# Malacology: A Programmable Storage System

Paper ID: 123

Total Page: 12


## Abstract

Storage systems are caught between rapidly changing data processing applications and the increasing speed of storage devices. This puts tremendous pressure on storage systems to support rapid evolution 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 Malacology, a prototype programmable storage system to explore how existing abstractions of common services found in storage systems can be leveraged to address new data processing systems and the increasing speed of storage devices. This approach allows a large degree of flexibility for storage systems to evolve without sacrificing the robustness of their code-hardened subsystems. We illustrate the advantages and challenges of programmability by composing existing primitives into two new higher-level services: a file system metadata load balancer and a high-performance distributed shared-log that leverages flash devices. The evaluation demonstrates that our services inherit desirable qualities of the back-end storage system, including the ability to balance load, efficiently propagate cluster metadata, recover from failure, and to make trade-offs for latency/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 [13]. 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 [3].

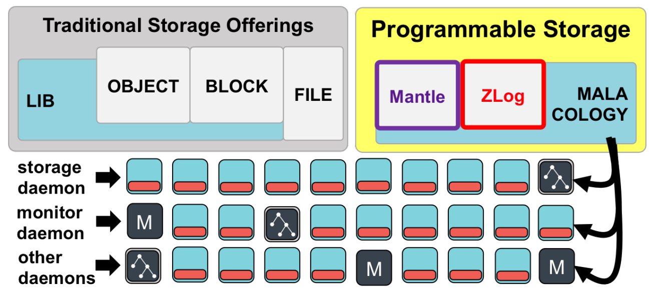


**Figure 1.** Scalable storage systems consist of storage daemons which store data fragments, monitor daemons that maintain cluster state, and service-specific daemons such as file system metadata servers. Malacology enables the programmability of internal abstractions (indicated by bold arrows) in order to re-use and compose existing subsystems. With Malacology, we add two new services, ZLog and Mantle, that sit alongside the traditional user-facing APIs (file, block, and object) in the storage stack.

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 [22, 23]. Second, emerging "big data" applications demand interface evolution to support flexible consistency as well as flexible structured data representations. [2]. 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 [27, 32].

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 paper 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 abstraction one can reuse subsystems and leverage their optimizations, established correctness, robustness, and efficiency.

***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 metadata services and synchronization and monitoring capabilities) as interfaces that can be "glued together" in a variety of ways using a high-level language. In contrast to programmable storage, *active storage* differs through its focus on the injection and execution of code within a storage system or storage device.

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 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). As we will show in this paper, Malacology is expressive enough to provide the functionality necessary for implementing new services. Our contributions are:

A prototype programmable storage system that 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 underyling storage stack's object store.

***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 [30]

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 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 (§**??**). Finally we discuss related work and conclude.

## 2. Application-Specific Storage Stacks

Building storage stacks from the ground up for a specific purpose results in the best performance. For example, HDFS was designed specifically to serve Hadoop's jobs, and uses techniques like exposing data locality and relaxing POSIX constraints to to achieve application-specific I/O optimizations. [**?** ]. Alternatively, general-purpose storage stacks are built with the flexiblity 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 this trend in 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 [15]. Ceph's Reliable Autonomous Distributed Object Storage (RADOS) system is a cluster of storage devices (OSDs) that provide Ceph with data durability and integrity using replication, erasure-coding, and scrubbing [40]. 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 groupings 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 a gap 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.

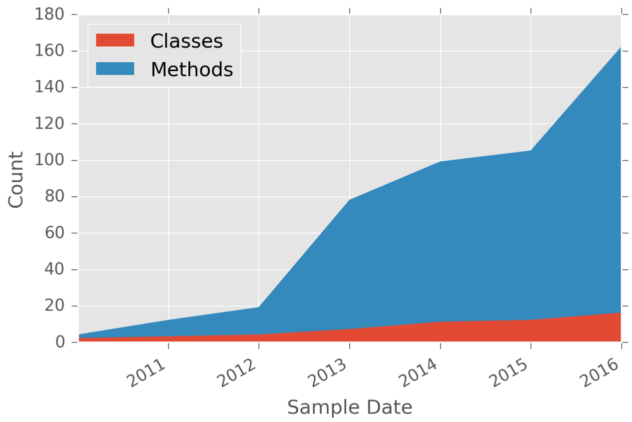


**Figure 2.** [source] Since 2010, the growth in the number of co-designed object storage interfaces in Ceph has been accelerating. This plot is the number of object classes (a group of interfaces), and the total number of methods (the actual API end-points).

| Category | Specialization | # |
|---|---|---|
| Locking | Shared<br>Exclusive | 6 |
| Logging | Replica | 3 |
| | State | 4 |
| | Timestamped | 4 |
| Metadata Managment | RADOS Block Device | 37 |
| | RADOS Gatway | 27 |
| | User | 5 |
| | Version | 5 |
| Garbage Collection | Reference Counting | 4 |

