Malacology: A Programmable Storage System Built on Ceph

Abstract

Storage systems are caught between rapidly changing data processing systems and the increasing speed of storage devices. This puts tremendous pressure on storage systems to adapt 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 unprecedented flexibility for storage systems to evolve without sacrificing the robustness of its code-hardened subsystems. We illustrate the advantages and challenges of programmability by constructing two services out of existing abstractions: a file system metadata load balancer and a high-performance distributed shared-log that leverages flash devices. RESULTS

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 can dominate overall performance if not used carefully. Over the last few decades members of the storage systems community have developed ingenious strategies to meet correctness requirements while somewhat hiding the latency of traditional storage media. 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.

A number of recent developments are disrupting traditional storage systems: (1) 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; (2) demand for managing structured data and flexible consistency semantics at scale pressure big data processing systems to use storage abstractions that can meet these demands; and (3) 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.

These three trends put evolutionary pressure on storage systems and raise the question whether there are principles that storage systems designers can follow to evolve storage systems efficiently and without jeopardizing years of code-hardening and performance optimization efforts that are important for users to continue to entrust their data to the storage system.

In this paper we investigate an approach that focusses on generalizing existing storage system resources, services, and abstractions that in generalized form can be used to *program* new services. By doing so one can reuse subsystems and their optimizations and leverage their established correctness, robustness, and efficiency. We will refer to this programmability as *programmable storage*, which differs from *active storage* (the injection and execution of arbitrary code in a storage system) and *software-defined storage* (the control of thin-provisioning of storage).

To illustrate the benefits and challenges of this approach we examine the programmability of Ceph [1], the increasingly popular, production-level open-source distributed storage system. Something of a storage swiss army knife, Ceph supports file, block, and object interfaces simultaneously in a single cluster. By introducing programmability concepts into Ceph, we can build new services by carefully exposing internal storage services to applications.

For example, Ceph [1] contains a cluster of monitoring nodes (MONs) that use PAXOS to maintain a consistent view of system-wide metadata such as data distribution parameters and cluster membership. The RADOS object storage system is a cluster of storage devices (OSDs) that provide Ceph with data durability and integrity using replication, erasure-coding, and scrubbing [2]. The data dis-

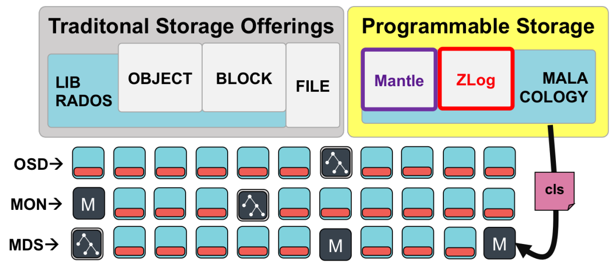


Figure 1: A Ceph installation consists of a cluster running mostly storage daemons (OSDs) to ensure data durability, a few monitor daemons (MONs) to reach consensus of the cluster state and to ensure its consistent versioning, and a few metadata service daemons (MDSs) for scalable metadata service. Ceph's APIs include object, block, file abstractions. Malacology enables the programmability of these abstractions and services. To illustrate Malacology we implement two services on Ceph, Zlog and Mantle, by programming Ceph's durability, consistent versioning, consensus, and metadata subsystems.

trubiton metadata maintained by the cluster monitors is used by RADOS servers and clients to provide consistent reads and writes, and both RADOS and the monitoring services are used by a cluster of file system metadata servers that provide POSIX semantics using sophisticated indexing and distributed locking services. We contend that re-using and re-purposing these code-hardened subsystems is paramount to successfully adapting storage systems to new APIs and new storage devices without losing the benefits from years of code-hardening work.

To illustrate the benefits and challenges of this approach we have designed and evaluated Malacology, a programmable storage system capable of incorporating new functionality and re-purposing existing subsystems of Ceph. We build the framework on Ceph by leveraging powerful subsystem abstractions used by the monitor daemons (MONs), object storage daemons (OSDs), and metadata server daemons (MDSs). As shown in Figure 1, this framework is expressive enough to provide the functionality necessary for implementing new services. Our contributions are:

- a programmable storage system implementation that re-uses and extends existing abstractions. It includes:
    1. inode abstractions for sequencer services
    2. consensus abstractions for policy versioning
    3. storage objects for persisting policies
    4. interface classes for new logical storage/metadata devices
- example systems that use this framework
    1. shared log service based on CORFU [3]
    2. metadata load balancer based on Mantle [4]

First, we describe existing techniques for bridging the gap between the performance of storage systems and the requirements of applications (§2). Next, we discuss the "distributed systems" inspired abstractions employed by Ceph (§3) and allude to their re-usability as components in other services. The next section enumerates the abstractions that Malacology exposes in the implementation section (§4). We then show how we use Malacology to synthesize entirely new storage services on an existing system through configuration and small changes (§5). We conclude by evaluating real-world use cases (§6).

