonstrates a configurable trade-off between throughput and latency. In the experiment two clients are run each with a fixed 0.25 second maximum reservation on the capability, and we vary the size of the log position quota running each configuration for two minutes. The total operations per second is the combined throughput of the two clients, and the average latency is number of microseconds required to obtain a new log position. With a small quota more time is spent exchanging exclusive access, while a large quota reservation allows clients to experience a much lower latency because they experience uncontended access for a longer period of time.

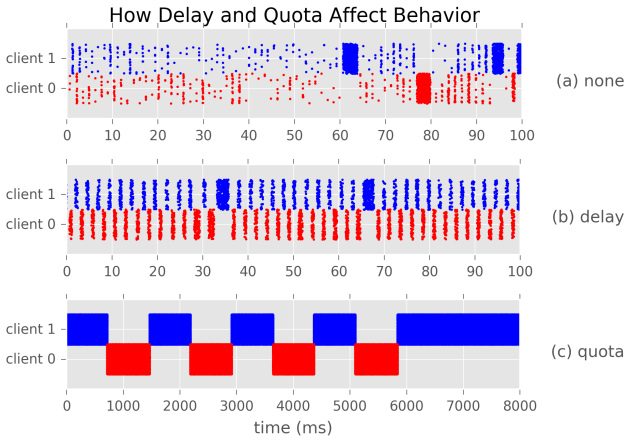

Figure 5: Each dot is an individual request, spread randomly along the y axis. The default behavior is unpredictable, "delay" lets clients hold the lease longer, and "quota" gives clients the lease for a number of operations.

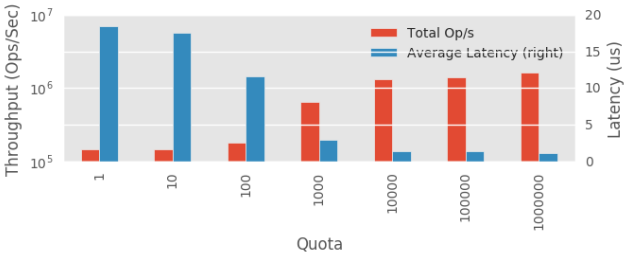

Figure 6: Sequencer throughput by re-using various services. The highest performance is achieved using a single client with exclusive, cacheable privilege. Round-robin sharing of the sequencer resource is affected by the amount of time the resource is held, with best-effort performing the worst.

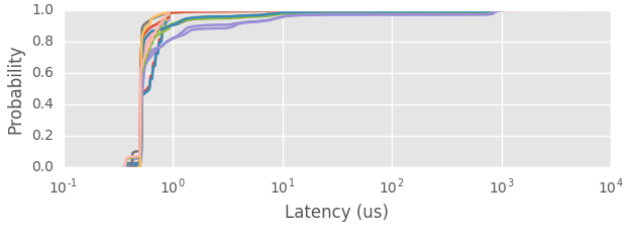

Figure 7: The latency distribution shows that if the clients hold their capabilities longer, performance decreases. Changing the delay can help the sequencer and clients strike a balance between throughput and latency.

To get a better picture of latency, Figure 7 shows the CDF of latency for each client in all experiment configurations. At the 99th percentile clients accessed the sequencer is less than a millisecond. The CDF is cropped at the 99.999th percentile due to large outliers that we believe occur in instances in which the MDS is performing I/O while it is in the process of re-distributing the capability to another client.

Malacology exposes the internal capability management service and allows users to navigate latency and throughput trade-offs. Other approaches to designing the sequencer service also exist, such as using a centralized service in which each access is a network round-trip. In contrast to the mechanism we explored which is appropriate for clients with bursty workloads, it may be easier to provide predictable performance using a centralized service, and we will be exploring in the future how this can be achieved using the capability system.

6.1.2 Interface Propagation

Domain-specific data interfaces (Section 2) allow co-design between applications and the storage system. Malacology supports custom object interfaces in RADOS that require interface implementations to be installed on the storage devices in the system, supporting the evolution of interfaces through automatic system-wide versioning and installation through the service metadata interface (Section 4.1). We evaluate the performance of installing a new interface version in the cluster, which is an important metric for applications that frequently evolve interfaces.