**Table 1.** A variety of RADOS object storage classes exist to expose interfaces to applications. # is the number of methods that use 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 it is CPU or IO bound, if it has locality, if its size has the potential to overload a single proxy node, etc. The programmers 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:

- 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 out not to introduce additional bugs. These policies and optimiations are usually cross-cutting solutions to concerns or trade-offs that cannot be fully explored yet (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. Malacology is both a prototype for a programmable storage system and a 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. In this paper, we do not present an exhaustive list of possible internal abstractions nor do we contend that the abstractions we choose provide the best trade-offs for all applications. 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.

Malacology implements and demonstrates the ideas of programmable storage in Ceph. 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

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).

There are multiple reasons that make Lua an attractive runtime for implementing Malacology. Lua is a portable, embedded scripting language that 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 [36], including storage systems where it has been effectively used both on [19, 21, 39] and off [30] 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 the state of 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 heterogenous hardware in highly dynamic environments.

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-based 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 [14], allowing clients to identify when new values (termed epochs in Ceph) are associated to given maps. While Ceph does not expose this service directly, a key-value service designed for managing configuration metadata that is built on top of the consensus engine is an example of a high-level service within Ceph that re-uses existing abstractions. While a key-value service is generally useful, it doesn't provide many of the useful services hidden within the monitoring framework that would be required for applications with more demanding requirements, such as creating arbitrary maps or associating application-specific logic that can be executed for particular maps (or values in a map). Since the monitor is intended to be out of high-performance I/O paths, a general guideline is to make use of this functionality infrequently and to assign small values to maps.

***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 to trigger 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 (§**??**) 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 throughput the cluster and that clients are properly synchronized with the latest interfaces.

### 4.2 Object & Data Interface

Briefly described in Section 2, Ceph supports the creation of application-specific object interfaces [40]. 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.

An object interface is a plugin structured in a similar way to that of an RPC in which a developer creates a named func-

## Common Internal Abstractions

| Abstraction | Example in Production Systems | Example in Ceph | Provided Primitives |
|---|---|---|---|
| *Service Metadata* | Zookeeper coordination [25], Chubby [12] | cluster state management [34] | consensus & consistency |
| *Shared Resource* | MPI collective IO, burst | POSIX metadata protocols [? ] | serialization & batching |
| *Object & Data* | Swift in situ storage/compute [1] | object interface classes [37] | transaction & atomicity |
| Durability | S3/Swift interfaces (RESTful API) | object store library [40] | interface to persistence |
| Load Balancing | VMWare's VM migration [18, 24] | migration in metadata cluster [41] | migration & sampling |
| Beacon | Apache RabbitMQ [? ] | inter-daemon messages, centralized logging | debugging & monitoring |

**Table 2.** The same "internal abstractions" are 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 that higher level applications can use; *italicized abstractions* are interfaces contributed in this paper.

tion 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, and controlling snapshots and clones. These object interfaces may be transactionally composed with access to a sorted key-value service allowing applications to create rich semantics. An example is using a read-modify-write guard to protect a shared index defined over content stored in a byte stream.

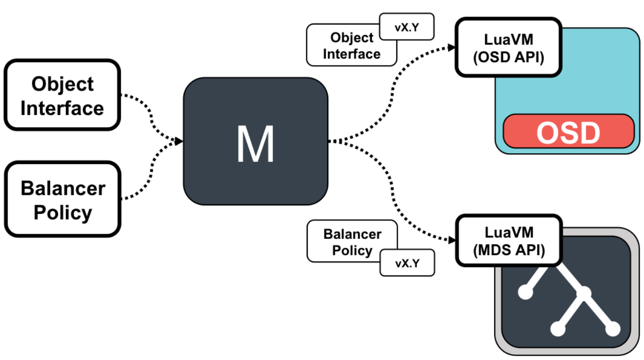


**Figure 3.** Malacology re-uses Ceph subsystems to allow users to dynamically define new object/data interfaces and load balancing policies. It achieves this by extending the object storage daemon (OSD) and metadata server daemon (MDS) subsystems with an embedded Lua VM. It uses the of the strongly-consistent service metadata interface to propogate the object interfaces and 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 [19].

All of these modifications work in tandem to implement the desired behavior, and are typically co-designed together such that each depends on the other to behave as expected.

### 4.3 Distributed Metadata Interfaces

The distributed metadata service in Ceph provides clients with a POSIX file system abstraction [41]. In general, distributed file systems protect resources by providing hierarchical indexing and distributed locking services. In Ceph, the locking service implements a capability-based system that expresses what data and metadata clients are allowed to access as well as what state they may cache and modify locally. While designed for a fixed 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.