2 Highly Tailored and Application-Specifc Storage Systems

A consequence of complicated data management frameworks and faster devices is that the storage system cannot meet the needs of the general-purpose storage system. Workarounds for helping the application meet its performance goals roughly fall into one of three categories: "bolt-on" services, application changes, and storage changes.

2.1 "Bolt-on" services

So called "bolt-on" services are 3rd party systems that are integrated into the system to provide functionality and improve performance for common use-cases (e.g. a metadata service). In addition to sacrificing simplicity, "bolt-on" services often duplicate functionality and execute redundant code, unnecessarily increasing the likelihood of bugs or worse, introducing unpredictable performance problems. For example, we are developing a distributed shared-commit log on Ceph called ZLog. ZLog uses a "sequencer" to distribute tokens to clients that want to append to the end of the log. When designing the sequencer, we considered bolting on Zookeeper [] as a service. Zookeeper would satisfy our requirements by proiding a network and transaction stack but it would perform poorly because Zookeeper persists its data and the sequencer can be in-memory and volatile. A Zookeeper sequencer would adhere to the preconceived notions of storage, namely that data in storage systems should be safe from data loss, but would introduce unecessary scalability limits [**?**].

2.2 Application Changes

The second approach to adapting to a storage system deficiency is to extend the responsibility of the application. This means changing the application itself by adding more data management intelligence or domain-specific middleware (e.g., a new data layout). For instance, an application may change itself to exploit data locality or I/O parallelism in a distributed storage system.

For example, SciHadoop [5,6] changes both the Hadoop application and the Hadoop framework itself to leverage the structure of scientific (3D array-based data, more specifically) to increase performance, locality, and the number of early results. This is not a bad proposition, but creates a coupling that is highly tied to the underlying physical properties of the system, making it difficult to adapt to future changes at the storage system level.

2.3 Storage Changes

When these two approaches fail to meet the needs of the application, developers turn their attention to the storage system itself. Traditionally, storage system behaviour can be altered using two techniques: tuning the system or introducing changes to the system. The difficulty of both of these approaches has given rise to a third technique called active storage.

2.3.1 Tuning

Tuning large and complicated systems is difficult because of the number of parameters (e.g., Hadoop 2.7.1 has 210 tunables) and because the tunables are difficult to understand or quantify (e.g., Ceph tunables [4]). To succesfully tune the system, the developer must have domain and system specific knowledge. Without this intimate knowledge, the tuning turns into somewhat of a black art, where the only way to figure out the best settings is trial and error.

Auto-tuning techniques attempt to find a good solution among a huge space of available system configurations. However, in practice auto-tuning is limited to only the configuration tunables that the storage system exposes (e.g. block size) and can be overwhelmed with too many parameters. Starfish [7] made an attempt of this for Hadoop but this technique would need to severely limit the space of parameters in order to feasible, similar to the approach used in [8]. For instance, auto-tuning may be capable of identifying instances in which new data layouts would benefit a workload, but unless the system can provide such a transformation, the option is left off the table.

2.3.2 Software Changes

As a last resort, the application developer can introduce changes to the storage system itself. This endeavour is especially difficult because the developer must first familiarate themselves with the storage system, and even if they manage to make the necessary changes, they must get their changes upstream for that specific project or become mantainer for their in-house version. Being a maintainer means they re-base their versions against new versions of the storage system and address any bugs for their branched code. A successful example of this is the work that focused on HDFS scalability for metadata-intensive workloads [9]. This has lead to modifications to its architecture or API [10] to improve performance.

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

Category	Specialization	#
Locking	Shared 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

2.3.3 Hybrid Approaches

As an alternative, some systems allow developers to extend the interfaces of the storage devices to perform arbitrary operations on data stored on disk. In Ceph, object interfaces are C/C++ plugins that add new functionality to the OSDs. As an example use-case, ZLog needs to store metadata information with each object so for each operation, it checks the epoch value and write capability (2 metadata reads) and writes the index (metadata write). Figure 1 shows the throughput (y axis) over time (x axis) of two OSD implementations for storing metadata: in the header of the byte stream (data) vs. in the object extended attributes (XATTR). The speed for appending data without any metadata operations (data raw) is also shown as a baseline for comparison. For append-heavy workloads storing metadata as a header in the data performs about 1.5x better than storing metadata as an extended attribute.