We demonstrate the feasibility of utilizing the Ceph monitoring sub-system by evaluating the performance of installing and distributing interface updates. Figure 8 shows the CDF of the latency of interface updates. The interfaces are Lua scripts embedded in the cluster map and distributed using a peer-to-peer gossip protocol. The latency is defined as the elapsed time following the Paxos proposal for an interface update until each OSD makes the update live (the cost of the Paxos proposal is configurable and is discussed below). The latency measurements were taken on the nodes running Ceph server daemons, and thus exclude the client round-trip cost. In each of the experiments 1000 interface updates were observed.

Figure 8 shows the lower bound cost for updates in a large cluster. In the experiment labeled "120 OSD (RAM)" a cluster of 120 OSDs (10 OSDs x 12 servers) using an in-memory data store were deployed, showing a latency of less than 54 ms with a probability of 90% and a worst case latency of 194 ms. These costs demonstrate the penalty of distributing the interface in a large cluster. In practice the costs include, in addition to cluster-wide propagation of interface updates, the network round-trip to the interface management service, the Paxos commit protocol itself, and other factors such as system load. By default Paxos proposals occur periodically with a 1 second interval in order to accumulate updates. In a minimum, realistic quorum of 3 monitors using HDD-based storage, we were able to decrease this interval to an average of 222 ms.

6.2 Load Balancing ZLog Sequencers with Mantle

In practice, a storage system implementing CORFU will support a multiplicity of independent totally-ordered logs for each application. For this scenario co-locating sequencers on the same physical node is not ideal but building a load balancer that can migrate the shared resource (*e.g.*,, the resource that mediates access to the tail of the log) is a time-consuming, non-trivial task. It requires building subsystems for migrating resources, monitoring the workloads, collect-

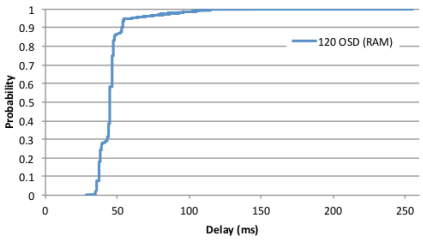

Figure 8: Cluster-wide interface update latency, excluding the Paxos proposal cost for interface commit.

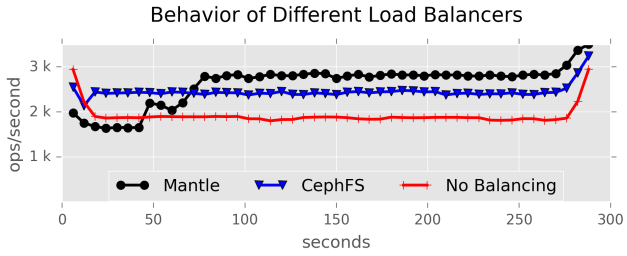

Figure 9: CephFS/Mantle load balancing have better throughput than co-locating all sequencers on the same server. Sections 6.2.1 and 6.2.2 quantify this improvement; Section 6.2.3 examines the migration at 0-60 seconds.

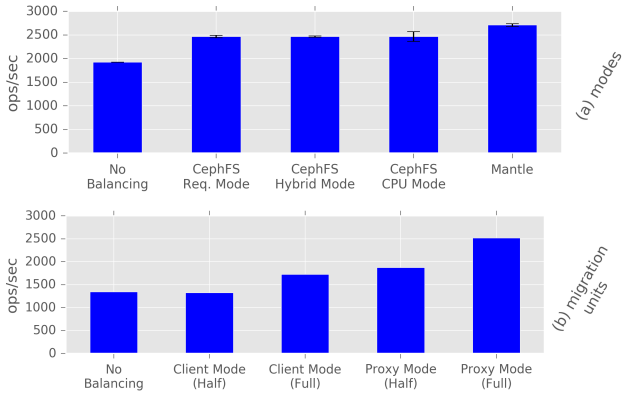

Figure 10: In (a) all CephFS balancing modes have the same performance; Mantle uses a balancer designed for sequencers. In (b) the best combination of mode and migration units can have up to a 2× improvement.

ing metrics that describe the utilization on the physical nodes, partitioning resources, maintaining cache coherence, and managing multiple sequencers. The following experiments demonstrate the feasibility of using the mechanisms of the Malacology load balancing interface to inherit these features and to alleviate load from overloaded servers.