Ceph also addresses the challenge of balancing metadata load with a separate metadata cluster. This cluster uses load balancing policies to migrate directory inodes around the cluster to alleviate load. 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:*** interfaces are added in strategic spots in the metadata service for guarding shared resources and load balancing.

**Shared Resource Interface.** The metadata service also manages sessions with its clients. This gives both subsystems (metadata servers and clients) the ability to cache data and obtain locks. The metadata service is responsible for responding to client requests for capabilities by first revoking caps from other clients. Currently, clients voluntarily release their capabilities, and the metadata server maintains a queue of requests. Both clients and metadata servers use a best-effort approach to revoking or releasing capabilities.

The clients retain the best-effort, voluntary release of capabilities, but in Malacology, the queuing policies are generalized within the MDS to allow for centralized policies such as fairness or priority.

**Load balancing.** Mantle [30], a programmable metadata load balancer, has been implemented on Malacology so that its infrastructure can enjoy all the properties of existing internal abstractions. The load balancing logic is intact but the infrastructure has been improved to safely load, manage, and persist its load balancing policies. Composing Mantle using the service metadata, durability and beacon interfaces, Mantle can safely version balancer policies, save balancer policies in the back end object store and centralize warnings/errors.

### 4.4 Durability Interface

Ceph provides storage by striping and replicating data across RADOS [40], the reliable distributed object store. RADOS uses many techniques to ensure that data is not corrupted or lost, such as erasure coding, replication, and data scrubbing. Furthermore, many of these techniques try to be autonomous so that work is distributed across the cluster. For example, when placement groups change, the OSDs re-balance and re-shard data in the background in a process called placement group splitting.

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 MDS such that clients and daemons are synchronized on correct implementations without restarting.

| Service | Used to... | Malacology Interface |
|---|---|---|
| Mantle | load balance metadata<br>agree on current balancer<br>control client write lock<br>persist policy logic | Load Balancing<br>Versioning<br>Shared Resource<br>Durability |
| ZLog | implement CORFU<br>install ZLog interfaces<br>persist log entries | Logical Device<br>Versioning<br>Durability |
| ZLog Seqr. | load balance sequencers<br>guard shared resource | Load Balancing<br>Shared Resource |

**Table 3.** Mantle and ZLog are built using Malacology interfaces so they inherit the robustness of the subsystems of the underlying stack.

## 5. Services Built on Malacology

In this section we describe two services built on top of Malacology. The first is Mantle, 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 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. In [30] the authors showed how using Mantle, one can implement a single node metadata service, a distributed metadata service with hashing, and a distributed metadata service with dynamic subtree partitioning.

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 monitor daemons and (2) the durability and distribution provided by Ceph's reliable object store.

#### 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 ignored in the original implementation. The user had to set the version on each individual server and it was trivial to make the version inconsistent.

With Malacology, Mantle stores the version of the current load balancer in the Ceph map using the 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 consensus and consistency of Ceph's internal monitor daemons.

The user changes the version of the load balancer using a new CLI command:

```
fs set lua_balancer_class <version>
```

#### 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. The objects containing load balancer code are stored in a system-maintained group of objects (i.e. a pool). 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 results in inconsistencies if any MDS accidently modifies the code.

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 cancelled 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 only possible because we use Lua as the 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 the local log on each MDS. 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 their causality.

With Malacology, Mantle re-usues the centralized logging features of the MON subsystem. Important errors, warnings, and info messages are sent with the MON beacon 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 used discretly, so as not to spam the monitor with frivolous debugging but the large 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]. A shared-log is a general, yet powerful abstraction used to build many distributed systems, such as cloud-based metadata management [6] and elastic database storage engines [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 service, called a *sequencer*, for obtaining the entry at the tail of log. This metadata operation that avoids all persistent I/O in the common case.

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 an application-specific custom storage interface to discover the tail of the log, while simultaneously invalidating stale client requests using an epoch-based protocol.

We implement the sequencer service in Malacology as an application-specific file type that associates a small amount of metadata, including the 64-bit log tail position, to the inode state managed by the Ceph metadata service. Since a sequencer is associated with a unique log, this approach has the added benefit of allowing the metadata service to also handle naming, by representing each log service instance within the standard POSIX hierarchical namespace.

**Sequencer interface.** The sequencer resource supports the ability to *read()* the current tail value as well get the *next()* position in the log which also atomically increments the tail position. We implement this functionality as methods on ZLog inode resources. The primary challenge in mapping the sequencer resource to the metadata service is handling serialization correctly.

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 multiple clients that open a shared file to temporarily cache the inode resource, resulting in a round-robin, best-effort batching behavior. When a single client is accessing the sequencer resource it is able to cache state locally without interruption.

