Malacology: A Programmable Storage System Built on Ceph

Carlos Maltzahn, Michael Sevilla, Noah Watkins, Ivo Jimenez
UC Santa Cruz
{carlosm,msevilla,jayhawk,ivo}@soe.ucsc.edu

***Abstract*—We 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. Ceph is a distributed storage system with monitor processes that maintain cluster state using consensus, versioning, and consistency protocols; object storage daemons that store data on disk using peer-to-peer techniques like replication, load balancing, and consensus; and metadata daemons that act as gateways for file-based storage using protocols for consistency, balancing load, and mediating shared access. In this work, we take a "programmable storage" approach to leveraging these subsystems to build two services, a POSIX metadata load balancer and a distributed shared commit-log.**

I. Introduction

Large distributed systems tackle difficult distributed systems problems with code-hardened subsystem, but many of these components are never re-used or re-purposed. New scale requirements for data processing architectures and the increasing speed of storage devices are disruptors for storage and data management. As a consequence, layers of the software stack become obselete and code paths grow longer and more obfuscated. Evolution of the storage system can affect the robustness of code-hardened subsystems, yet there needs to be an equally flexible storage system to address the diverse performance requirements of the application.

For example, Ceph~[**?**] addresses *durability* with its RADOS object store (e.g., replication, erasure coding, and data scrubbing), *consistent versioning* by having daemons exchange "maps" of the cluster configuration, and *consensus* by having monitor daemons (MONs) use PAXOS. We contend that re-using and re-purposing these code-hardened subsystems is paramount to (1) improving the longevity and community uptake of "research quality" code and (2) avoiding duplication of the same protocols and algorithms throughout the system. Unfortunately, many internal subsystems are not exposed to other parts of the systems.

In this paper, we examine the programmability of Ceph, the open-source storage system solution backed by Red Hat. Ceph is known as the swiss army knife of storage, offering file, object, and block APIs for applications. The main draw of Ceph is the flexibility to use all three layers on the same storage system. While Ceph is one of the most flexible distributed systems out there, we contend that the system could be *even more programmable*. With minimal changes to the architecture, and building on many of the subsytems already baked into Ceph, we can build large research-quality systems.

We present Malacology, a programmable storage system capable of incorporating new functionality and re-purposing existing subsystems. We build the framework on Ceph~[**?**] by leveraging the subsystems in the monitor daemons (MONs), object storage daemons (OSDs), and metadata server daemon(MDSs). As shown in Figure~**??**, this framework is expressive enough to provide the functionality necessary for implementing other research-quality systems and services. Our contributions are:

- a programmable storage system implementation
- re-using code-hardened systems: sandboxed and vetted
- example systems that use this framework

1. shared log service based on CORFU~[**?**]
2. metadata load balancer based on Mantle~[**?**]

In the remainder of this paper we demonstrate the power of programmable storage. First we describe programmable storage in more depth (§**??**). Next we introduce our implementation of a programmable storage framework that exposes and re-uses many structures in Ceph (§IV). We conclude with descriptions and evaluations of these ideas by synthesizing entirely new storage services on an existing system through configuration and small changes: a distributed shared log (§**??**) and a programmable metadata load balancer (§**??**).

II. 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. In this section we touch on each subject using examples from both the Ceph and Hadoop communities to show the breadth and extent of the problem.

A. "Bolt-on" services

So called "bolt-on" services are 3rd party systems that are integrated into the targeted software stack to accomodate data management and faster devices. But these extra services come at the expense of additional sub-systems, dependencies, and data layouts that the application must manage, as well as trust. For example, MapReduce performs poorly for iterative and interactive computation because of its failure model that relies on on-disk storage of intermediate data. Many have added services to Hadoop to keep more data in the runtime (e.g., HaLoop[?], Twister[?], CGL-MapReduce[?], MixApart[?]). While performance improves, it comes at the cost of simplicity.

Furthermore, "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 pre-conceived notions of storage, namely that data in storage systems should be safe from data loss, but would introduce unecessary scalability limits [?].

B. 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. For instance, an application may change itself to exploit data locality or I/O parallelism in a distributed storage system.

For example, SciHadoop~[] 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.

C. 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.

1) Tuning: Tuning the system usually refers to setting parameters about the system. For large complicated systems this can be near impossible because the systems have so many knobs (xxxx tunables in Hadoop [?]) and sometimes the knobs are difficult to understand or quantify (xx tunable in Ceph [?]). 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 "knobs" that the storage system exposes (e.g. block size) and can be overwhelmed with too many parameters. Starfish [] 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 [?]. 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.

As an alternative, developers have started using the active storage component of Ceph, which is called object interface classes (or `cls`). Figure 2 shows a dramatic growth in the use of co-designed interfaces in the Ceph community since 2010. Figure 3 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 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 3 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 I, show that this active storage development is present throughough the Ceph software stack.

The popularity of the active storage component of Ceph, which is called object interfaces, hints 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 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.

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~[?]. This has lead to modifications to its architecture or API~[?] to improve performance.

As an example, ZLog needs to store metadata information with each object. 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.