The experiments are run on a cluster with 10 nodes to store objects, one node to monitor the cluster, and 3 nodes that can accommodate sequencers. Instead of measuring contention at the clients like Section 6.1.1, these experiments measure contention at the sequencers by forcing clients to make round-trips for every request. We implement this using the shared resource interface that forces round-trips. Because the sequencer's only function is to hand out positions for the tail of the log, the workload is read-heavy.

First, we show how the ZLog service can orchestrate multiple sequencers using the Malacology load balancing interface. Figure 9 shows the throughput over time of different load balancers as they migrate 3 sequencers (with 4 clients) around the cluster; "No Balancing" keeps all sequencers on one server, "CephFS" migrates sequencers using the CephFS load balancers, and "Mantle" uses a custom load balancer we wrote specifically for sequencers. The CephFS load balancers are hard-coded into Ceph while Mantle is an API for balancing load presented in Mantle and re-implemented on top of Malacology in this paper. The increased throughput for the CephFS and Mantle curves between 0 and 60 seconds are a result of migrating the sequencer(s) off overloaded servers.

In addition to showing that migrating sequencers improves performance, Figure 9 also demonstrates features that we will explore in the rest of this section. Sections 6.2.1 and 6.2.2 quantify the differences in performance when the cluster stabilizes at time 100 seconds and Section 6.2.3 examines the slope and start time of the rebalancing phase between 0 and 60 seconds by comparing the aggressiveness of the balancers.

6.2.1 Feature: Balancing Modes

Next, we quantify the performance benefits shown in Figure 9. To understand why load balancers perform differently we need to explain the different balancing modes that the load balancer service uses and how they stress the subsystems that receive and forward client requests in different ways. In Figure 9, the CephFS curve shows the performance of the balancing mode that CephFS falls into *most of the time*. CephFS currently has 3 modes for balancing load: CPU mode, workload mode, and hybrid mode. All three have the same structure for making migration decisions but vary based on the metric used to calculate load. For this sequencer workload the 3 different modes all have the same performance, shown in Figure 10 (a), because the load balancer falls into the same mode a majority of the time. The high variation in performance for the CephFS CPU Mode bar reflects the uncertainty of using something as dynamic and unpredictable as CPU utilization to make migration decisions. In addition to the suboptimal performance and unpredictability, because CephFS balancers all fall into the same mode despite using different balancers is a problem because it prevents administrators from properly exploring the balancing state space.

Mantle gives the administrator more control over balancing policies; for the Mantle bar in Figure 10 (a) we use the load balancing interface to program logic for balancing read-heavy workloads, resulting in better throughput and stability. When we did this we also identified two balancing modes relevant for making migration decisions for sequencers.

Using Mantle, the administrator can put the load balancing service into "proxy mode" or "client mode". In proxy mode one server receives all requests and farms off the re-

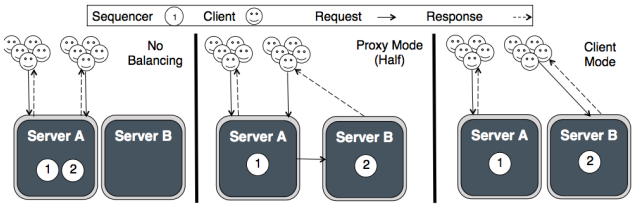

Figure 11: In client mode clients sending requests to the server that houses their sequencer. In proxy mode clients continue sending their requests to the first server.