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 **??** 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 in 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 generic balancing policy that may strive 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 design. Clients independently map log positions that they have obtained from a 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 cached tail value during recovery of a sequencer, the maximum position in the log must be obtained. To do this, the storage interface exposes an additional *seal* method that atomically installs a new epoch value and returns the maximum log position that has been written. Since the recovery protocol does not inform clients of a new epoch value until the end of the process, the validity of the tail can be guaranteed despite the iterative process of contacting storage devices. 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 (and is inherited from) the Ceph metadata service.

## 6. Evaluation

We wanted to implement CORFU and we had two choices: build it from the ground up or do it on top of Malacology. To show the feasability of building on Malacology our evaluation focuses on the performance of the internal abstractions when used for to construct the Mantle and ZLog services. First, we benchmark scenarios where contention is at the sequencer by examining the intefaces used to map ZLog onto Malacology; specifically, we describe the sequencer implementation and the propogation of object & data interfaces interfaces. Next, we benchmark scenarios when there are multiple logs by using Mantle to (1) balance sequencers across a cluster and (2) recover from failure.

Detailed system-wide performance measurements and analysis is forthcoming for both Mantle and ZLog, as we try to find the best metadata balancing policies and physical design implementation trade-offs respectively. Since this work focuses on the programmabilty of Malacology, the goal of the benchmarks is to show that the components and subsystems that support the Malacology interfaces provide reasonable performance.

We have conducted our evaluation on CloudLab as well as our own 20 node cluster. We follow the “Popper Convention” [5] so as to try and make our experiments as reproducible as possible. We have a GitHub repository with our source code, experimental environment, configuration parameters, and workloads but due to the double-blind policy, we have temporarily disabled the links to our graphs and experimental details.

### 6.1 Mapping ZLog onto Malacology

We evaluate the mapping of ZLog onto Malacology by examining the costs of utilizing the metadata service as a sequencer, and propagating interface changes throughout the cluster using the monitoring service. We have omitted evaluation of the overhead costs of adding object interfaces themselves due to lack of space, and the fact that they are heavily tested and used already in production with acceptable performance, introducing a small constant overhead per operation but otherwise does not affect scalability.

### 6.1.1 Sequencer Implementation

We evaluated the performance of the ZLog Sequencer implemented in several different configurations by showing a trade-off between aggregate throughput achieved (shown in Figure 4), and latency distribution (shown in Figure 5).

In Figure 4 the highest throughput labeled $1C$ shows that a single client caching the sequencer resource may obtain new sequences at a rate of nearly 6 million positions per second. This is not surprising given that all state is cached locally in the client. When this resource is shared between two clients, the MDS uses a best-effort approach to granting the sequencer resource to each client (value $2C, BE$). This sharing has a large impact on performance, as shown by the nearly three orders of magnitude drop off in throughput.

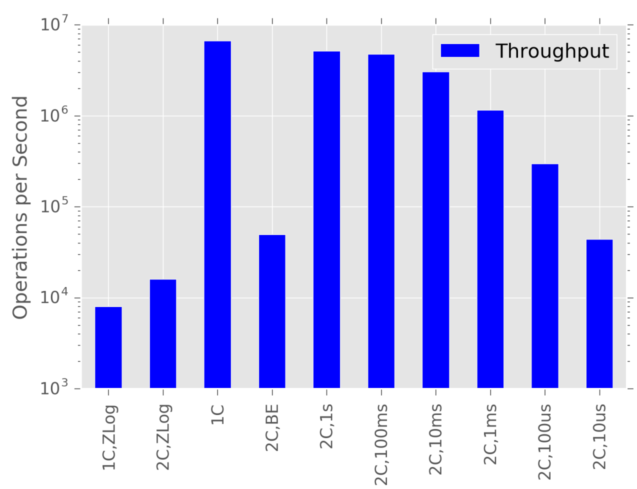


**Figure 4.** [source] Sequencer throughput by re-using various services. The highest performance is achieved using a single client with exclusive, cacheable privilege. Round-robin sharing of the sequencer resource is affected by the amount of time the resource is held, with best-effort performing the worst.

While a best-effort approach to sharing capabilities makes sense for a file system client, throughput drops significantly with two clients as the majority of the time neither client has the capability as it ping-pongs between clients. This observation reveals a tuning parameter: how long a client maintains its capability lease. In throughput labeled $2C, < time >$ we show aggregate throughput as we decrease the amount of time that a client holds its lease before responding to a request for the capability. At lease times as low as 10ms throughput achieve a significant proportion of the locally cached throughput.