Object interface classese are used in production Ceph environments; Figure 3 shows a dramatic growth in the use of co-designed interfaces in the Ceph community since 2010. Figure 4 examines this growth in interfaces further by showing the lines of code that are changed (y axis) over time (x axis). The prevalence of blue dots indicates that users frequently update their interfaces close to release cycles and that massive changes come in bunches. This

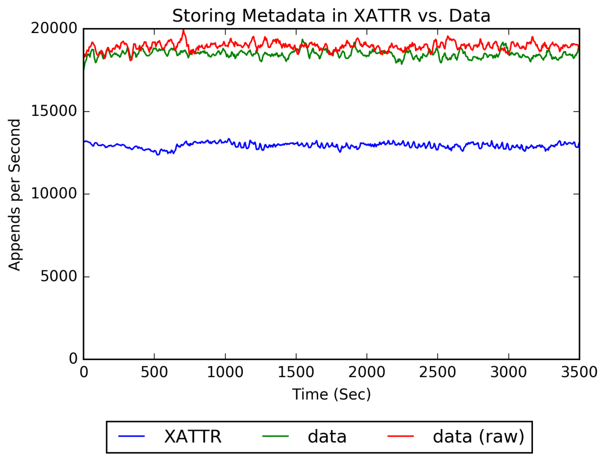


Figure 2: [source] When appending data to objects, the object class that stores metadata in a header in the data byte stream (data) performs 1.5x better than the object class that stores metdata in the extended attributes of the object (XATTR); it is almost as fast as appending data without updating metadata (data raw).

interface churn reflects both the popularity of object interfaces and the complexity of the processing being done by the OSDs. What is most remarkable about Figure 4 is that this trend contradicts the notion that API changes are a burden for users. The types of object interfaces, which are enumerated in Table 1, show that this active storage development is present throughough the Ceph software stack.

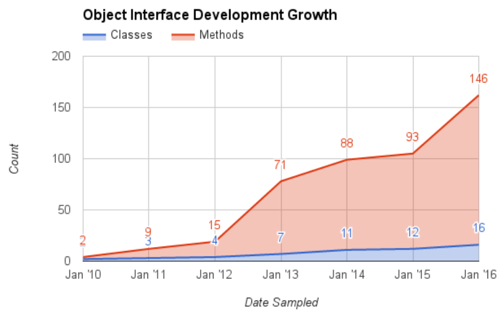


Figure 3: Since 2010, the growth in the number of co-designed object storage interfaces in Ceph has been accelerating. This plot is the number of object classes (a group of interfaces), and the total number of methods (the actual API end-points).

The popularity of the active storage component of Ceph hint at three trends in the Ceph community: (1) increasingly, the default algorithms/tunables of the storage system are insufficient for the application's performance goals, (2) programmers are becoming more aware of their application's behavior, and (3) programmers know how to manage

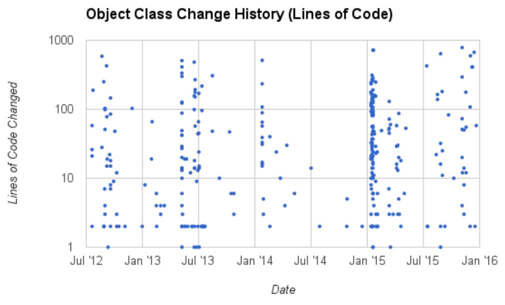


Figure 4: Source code changes over time indicate the dynamic interface development will need to support a high frequency of interface churn. Each dot represents a Ceph commit with a corresponding number of lines of code changed.

resources to improve performance. Programmers gravitate towards object interfaces because it gives them 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 object interfaces, had no way to tell the storage system how to handle their data.

These implementations use the active storage interfaces in Ceph. We refer to this as *storage programmability* which is a method by which an application communicates its requirements to the storage system in a way that allows the application to realize a new behavior without sacrificing the correctness of the underlying system. This evaluation and resulting implementation would have been a burden without object classes.

3 Re-Usable Components in Ceph

Ceph is a production-quality distributed system and in this section we touch on some of the subsystems it uses to provide a general-purpose storage system. In the next section, we describe how we leverage these subsystems to build new services.

3.1 Durability

Ceph provides storage by striping and replicating data across RADOS [2], 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 rebalance and re-shard data in the background in a process called placement group splitting.

3.2 Consistency/Versioning of Cluster State

Ceph needs to keep track of cluster state and it does this with separate "services": client authentication, logging, MDS maps, MON maps, OSD maps and placement group (PG) maps. These services are all managed in the MONs and they all talk to a PAXOS instance; this PAXOS instance goes and talks to other PAXOS instances on other MONs so that they can all agree on the correct version of the service; essentially they reach an agreement about what the state of the cluster is.

3.3 Active Storage

Active storage attempts to push computation closer to the data by exposing interfaces to execute code remotely. For example, the storage server could process data before sending data over the network or sorting it. The active storage component in Ceph is called "object interface classes" (or `cls` for short). The Ceph object interface classes can be loaded at runtime and customized with user-defined functionality. The basic idea is shown in Figure~**??**, where the `libcls_md5.so` shared library performs the MD5 hash on an object at the oSD instead of transferring data over the network. This ability to carry out arbitrary operations on objects stored on OSDs helps applications improve performance by optimizing things like network round trips, data movement, and remote resources which may be idle.