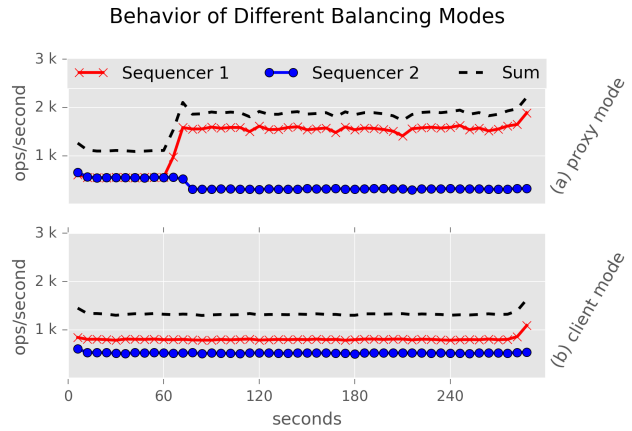

Figure 12: The performance of proxy mode achieves the highest throughput but at the cost of lower throughput for one of the sequencers. Client mode is more fair but results in lower cluster throughput.

quests to slave servers; the slaves servers do the actual tail finding operation. In client mode, clients interact directly with the server that has their sequencer. These modes are illustrated in Figure 11: "No balancing" is when all sequencers are co-located on one physical server – performance for that mode is shown by the No Balancing curve in Figure 9. "Client Mode" in the lower left shows how clients for sequencer 2 are re-directed to server B. The two "Proxy mode" diagrams on the right show how clients continue sending requests to server A even though sequencer 2 has migrated to server B (the meaning of "(Half)" is discussed in the next section).

Figure 12 shows the throughput over time of the two different modes for an environment with only 2 sequencers (again 4 clients each) and 2 servers. The curves for both sequencers in Figure 12(a) start at less than 1000 ops/second and at time 60 seconds Mantle migrates Sequencer 1 to the slave server. Performance of Sequencer 2 decreases because it stayed on the proxy which now handles all client requests, processes requests for Sequencer 2, and forwards requests for Sequencer 1. The performance of Sequencer 1 improves dramatically because distributing the sequencers in this way separates (1) the handling of the client requests and (2) finding the tail of the log and responding to clients. Doing both steps is too heavy weight for one server and sequencers on slave nodes can go faster if work is split up; this phenomenon is not uncommon and has been observed in chain replication [42].

Cluster throughput improves at the cost of decreased throughput for Sequencer 2. Figure 12(b) is set to sequencer mode manually (no balancing phase) and shows that the cluster throughput is worse than the cluster throughput of proxy mode. That graph also shows that Sequencer 2 has less throughput than Sequencer 2. In this case, the scatter-gather process used for cache coherence in the metadata protocols causes strain on the server housing Sequencer 2 resulting in this uneven performance.

6.2.2 Feature: Migration Units

Another factor that affects performance in this environment is how much load is on each server; these experiments quantify that effect by programming the load balancing interface to control the amount of load to migrate. We call this metric a "migration unit". Expressing this heuristic is not easily achievable with outward facing tunable parameters (i.e. system knobs) but with Mantle's programmable interface, we can force the load balancer to change its migration units. To force the balancer into the Proxy Mode (Half) scenario in Figure 11, which uses migration units equal to half the load on the current server, we can use: `targets[whoami+1] = mds[whoami]["load"]/2` .

This code snippet uses globally defined variables and tables from the Mantle API to send half of the load on the current server (whoami) to the next ranked server (whoami + 1); the `targets` array is a globally defined table that the balancer uses to do the migrations. Alternatively, to migrate all load a time step, we can remove the division by 2.

Figure 10 (b) shows the performance of the modes using different migration units. Recall that this setup only has 2 sequencers and 2 servers, so performance may be different at scale. Even so, it is clear that client mode does not perform as well for read-heavy workloads. We even see a throughput improvement when migrating all load off the first server, leaving the first server to do administrative tasks (this is common in the metadata cluster because the first server does a lot of the cache coherence work) while the second server does all the client work. Proxy mode does the best in both cases and shows large performance gains when completely decoupling client request handling and operation processing in Proxy Mode (Full). The parameter that controls the migration units helps the administrator control the sequencer co-location or distribution across the cluster. This trade-off was explored extensively in the Mantle paper but the experiments we present here are indicative of an even richer set of states to explore.