**The Malacology Approach:** Re-use the capability management features of the MDS. While the batching mechanism can achieve high-throughput, it is not appropriate for

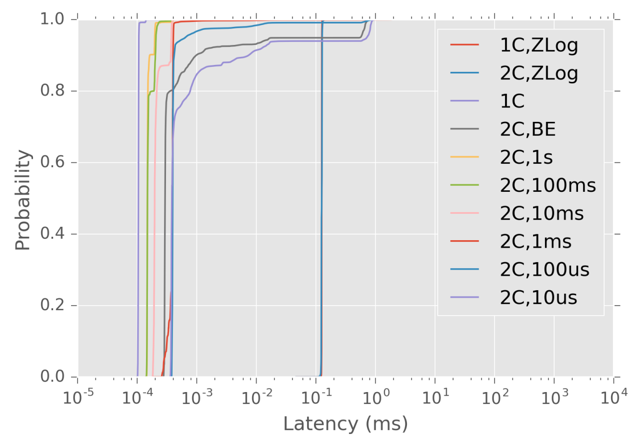


**Figure 5.** [source] The latency distribution shows that if the clients hold their capabilities longer, performance decreases. Changing the delay via policy can help the sequencer and clients strike a balance between throughput and latency.

applications that require predictable latency. The throughput in Figure 4 labeled $2C, ZLog$ corresponds to a centralized sequencer implementation. While throughput is lower for two clients (because of the synchronous network round-trip required for each request), Figure 5 shows that this approach achieves more predicable latency compared to all other approaches except for the single client locally cached mode.

### 6.1.2 Interface Propagation

In Section **??** we described an interface management service that is responsible for the durable and consistent cluster-wide distribution of interface versions. We evaluate the costs of such a service by embedding interface definitions in a Ceph cluster membership data structure that is managed by a Paxos instance, and distributed using a scalable, peer-to-peer gossip protocol that is built into Ceph. We assume an architecture that consists of a light-weight interface that enforces the semantics of interface management resulting in one or more interface updates that must be consistent across the cluster. The following experiments focus on the cost of achieving distribution of consistent a interface version.

Figure 6 shows the CDF of the latency of installing and distributing an interface update on all OSDs in the cluster. 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.

The first experiment shown in Figure 6 illustrates a lower bound cost for updates in a large cluster by avoiding disk I/O for storing interface updates locally within each OSD. In the experiment labeled ”120 OSD (RAM)” a cluster of 120 OSDs (10 OSDs x 12 servers) using an in-memory data

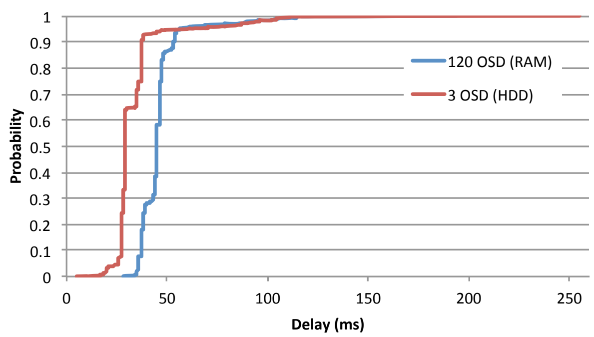


**Figure 6.** [source] Cluster-wide interface update latency, excluding the Paxos proposal cost for interface commit. Each of the 120 OSDs used a RAM-based object store, while the 3 OSDs used an HDD-backed object store.

store were deployed, showing a latency of less than 54 ms with a probability of 90% and a worst case latency of 194 ms. The curve labeled ”3 OSD (HDD)” is a smaller cluster where each of the three OSDs uses a spinning disk as the object store that saves a copy of the cluster map, and shows a latency of 37 ms with a 90% probability and a worst case latency of 255 ms. Since the cost of a disk I/O is larger than the same I/O executed against the in-memory device, the predominant cost in a lightly loaded cluster appears to be the number of OSDs.

**The Malacology Approach:** Re-use Ceph's Monitor functionality to propagate application-specific state that needs to be highly-available and strongly consistent. The real cost of interface management, in addition to cluster-wide propagation of interface updates, includes the network round-trip to the interface management service, the Paxos commit protocol, 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 ZLog and Mantle in a Multi-tenant Environment

For scenarios with multiple shared logs and many clients, co-locating sequencers on the same physical node is not ideal. Unfortunately, building a load balancer that can migrate the shared resource (in this case, the resource that guards the current token) would require building subsystems for migrating resources, monitoring the workloads, monitoring physical nodes, partitioning resources, and managing multiple sequencers.

The following experiments demonstrate the feasibility of using the mechanisms of the Malacology load balancing interface (e.g., migrating, monitoring, and partitioning load) to (1) alleviate load when multiple sequencers overwhelm a physical servers and (2) recover from failure.