Object storage interfaces are compiled into shared libraries and loaded into a running OSD daemon using `dlopen()`. Because the shared library has to link against symbols found in the running executable, the interfaces are stored on the local file systems of each OSD so they can be versioned and distributed alongside the same Ceph binaries. This policy is for safety, since the OSDs could be on different distros that provide different versioning capabilities. Also, this approach allows bug fixes to co-evolve with the rest of the Ceph installation.

Although customizable OSD interfaces are powerful, the current implementation has drawbacks. OSD interfaces are written in C/C++ and compiled into a shared library, so the developer must account for different target architectures. Second, C/C++ provides more functionality than is necessary since the snippets of interface code mainly set policies and perform simple operations. Finally, developing these interfaces in C/C++ has high overhead. The developer must learn how the OSD dynamically loads the shared libraries (*e.g.*, OSDs rely on a strict naming convention to find shared libraries), how to get their interfaces compiled using the Ceph `make` system, and how to debug issues that are not related to their specific interface (*e.g.*, the OSD cannot find the shared library) - this learning curve is unacceptable for non-Ceph developers, especially since most interfaces are one-off solutions specific to their applications.

Ceph has a whole infrastructure for getting dynamic code into the OSD. The `ClassHandler` class safely opens object interfaces, executes commands defined by the shared library, and fails gracefully with helpful errors if anything goes wrong. Instead of injecting strings directly into Ceph (like we did with the original Mantle implementation), the `ClassHandler` class safely opens shared code, even code with dependencies that are not in Ceph itself (e.g., Lua). With the `ClassHandler`, we get the safety and robustness of loading dynamic code, the ease of transferring state between object interfaces and the oSD internals, integration with testing and correctness suites, and `structs` for interface data and handlers.

While we consider active storage to be an excellent example of programmability, what separates our proposal from previous work is the observation that so much *more* of the storage system can be reused to construct advanced, domain-specific interfaces.

4 Malacology Implementation

To enable the programmability of Ceph subsystems, Malacology relies heavily on interface classes for introducing new functionality into the MDSs and OSDs. Malacology introduces 3 components to manage object/metadata interfaces:

1. additional interface hooks and classes for OSDs and MDSs using the existing Ceph `cls` infrastructure
2. durable interface classes using the Ceph object store
3. versioned and consistent interfaces using Ceph's consensus service

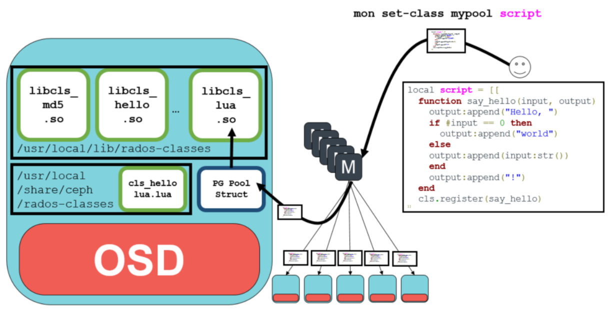
In the following section, we will discuss the existing Ceph infrastructure and the changes necessary for Malacology. Our focus will be to demonstrate the viability and usefulness of the Malacology appraoch with example implementations that due to limited time for the submission of this paper are not necessarily fully optimized.

4.1 Additional Interface Hooks and Classes

Malacology adds new hooks to Ceph and a new interface class for both the MDS and OSD that uses Lua [11][12].

Our framework has added a mechanism for defining and running object and metadata balancer classes using Lua. Our Lua bindings expose functions and symbols both ways; the host program can call functions defined in Lua and the Lua scripts can call functions defined in native C++. These bindings are merged upstream. We choose Lua for 4 reasons: performance, portability, size, and and security.

Lua is a fast scripting language. It was designed to be an embedded language and the LuaJIT virtual machines boasts near-native performance [13][14]. Lua is frequently used in game engines to set policies but we use it here because most of the user-defined object classes in Ceph are policies as well (see §1). Separating policy from mechanism is a driving factor in using Lua.



```
mon set-class mypool script
```

```
local script = [[
  function say_hello(input, output)
    output:append("Hello, ")
    if #input == 0 then
      output:append("world")
    else
    output:append(input:str())
    end
    output:append("!")
  end
  cls.register(say_hello)
]]
```

Figure 5: Clients add functionality by injecting Lua scripts with a new monitor (M) command. After the "proposal" (using PAXOS terminology) is accepted by the cluster of monitors, the script is redundantly and consistently distributed to OSDs.

For object interfaces, clients send Lua classes to the OSD and then they can invoke operations in that class remotely on the OSD. In reality, this feature is a statically loaded object class written in C/C++ that runs dynamically defined object interfaces, where clients access the classes using an `exec()` wrapper. Although sending the class with each client request is stateless (which has its own set of nice features), it is costly for network bandwidth and difficult to organize at the application level. Also, while the execute wrapper simplifies the implementation, it burdens the applications, since they need to be recompiled to use the `exec()` function.