6.2.3 Feature: Backoff

Tuning the aggressiveness of the load balancer decision making is also a trade-off that administrators can control and explore. The balancing phase from 0 to 60 seconds in Figure 9 shows different degrees of aggressiveness in making

migration decisions; CephFS makes a decision 10 seconds into the run and throughput jumps to 2500 ops/second within 10 seconds while Mantle takes more time to stabilize. This conservative behavior is controlled by programming the balancer to (1) use different conditions for when to migrate and (2) using a threshold for sustained overload.

We control the conditions for when to migrate using `when()`, a callback in the Mantle API. For the Mantle curve in Figure 9 we program `when()` to wait for load on the receiving server to fall below a threshold. This makes the balancer more conservative because it takes 60 seconds for cache coherence messages to settle. The Mantle curve in Figure 9 also takes longer to reach peak throughput because we want the policy to wait to see how migrations affect the system before proceeding; the balancer does a migration right before 50 seconds, realizes that there is a third underloaded server, and does another migration.

The other way to change aggressiveness of the decision making is to program into the balancer a threshold for sustained overload. This forces the balancer to wait a certain number of iterations after a migration before proceeding. In Mantle, the policy would use the save state function to do a countdown after a migration. Behavior graphs and performance numbers for this backoff feature is omitted for space considerations, but our experiments confirm that the more conservative the approach the less overall throughput.

Malacology pulls the load balancing service out of the storage system to balance sequencers across a cluster. This latent capability also gives future programmers the ability to explore the different load balancing trade-offs including: load balancing modes to control forwarding vs. client redirection, load migration units to control sequencer distribution vs. co-location, and backoffs to control conservative vs. aggressive decision making.

7. Future Work

Malacology is a first step towards showing how general-purpose storage systems can be adapted to target special-purpose applications. By encapsulating storage system functionality as reusable building blocks, we enable application developers to leverage storage capabilities based on interfaces that are proven and understandable. However, creation and composition of interfaces is complex; constructs must be combined safely in order to provide correctness, performance and security. We will study additional Malacology-based services in order to learn techniques that support safe composition.

Some higher-level services that we plan to build using the interfaces in Table 2 are: an elastic cloud database, a data processing engine, and a data layout manager. Approaches proposed so far use the data I/O interface to push down predicates and computation, the file type interface to maintain access paths and metadata efficiently, and the durability interface to manage ingestion and movement. Using the programmable storage approach helps us build higher-level services that work well with the storage system not in spite of it.

Our experience with ZLog and Mantle demonstrates that the labor of wrapping existing services in reusable interfaces is justified by the power and flexibility that this encapsulation affords to programmers. In exchange for this flexibility, however, programmers may forfeit the *protection from change* afforded by narrow storage interfaces such as the POSIX API. To implement applications on programmable storage systems such as Malacology, programmers must find solutions by navigating a complex design space, simultaneously addressing functional correctness, performance and fault tolerance. Worse still, their solutions may be sensitive to changes in the underlying environment, such as hardware upgrades, software version changes and evolving workloads. For example, a major version change in Ceph required us to rewrite significant parts of ZLog to maintain acceptable performance. Each such evolution costs developer time and risks introducing bugs.

We are actively exploring the use of high-level *declarative* languages based on Datalog [2] to program data access and storage APIs. Using this approach, a systems programmer can specify the functional behavior in a relational (or algebraic) language, allowing an optimizer to search through the space of functionally equivalent physical implementations and select a good execution plan, re-optimizing when storage characteristics or statistics change. Much like query planning and optimization in database systems [24], this approach will separate the concerns of correctness and performance, protecting applications (which usually evolve slowly) against changes in more dynamic storage system.

8. Related Work

Programmability of operating systems and networking resources, including distributed storage systems is not new, but we are not aware of work that makes generalization of existing services into programmable resources a key principle in storage systems design.