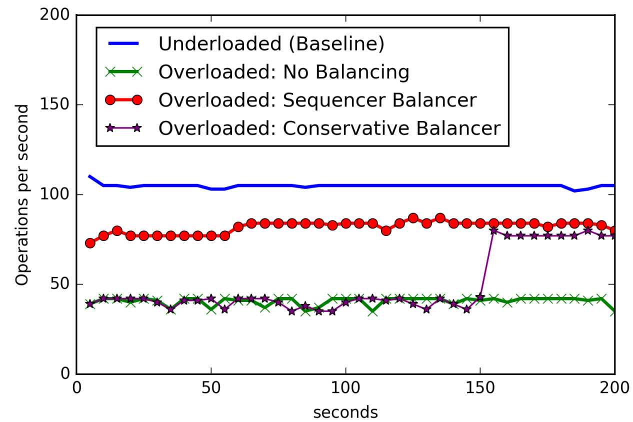


**Figure 7.** [source] The load balancing interface alleviates load for multiple sequencers in resource-constrained environments. “Underloaded (Baseline)” is the throughput of one sequencer running alone on a cluster; the other curves show how load balancing affects the throughput of one sequencer when sequencers for other logs are initially co-located on the same physical server (i.e. an ”Overloaded” environment).

### 6.2.1 Load Balancing Sequencers

We evaluate the load balancing interface by showing how the cluster of servers housing sequencers alleviates load. The experiments are run on a cluster with 10 nodes to store objects, one node to monitor the cluster, and 3 nodes that can accomodate sequencers. The workload has up to 6 logs each with 2 clients. Figure **??** shows the throughput per sequencer when all seqeuencers are co-located of one physical server. Two sequencers on the same server show no performance degradation but co-locating 3, 4, 5, or 6 sequencers leads to a 2x, 3x, 4x, and 5x slowdown, respectively. For the rest of the experiments we load balance 3, 4, 5, or 6 sequencers across the cluxter because this is hte point at which a single physical server gets overloaded.

Figure **??** shows the performance profiles of the sequencers as they migrate around the cluster. Each graph in that figure shows different “waves” of sequencers, where a “wave” is a burst of clients trying to append to the end of the log. These waves are examples of hotspots and flash crowds; they are dynamic and resource constrained because clients access a shared resource. Wave 1, policy 1 shows how the sequencers initially get distributed poorly. From x to x seconds, all seqeuencers are co-located. At x seconds, 3 of the sequencers (1, 2, 3) migrate off the first server and onto the second one; this is what we want.

We use a balancer policy specifically designed for multiple sequencers. The balancer aggressively sheds half its load to the next server as soon as it can. It is based off the “Greedy Spill Even” approach in the Mantle [30] paper but is reproduced in Figure 10 for the reader's convenience. Each line

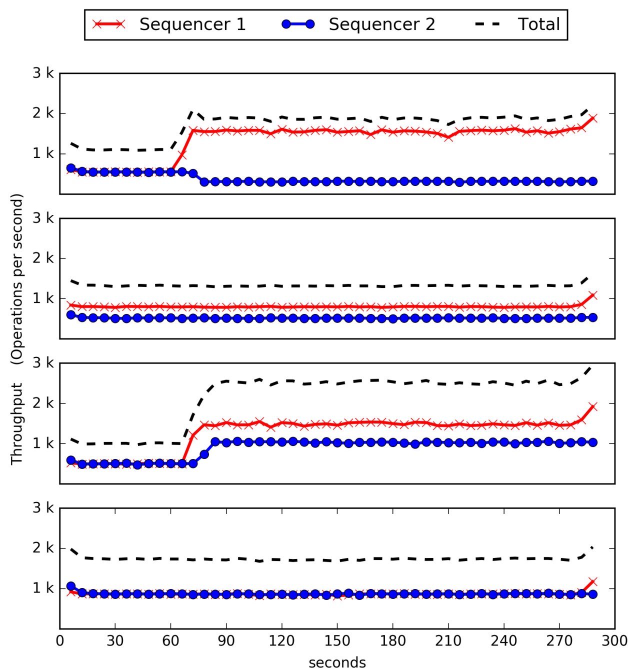


**Figure 8.** [source] The load balancing interface alleviates load for multiple sequencers in resource-constrained environments. "Underloaded (Baseline)" is the throughput of one sequencer running alone on a cluster; the other curves show how load balancing affects the throughput of one sequencer when sequencers for other logs are initially co-located on the same physical server (i.e. an "Overloaded" environment).