Our framework has a mechanism for defining and running object and metadata balancer classes with scripting. Scripting is useful in large systems for 3 reasons: - accelerates deployment: tweaking policies, like changing a threshold or metric calculation, does not require a full recompile or system start-up. - transparency: if the bindings carefully expose the correct metrics and functions, the user does not need to learn the internal functions, variables, or types of the system. - portability: scripts can be exchanged across platforms and system versions without compatibility issues.

This effectively decouples policy from mechanism, which helps future designers explore trade-offs and isolates policy development from code-hardened systems. Decoupling policy mechanism is not a new technique but designing the storage system with programmability as a first-class citizen is credits policy/mechanism separation as the biggest reason for the success of the block allocation mechanisms in the Fast File System~[**?**] yet there are no mechanisms for transparently exposing and modifying the underlying logic in modern file systems.

Malacology has Lua bindings that expose functions and symbols both ways; the host program can call functions defined in Lua and the Lua scripts can call functions defined in native C++. These bindings are merged upstream. We choose Lua for 4 reasons: performance, portability, size, and security.

4.1.1 Using Lua

Lua is a performs well as an embeddable scripting language. Although it is an interpreted language, the bindings are designed to make the exchange of variables and functions amongst languages straightforward. The LuaJIT virtual machine boasts near-native performance, making it a logical choice for scriptability frameworks in systems research [15]. In storage systems, it has been effectively used both on [16,17] and off [4] the critical path, where performance is important. Lua is frequently used in game engines to set policies but we use it here because most of the user-defined classes in Ceph are policies as well! We do not want to provide specific implementations, like pulling data from objects or transferring them over the network, but instead strive to say *what to do* with the data once we have it. Separating policy from mechanism is a driving factor in using Lua.

Because Lua is interpreted, it is also portable. The portability allows functions to be shipped around the cluster without having to compile or reason about compatibility issues. Traditionally, intefaces are written in C/C++, so they must be compiled into shared libraries before being injected into the daemon at runtime. Shared libraries need to be compiled on the host they will be run on to accomodate different architectures and dependendencies. Embedding the Lua interpretator, on the other hand, gives us the flexbility to store the actual Lua code in other places, like RADOS, so we can enjoy all the other properties provided by that particular back-end.

The other two advantages, that Lua leaves a small memory footprint and that is is secure, are discussed at length in [15,18]. In short, Lua provides relatively robust sandbox capabilities for protecting the system against some malicious cases, but does not protect the system from poor policies. Since Malacology does not use these features (yet), we will avoid discussion here.

4.1.2 Generalizing the Lua VM

We generalize the Lua VM since it will be used in both the OSD and MDS. We put the core sandbox wrapper for the Lua object interface in a common directory in Ceph and link against it. This core Lua wrapper contains just the dependencies, symbols, and functions, need to run the Lua VM. Some of these functions include `clslua_log()` for transferring logs to the daemon logs and `clslua_pcall()` for calling Lua functions.

Dependencies, symbols, and functions specific to the object or balancer interface are put in the `cls` or `bal` directories, respectively. For example, the Lua object interface uses functions that have placement group filters, cryptography functions, and object metadata (*it i.e.* `cxx_omap_getvals()`, object data, object extended attributes, and object versions – all these are dependencies, symbols, and functions are part of the OSD process but not the MDS process. As a result, we put interface code that uses these in `cls` directory. With this scheme, The OSD will `dlopen()` the shared library created by Lua object interface shared library while the MDS will `dlopen()` the Lua balancer interface shared library; both shared libraries will the Lua core.

Both the object and balancer interfaces define functions and attach them to the core Lua sandbox. For example, the Lua object storage interface class attaches data, extended attribute, object map, and object version functions, to the the Lua core functions; the exact functions are shown in Listing~**??**. The advantages of this is that we avoid duplicating code, we provide a framework for putting Lua code in other parts of the system, and we remove components and APIs that are too integrated with the OSD.

4.1.3 Generalizing the Class Handler

Our framework can also load Lua interfaces from the local file system – the same technique as the C/C++ object interfaces. First, the OSD looks for C/C++ shared library. If the OSD cannot find the file, it looks for a Lua script of the same name. The script is read into a string of the C++ object class; this string is later forwarded to the native Lua class. When a requests comes in, the method name is passed with the `exec()` function and the corresponding function in the Lua class is called. Now clients need to only send their Lua object class once and the OSDs will store them locally. Also, clients can execute any Lua handler they want.

4.2 Storing Interfaces in RADOS

For balancer interfaces, clients put balancers into RADOS and the CephFS metadata balancer invokes the operations in the user-defined class remotely on the MDS.

[7:31] Nothing would stop C++ from being stashed in objects and recompiled on whatever platform they are being loaded on dynamically (like a DB compiles each query). There just hasn't been a need for such a feature. The next blog post, which I didn't at all allude to in the one you are reading, shows how the Lua scripts can be put into the OSD map and then monitors manage and version the script, distributing them to each OSD. Stored in objects vs stored in the OSD map is a debate, I guess. I am not sure which makes more sense.