Programmable storage systems can be viewed as an infrastructure for creating abstractions to better separate policies from mechanisms. This idea is not new. Software-defined networks (SDNs) create such an abstraction by separating the control plane from the data plane (see for example [27]). This notion of control/data separation was also applied in software-defined storage (SDS) [40, 41]. Similarly, IOStack [19] is providing policy-based provisioning and filtering in OpenStack Swift. According to a SNIA white paper [14], 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. A programmable storage system exposes internal abstractions so that end users (not necessarily operators) can create new services on top of the storage stack. Thus, 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.

Another view of programmable storage systems is one of tailoring systems resources to applications [5]. Related efforts include the Exokernel [15], SPIN [11] and Vino [35] projects; the latter two addressed the ability of injecting code into the kernel to specialize resource management. Another approach is to pass hints between the different layers of the IO stack to bridge the semantic gap between applications and storage [5, 32, 38].

Malacology uses the same Active and Typed Storage module presented in DataMods [45]; Asynchronous Service and File Manifolds can be implemented with small changes to the Malacology framework, namely asynchronous object calls and Lua stubs in the inode, respectively.

9. Conclusion

Programmable storage is a viable method for eliminating duplication of complex error-prone software used as workarounds for storage system deficiencies. We propose that systems expose their services in a safe way allowing application developers to customize system behavior to meet their needs while not sacrificing correctness. To illustrate the benefits of this approach we presented Malacology, a programmable storage system that facilitates the construction of new services by re-purposing existing subsystem abstractions of the storage stack.

References

[1] Ceph Architecture, http://docs.ceph.com/docs/master/architecture.

[2] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. Consistency Analysis in Bloom: A CALM and Collected Approach. In *In Proceedings 5th Biennial Conference on Innovative Data Systems Research*, pages 249–260, 2011.

[3] Apache Parquet Contributors. Parquet Columnar Storage Format, http://parquet.io.

[4] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. 53, 2010.

[5] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and control in gray-box systems. In *SOSP '01*, Banff, Alberta, Canada, 2001.

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

[7] 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.

[8] Philip A. Bernstein, Sudipto Das, Bailu Ding, and Markus Pilman. Optimizing optimistic concurrency control for tree-structured, log-structured databases. In *SIGMOD '15*, 2015.

[9] 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.

[10] Philip A. Bernstein, Colin W. Reid, Ming Wu, and Xinhao Yuan. Optimistic concurrency control by melding trees. In *VLDB '11*, 2011.

[11] B. N. Bershad et al. Extensibility safety and performance in the spin operating system. In *SOSP '95*, Copper Mountain, CO, 1995.

[12] Eric Brewer, Lawrence Ying, Lawrence Greenfield, Robert Cypher, and Theodore T'so. Disks for Data Centers. Technical Report, Google, 2016.

[13] Mike Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06.

[14] Mark Carlson, Alan Yoder, Leah Schoeb, Don Deel, Carlos Pratt, Chris Lionetti, and Doug Voigt. Software defined storage.

[15] D. R. Engler, M. F. Kaashoek, and Jr. J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *SOSP '95*, Copper Mountain, CO, 1995.

[16] Epping, Duncan and Denneman, Frank. VMware vSphere 5.1 Clustering Deepdive, accessed 03/21/2014, http://www.vmware.com/product/drs.

[17] Roxana Geambasu, Amit A. Levy, Tadayoshi Kohno, Arvind Krishnamurthy, and Henry M. Levy. Comet: an active distributed key-value store. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10.

[18] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *SOSP '03*, Bolton Landing, NY, 2003. ACM.

[19] Raúl Gracia-Tinedo et al. Iostack: Software-de ned object storage. *IEEE Internet Computing*, 20(3):10–18, May-June 2016.

[20] Matthias Grawinkel, Tim Sub, Gregor Best, Ivan Popov, and Andre Brinkmann. Towards Dynamic Scripted pNFS Layouts. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, 2012.

[21] Jim Gray. Tape is dead, disk is tape, flash is disk, RAM locality is king. *CIDR 2007 - Gong Show Presentation*, 2007.