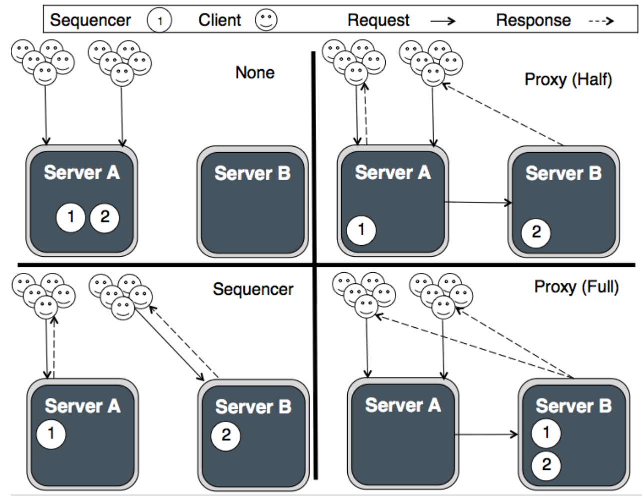


**Figure 9.** [source] The load balancing interface alleviates load for multiple sequencers in resource-constrained environments. "Underloaded (Baseline)" is the throughput of one sequencer running alone on a cluster; the other curves show how load balancing affects the throughput of one sequencer when sequencers for other logs are initially co-located on the same physical server (i.e. an "Overloaded" environment).

```
-- Server load
load = server[i]["req_rate"]

-- When policy
if server[whoami]["load"]>.01 and
   server[whoami+1]["load"]<.01 then

-- Where policy
targets[whoami+1]=load/2
```

**Figure 10.** The sequencer balancing policy that works best for bursty workloads.

of code uses variables exposed by the Mantle API and is executed as a callback by Mantle. "Server load" lets the administrator control what metric constitutes load on a server; "i" is a variable used by Mantle to iterate over servers and "req_rate" is a metric based on the exponenential decay of the request rate. "When policy" is used by Mantle to set a conditional for kicking off migration; in this case, the cluster migrates load if the current server, "whoami", has load (`>0.1`) and the next server (physical servers have "ranks") has a small amount load (`<0.1`). Finally, the "where policy" is used by Mantle to determine how much load to shed. The policy fills in a "targets" array to tell the migrator how much load to send to each server; in this case, we shed half of the current load to the next server. The unit of load is a single sequencer for a ZLog log.

Figure 9 shows a performance profile of one of the sequencers as the load balancing interface migrates sequencers around the cluster to alleviate load. Each curve shows the throughput ($y$ axis) over time ($x$ axis). "Underloaded (Baseline)" is the performance of one sequencer on one physical server serving 7 clients - there are no other sequencers or logs in the system and the curve represents the maximum throughput of a sequencer on an underutilized physical server. "Overloaded: No Balancing" adds a second sequencer of equal load to the same physical server with the intent of overloading the single physical server; recall that the throughput for this second sequencer is omitted. Compared to the baseline, the overloaded physical server with 2 sequencers has more than a 2× throughput degradation. The stability of throughput curves (i.e. the variation on the $y$ axis) indicate which sequencers are on overloaded physical servers. "Underloaded (Baseline)" is the best performance for this particular workload because there is no contention. Performance is stable and the MDS is under-loaded because we do not see the spike at the end like the other curves. "Overloaded: No Balancing" shows an overloaded sequencer because the performance of the monitored sequencer is low and unstable (high variation in performance).

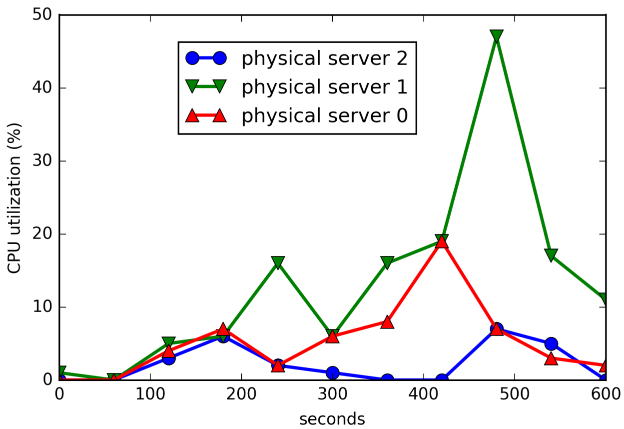


**Figure 11.** [source] The CPU utilization for a sequencer in an environment with the "Conservative Policy" load balancer shows that the load balancing service is not multi-threaded. The load balancing service is designed this way because the performance for a guarded resource will not improve with more cores.