4.3 Versioning Interfaces with MONs

We added a command, `ceph osd pool set-class <pool> <class> <script>`, that the user uses to inject the Lua object interface into the cluster. By leveraging the monitor daemons, we get the consistency from the monitors' version management, the distribution from the monitors' data structures (which are already distributed), and the durability from the robustness of the monitors and persistence with Paxos. The implementation uses the placement group pool data structure which maintains the policies (*e.g.*, erasure coding) and organization details (*e.g.*, snapshots) for each pool. While stuffing the information into the monitor map, the structure that describes the the monitor topology, was an option, the placement group pool allows finer grainer control over different typoes of data. In regards to the consistency, each time the user injects a new Lua object class it enters the monitor quorum as a proposal; updates to the placement group pool need to wait until accepted, at which point the state will be propogated and versioned. We hooked up the monitor command to the `cls_lua` class by having the OSD pull the script from the placement group instead of trying to dynamically load the shared library with `dlopen()` or the Lua script.

Advantages:

- lets us store/manage interfaces
- does most of the work (versioning, consistency, durabiltiy) for us
- gives a new abstraction for dealing/mutating interfaces

The cls-lua branch lets users write object classes in Lua, the lua-rados client library lets users write applications that talk to RADOS with Lua, and the cls-client uses the lua-rados library to send Lua object classes to the OSDs equipped with the LuaJIT VM.

Cap Owner	Failure	ZLog Recovery
mds	mds	mds
client	mds	xfer client cap
mds	client	n/a
client	client	mds

Table 2: Failure and recovery scenarios.

4.4 Specifying the Lua Class

Maintain versions and consistency

- cls-lua branch: write object classes in Lua
- lua-rados: talk to RADOS with Lua
- cls-client: use lua-rados + cls-lua branch to send Lua object classes to OSDs equipped with LuaJIT VM

5 Services Built on Malacology

5.1 Mantle: A Programmable Metadata Load Balancer

Many distributed file systems decouple metadata and data I/O so that these services can scale independently [1,19–23]. Despite this optimization, scaling the metadata services is still difficult because metadata accesses impose small and frequent requests on the underlying storage syste [24]. Many techniques for designing the metadata services have been proposed to accommodate this workload: Lustre [25], GFS [20], and HDFS [23] keep all metadata on one server; GIGA+ [26], IndexFS [27], Lazy Hybrid [28], GPFS [29], and pNFS [30] hash the file system namespace across a dedicated cluster and cache metadata; Panasas [22] and CephFS [31] partition the file system namespace into # d assign them to servers. These systems have novel mechanisms for Mantle [4] is a programmable metadata balancer that separates the metadata balancing policies from their mechanisms. Admistrators inject code to change how the metadata cluster distributes metadata. In the paper, we showed how to implement a single node metadata service, a distributed metadata services with hashing, and a distributed metadata service with dynamic subtree partitioning.

The Ceph team wants to merge Mantle because the scriptability is useful for debugging, controlling the metadata balancer, and examining trade-offs for different balancers. Unfortunately, this research quality system is not as robust as Ceph and the Ceph team wants more safety, durability, and consistency for the new functionality. For example, the original Mantle API is fragile: the API is not enforced by the runtime, the system allows injectable strings through the admin daemon, constructing the Lua script in C++ is clunky, and the administrator can inject really bad policies (e.g., while 1) that brings the whole system down.

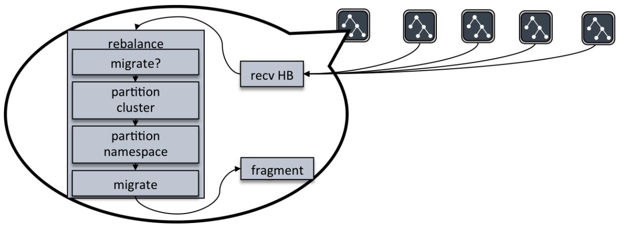


Figure 6: This is Mantle.

5.2 ZLog: A Distributed Shared Commit Log

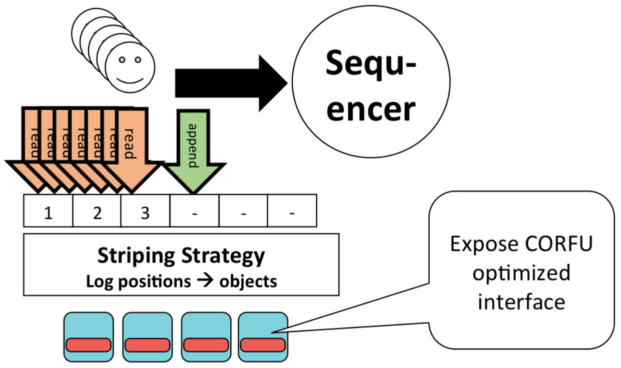


Figure 7: This is ZLog.

5.2.1 Consistency

5.2.2 Striping Strategies

5.2.3 Tiering

Malacology uses the same Active and Typed Storage module presented in DataMods [32]; 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.