[22] Jim Gray and Bob Fitzgerald. Flash Disk Opportunity for Server Applications. 6, 2008.

[23] Ajay Gulati, Ganesha Shanmuganathan, Anne Holler, and Irfan Ahmad. Cloud-scale resource management: challenges and techniques. In *Proceedings of the 3rd USENIX confer-*

ence on Hot topics in cloud computing, HotCloud'11, pages 3–3, Berkeley, CA, USA, 2011. USENIX Association.

[24] Joseph M Hellerstein and Michael Stonebraker. Anatomy of a database system. *Readings in Database Systems,*, 2005.

[25] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10.

[26] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes Filho. Lua-an extensible extension language. *Softw., Pract. Exper.*, 26(6):635–652, 1996.

[27] S. Jain et al. B4: Experience with a globally-deployed software defined wan. In *SIGCOMM '13*, Hong Kong, China, 2013.

[28] Luis Joao. Ceph's New Monitor Changes, https://ceph.com/dev-notes/cephs-new-monitor-changes.

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

[30] Linux Foundation. Kinetic Open Storage Project, 2015.

[31] John MacCormick, Nick Murphy, Marc Najork, andramohan A. Thekkath, and Lidong Zhou. Boxwood: Abstractions as the foundation for storage frastructure. San Francisco, CA, December 2004.

[32] Michael Mesnier, Feng Chen, and Jason B. Akers. Differentiated storage services. In *SOSP '11*, Cascais, Portugal, October 23-26 2011.

[33] Rackspace. Zerovm and openstack swift, http://www.zerovm.org/zerocloud.html.

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

[35] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *OSDI'96*, Seattle, WA, 1996.

[36] Michael A. Sevilla, Noah Watkins, Carlos Maltzahn, Ike Nassi, Scott A. Brandt, Sage A. Weil, Greg Farnum, and Sam Fineberg. Mantle: A programmable metadata load balancer for the ceph file system. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, 2015.

[37] Konstantin V. Shvachko, Hairong Kuang, Sanjay Radia, and bert Chansler. The hadoop distributed file system. In *MSST2010*, Incline Village, NV, May 3-7 2010.

[38] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dussea. Semantically-smart disk systems. San Francisco, CA, March 2003.

[39] SNIA. Implementing Multiple Cloud Storage APIs, November 2014.

[40] Ioan Stefanovici, Bianca Schroeder, Greg O'Shea, and Eno Thereska. sroute: Treating the storage stack like a network. In *FAST '16*, Santa Clara, CA, 2016.

[41] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. Ioflow: A software-defined storage architecture. In *SOSP '13*, Farmington, PA, November 3-6 2013.

[42] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 7–7, Berkeley, CA, USA, 2004. USENIX Association.

[43] Lourival Vieira Neto, Roberto Ierusalimschy, Ana Lúcia de Moura, and Marc Balmer. Scriptable Operating Systems with Lua. In *Proceedings of the 10th ACM Symposium on Dynamic Languages*, DLS '14, pages 2–10, New York, NY, USA, 2014. ACM.

[44] Noah Watkins. Dynamic object interfaces with lua, http://ceph.com/rados/dynamic-object-interfaces-with-lua.

[45] Noah Watkins, Carlos Maltzahn, Scott Brandt, and Adam Manzanares. DataMods: Programmable File System Services. In *PDSW'12*, 2012.

[46] Noah Watkins, Carlos Maltzahn, Scott Brandt, Ian Pye, and Adam Manzanares. In-Vivo Storage System Development. In *Euro-Par 2013: Parallel Processing Workshops*, pages 23–32. Springer, 2013.

[47] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, and Carlos Maltzahn. RADOS: a scalable, reliable storage service for petabyte-scale storage clusters. In *Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing '07*, PDSW '07, 2007.

[48] Sage A. Weil, Kristal T. Pollack, Scott A. Brandt, and Ethan L. Miller. Dynamic Metadata Management for Petabyte-Scale File Systems. In *Proceedings of the 17th ACM/IEEE Conference on Supercomputing*, SC'04, 2004.