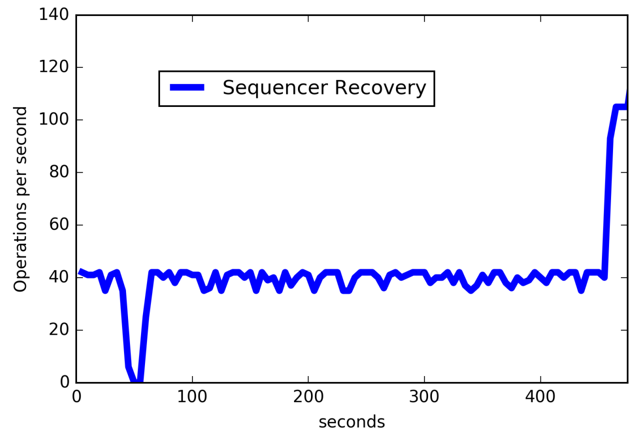


**Figure 12.** The sequencer using Mantle also inherits all the recovery protocols of the Ceph metadata cluster. The throughput of the sequencer drops when we terminal MDS 0. The performance quickly resumes and the sequencer is only down for 15 seconds.

The last two curves of Figure 9 show the performance improvements when sequencers are load balanced across the cluster. The load balancing policies differ by the threshold at which they start balancing: "Overloaded: Sequencer Policy" starts balancing immediately and "Overloaded: Conservative Policy" waits for 120 seconds before migrating sequencers.

Figure 11 shows how load balancing effects the physical servers by plotting the CPU utilization average over 1 minute ($y$ axis) over time ($x$ axis). It shows the "Conservative Policy" load balancer from the run in Figure 9, so recall that resources are constrained during the job by co-locating a second sequencer on the same physical server. CPU utilization spikes when physical servers receive load (at 2 minutes physical server 1 receives a sequencer from physical server 0) but the CPU utilization never eclipses 50%. The utilization is low because the load balancer interface is not multi-threaded. Recall the the load balancing interface re-uses the POSIX metadata cluster migration mechanisms; serving file system metadata is single-threaded because it has been shown that metadata performance does not improve with multiple threads [16].

**The Malacology Approach:** Re-use the Ceph metadata migration, load balancing, cache coherence, and capabilities subsystems of the metadata cluster. Applying Mantle's load balancing policy shows 2.02× performance improvement and only a 16% degradation in performance over the baseline, even though the baseline has half as many clients.

### 6.2.2 Recovery

This experiment shows how a sequencer handing out tokens can use the Ceph MDS recovery protocols to recover. For this workload, two sequencers are running on a metadata cluster *without* multiple active MDS nodes. In this cluster, MDS 1 and MDS 2 are in "standby" mode, meaning one of them will step in should the head MDS 0 fail. At the 60 second mark, we manually kill MDS 0.

Figure 12 shows the sequencer throughput ($y$ axis) over time ($x$ axis). When MDS 0 dies, the MON cluster notices and quickly assigns a new MDS to handle the namespace. This is when MDS 1 takes over the sequencer duties and starts handing out tokens.

**The Malacology Approach:** Re-use the Ceph metadata protocols for failure detection and recovery. The sequencer recovers in 15 seconds on a new MDS and it maintains the same throughput that it had before the failure.

## 7. 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 resourses 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 [26]). This notion of control/data separation was also applied in software-defined storage (SDS) [4, 33, 35]. Similarly, IOStack [20] is providing policy-based provisioning and filtering in OpenStack Swift.

Another view of programmable storage systems is one of tailoring systems resources to applications [4]. Related work includes work around the Exokernel [17] and SPIN [11] and Vino [29], 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 [4, 28, 31]).

Malacology uses the same Active and Typed Storage module presented in DataMods [38]; 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.

## 8. Conclusion and Future Work

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

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

## References


[1] Zerovm and openstack swift.

[2] Apache Parquet Contributors. Parquet Columnar Storage Format, 2014.

[3] 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.

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

[5] Anonymous Author. Omitted to blind submission. *Anonymous*, 1900.

[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] William J. Bolosky.

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

[14] 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.

[15] Ceph Contributors. Ceph Documentation, 2010.

[16] Konstantinos Chasapis, Manuel F. Dolz, Michael Kuhn, and Thomas Ludwig. Evaluating Lustre's Metadata Server on a Multi-socket Platform. In *Proceedings of the 9th Parallel Data Storage Workshop*, PDSW '14, 2014.

[17] 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.

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

[19] 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.

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

[21] 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.

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

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

[24] Ajay Gulati, Ganesha Shanmuganathan, Anne Holler, and Irfan Ahmad. Cloud-scale resource management: challenges and techniques. In *Proceedings of the 3rd USENIX conference on Hot topics in cloud computing*, HotCloud'11, pages 3–3, Berkeley, CA, USA, 2011. USENIX Association.

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

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

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

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

[30] 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.

[31] 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.

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

[33] 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.

[34] Ceph Team. Ceph's new monitor changes.

[35] 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.

[36] 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.

[37] Noah Watkins.

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

[39] 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.

[40] 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.

[41] 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.