5.3 MDS as a sequencer

5.3.1 Sequencer recovery with MDS

5.3.2 MDS Policy for Revoking Capability

The MDS has a queue of locks that it uses to revoke capabilities from the clients. If we put a Mantle-style hook here, we can control when those revoke messages go out.

5.3.3 Mantle Balancing Policies to Migrate Sequencers

6 Evaluation

1 Experiments: 2 3 1. mdtest separate directories 4 - just based on default traffic 5 - kill one of the MDSs 6 7 2. filebench profiles 8 - adaptable vs. greedy vs. fill&spill 9 10 3. sequencer balancer 11 - just based on default traffic 12 13 3. OpenSSL compile ~

6.1 Mantle

6.1.1 Experiment: FileBench

6.1.2 Experiment: OpenSSL Compile

6.2 MDS as ZLog Sequencer

6.2.1 Experiment: Object Class Programmability

Add quote from CORFU paper that talks about what they had to hack

6.2.2 Experiment: Multi-Client Burstiness

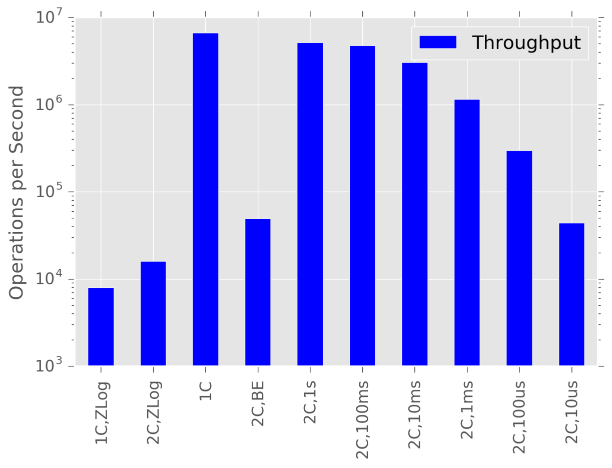


Figure 8: Forcing the client to drop their capabilities later (delay) improves throughput

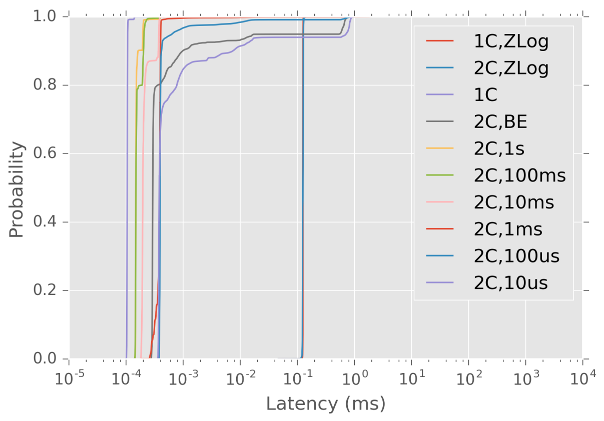


Figure 9: But if the clients hold their capabilities longer, it hurts latency for everyone.

6.2.3 CUT THIS Experiment: Client v. MDS caps + Failures

6.2.4 Experiment: Mantle Sequencer Balancer

7 Conclusion and Future Work

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

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

In conclusion, this paper should be accepted.

8 Bibliography

[1] S.A. Weil, S.A. Brandt, E.L. Miller, D.D.E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006.

[2] S.A. Weil, A.W. Leung, S.A. Brandt, and C. Maltzahn, "RADOS: A scalable, reliable storage service for

petabyte-scale storage clusters," *Proceedings of the 2nd international workshop on Petascale data storage: Held in conjunction with Supercomputing '07*, 2007.

[3] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. Davis, "Corfu: A Shared Log Design for Flash Clusters," *USENIX's OSDi 2012*, 2012.

[4] M.A. Sevilla, N. Watkins, C. Maltzahn, I. Nassi, S.A. Brandt, S.A. Weil, G. Farnum, and S. Fineberg, "Mantle: A programmable metadata load balancer for the ceph file system," *Proceedings of the international conference for high performance computing, networking, storage and analysis*, 2015.

[5] J.B. Buck, N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. Brandt, "SciHadoop: Array-based query processing in hadoop," *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, New York, NY, USA: ACM, 2011, pp. 66:1–66:11. Available at: http://doi.acm.org/10.1145/2063384.2063473.

[6] J. Buck, N. Watkins, G. Levin, A. Crume, K. Ioannidou, S. Brandt, C. Maltzahn, N. Polyzotis, and A. Torres, "SIDR: Structure-aware Intelligent Data Routing in Hadoop," *Proceedings of sC13: International conference for high performance computing, networking, storage and analysis*, Denver, Colorado: 2013, pp. 73:1–73:12. Available at: http://doi.acm.org/10.1145/2503210.2503241.