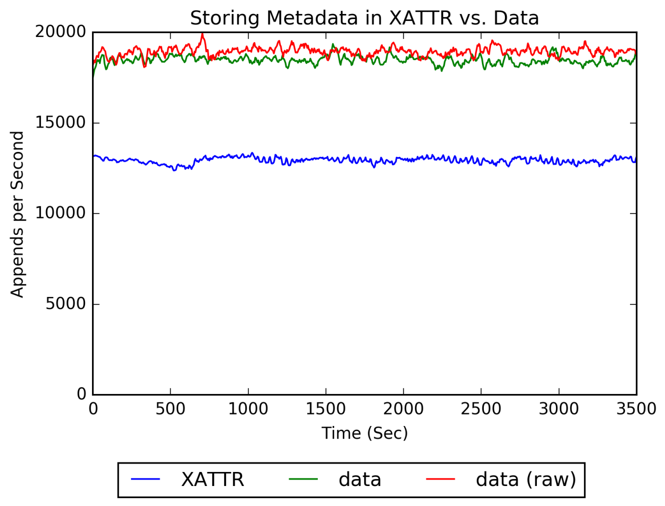

Fig. 1. [source] When appending data to objects, the object class that stores metadata in a header in the data byte stream (data) performs 1.5x better than the object class that stores metdata in the extended attributes of the object (XATTR); it is almost as fast as appending data without updating metadata (data raw).

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) Active Storage: Active storage is a hybrid approach to changing the application and storage system. Pushing computation closer to the data is not a new idea. Active storage techniques are used in production Ceph environments to have object storage devices (OSDs) process data before sending over the network or storing it. The code that does the processing in the OSD is called an `object storage interface` and it 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.

Developers and users actively develop new object storage interfaces. 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.

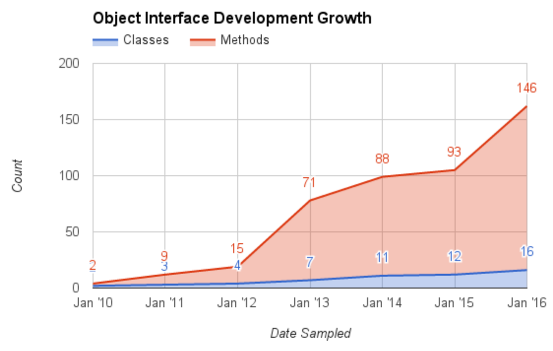

Fig. 2. Since 2010, the growth in the number of co-designed object storage interfaces in Ceph has been accelerating. This plot is the number of object classes (a group of interfaces), and the total number of methods (the actual API end-points).

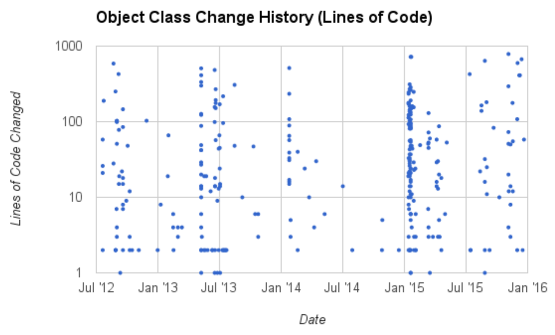

Fig. 3. Source code changes over time indicate the dynamic interface development will need to support a high frequency of interface churn. Each dot represents a Ceph commit with a corresponding number of lines of code changed.

TABLE I
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	6
	Exclusive	
Logging	Replica	3
	State	4
	Timestamped	4
Metadata Management	RADOS Block Device (RBD)	37
	RADOS Gatway (RGW)	27
	User	5
	Version	5
Garbage Collection	Reference Counting	4

III. 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.

A. Active Storage

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.

B. Durability

Ceph provides storage by striping and replicating data across RADOS, 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.

C. Consistency and Versioning of Cluster State

Ceph needs to keep track of cluster state and it does this with separate "services": client authentication, logging, metadata server (MDS) maps, monitor (MON) maps, object storage device (OSD) maps and placement group (PG) maps. These services are all managed in Ceph monitor processes (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.

IV. Implementation

We present a framework for managing Lua object classes comprised of 3 parts:

1. general and customizable shared libraries for the OSDs and MDSs using Lua
2. storing Lua object classes as RADOS objects
3. monitor command for specifying the Lua class

A. Distributing Work with Customizable Interfaces

Our framework has 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, portabability, 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~[**?**]. 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.

Lua is also portable. The object interfaces are

Lua is secure (sandboxing).

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.

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.

Finally, Lua is an interpreted language so object class interfaces written in Lua are more portable. 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.

1) 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.

2) Generalizing the Class Handler:

- Re-Used Components: Class Handler, Lua Class
- Durability with RADOS
- Send functionality with request
- Loading from the file system

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.

3) Storing Lua object classes as RADOS objects: [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.

B. Monitor command for specifying Lua class}

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.

C. 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

V. Services Built on Malacology

A. Mantle: A Programmable Metadata Load Balancer

Many distributed file systems decouple metadata and data I/O so that these services can scale independently~[**?**], [**?**], [**?**], [**?**], [**?**], [**?**]. Despite this optimization, scaling the metadata services is still difficult because metadata accesses impose small and frequent requests on the underlying storage system~[**?**]. Many techniques for designing the metadata services have been proposed to accommodate this workload: Lustre[?], GFS[**?**], and HDFS~[**?**] keep all metadata on one server; GIGA+~[**?**], IndexFS~[**?**], Lazy Hybrid~[**?**], GPFS~[**?**], and pNFS~[**?**] hash the file system namespace across a dedicated cluster and cache metadata; Panasas~[**?**] and CephFS~[**?**] partition the file system namespace into subtrees and assign them to servers. These systems have novel mechanisms for scalable metadata but the techniques are often “locked-in” to the systems they are implemented on.

Mantle~[**?**] is a programmable metadata balancer that separates the metadata balancing policies from their mechanisms. Administrators 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.

1) Quantifying Metrics:

2) Load Balancing:

3) Auto-tuning:

B. ZLog: A Distributed Shared Commit Log

1) Consistency:

2) Striping Strategies:

3) Tiering: Malacology uses the same Active and Typed Storage module presented in DataMods~[]; 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.

VI. 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.

```
    not sure why ./build fails without this
```

VII. Bibliography