[7] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F.B. Cetin, and S. Babu, "Starfish: A Self-tuning System for Big Data Analytics," *Conference on innovative data systems research*, 2011, pp. 261–272.

[8] B. Behzad, H.V.T. Luu, J. Huchette, S. Byna, R. Aydt, Q. Koziol, M. Snir, and others, "Taming parallel i/O complexity with auto-tuning," *Proceedings of the international conference on high performance computing, networking, storage and analysis*, ACM, 2013, p. 68.

[9] K. Shvachko, "HDFS Scalability: The limits to growth," *login*, vol. 35, 2010.

[10] A. Balmin, T. Kaldewey, and S. Tata, "Clydesdale: Structured Data Processing on Hadoop," *Proceedings of the 2012 aCM sIGMOD international conference on management of data*, Scottsdale, Arizona, USA: 2012, pp. 705–708. Available at: http://doi.acm.org/10.1145/2213836.2213938.

[11] R. Ierusalimschy, W. Celes, and L.H. de Figueiredo, "Lua - the programming language. Web page. www.lua.org/home.html."

[12] R. Ierusalimschy, L.H. de Figueiredo, and W. Celes, "Lua – an extensible extension language," *Software - Practice and Experience*, vol. 26, 1996, pp. 635–652.

[13] M. Pall, "LuaJIT. Web page. luajit.org/luajit.html."

[14] M. Grawinkel, T. Süß, G. Best, I. Popov, and A. Brinkmann, "Towards dynamic scripted pNFS layouts," *PDSW'12*, Salt Lake City, UT: 2012.

[15] L. Vieira Neto, R. Ierusalimschy, A.L. de Moura, and M. Balmer, "Scriptable Operating Systems with Lua," *Proceedings of the 10th aCM symposium on dynamic languages*, New York, NY, USA: ACM, 2014, pp. 2–10. Available at: http://doi.acm.org/10.1145/2661088.2661096.

[16] M. Grawinkel, T. Sub, G. Best, I. Popov, and A. Brinkmann, "Towards Dynamic Scripted pNFS Layouts," *Proceedings of the 2012 sC companion: High performance computing, networking storage and analysis*, 2012.

[17] N. Watkins, C. Maltzahn, S. Brandt, I. Pye, and A. Manzanares, "In-Vivo Storage System Development," *Euro-par 2013: Parallel processing workshops*, Springer, 2013, pp. 23–32.

[18] R. Ierusalimschy, *Programming in Lua, Second Edition*, 2006.

[19] S.R. Alam, H.N. El-Harake, K. Howard, N. Stringfellow, and F. Verzelloni, "Parallel I/O and the Metadata Wall," *Proceedings of the 6th workshop on parallel data storage*, 2011.

[20] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," *Proceedings of the 19th aCM symposium on operating systems principles*, 2003.

[21] D. Hildebrand and P. Honeyman, "Exporting Storage Systems in a Scalable Manner with pNFS," *Proceedings of the 22Nd iEEE / 13th nASA goddard conference on mass storage systems and technologies*, 2005.

[22] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhu, "Scalable Performance of the Panasas Parallel File System," *Proceedings of the 6th uSENIX conference on file and storage technologies*, 2008.

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

[24] D. Roselli, J.R. Lorch, and T.E. Anderson, "A Comparison of File System Workloads," *Proceedings of the annual conference on uSENIX annual technical conference*, 2000, pp. 4–4.

[25] K. Chasapis, M.F. Dolz, M. Kuhn, and T. Ludwig, "Evaluating Lustre's Metadata Server on a Multi-socket Platform," *Proceedings of the 9th parallel data storage workshop*, 2014.

[26] S.V. Patil and G.A. Gibson, "Scale and Concurrency of GIGA+: File System Directories with Millions of Files," *Proceedings of the 9th uSENIX conference on file and storage technologies*, 2011.

[27] K. Ren, Q. Zheng, S. Patil, and G. Gibson, "IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion," *Proceedings of the 20th aCM/IEEE conference on supercomputing*, 2014.

[28] S.A. Brandt, E.L. Miller, D.D.E. Long, and L. Xue, "Efficient Metadata Management in Large Distributed Storage Systems," *Proceedings of the 20th iEEE/11th nASA goddard conference on mass storage systems and technologies*, 2003.

[29] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," *Proceedings of the 1st uSENIX conference on file and storage technologies*, 2002.

[30] D. Hildebrand, L. Ward, and P. Honeymoon, "Large files, small writes, and pNFS," *Proceedings of the 20th annual international conference on supercomputing (iCS06)*, 2006.

[31] S.A. Weil, K.T. Pollack, S.A. Brandt, and E.L. Miller, "Dynamic Metadata Management for Petabyte-Scale File Systems," *Proceedings of the 17th aCM/IEEE conference on supercomputing*, 2004.

[32] N. Watkins, C. Maltzahn, S. Brandt, and A. Manzanares, "DataMods: Programmable File System Services," *PDSW'12*, 2012.