In-Vivo Storage System Development

Noah Watkins[1], Carlos Maltzahn[1], Scott Brandt[1], Ian Pye[2], and Adam Manzanares[3]

[1]University of California, Santa Cruz, {jayhawk,carlosm,scott}@cs.ucsc.edu
[3]California State University, Chico, amanzanares@csuchico.edu
[2]CloudFlare, Inc., ianpye@gmail.com

Abstract

The emergence of high-performance open-source storage systems is allowing application and middleware developers to consider non-standard storage system interfaces. In contrast to the common practice of translating all I/O access onto the POSIX file interface, it will soon be common for application development to include the co-design of storage system interfaces. In order for developers to evolve a co-design in high-availability clusters, services are needed for *in-vivo* interface evolution that allows the development of interfaces in the context of a *live* system.

Current clustered storage systems that provide interface customizability expose primitive services for managing static interfaces. For maximum utility, creating, evolving, and deploying *dynamic* storage interfaces is needed. However, in large-scale clusters, dynamic interface instantiation will require system-level support that ensures interface version consistency among storage nodes and clients. We propose that storage systems should provide services that fully manage the life-cycle of dynamic interfaces that are aligned with the common branch-and-merge form of software maintenance, including isolated development workspaces that can be combined into existing production views of the system.

1 Introduction

The emergence of high-performance open-source storage systems are permitting applications and middleware architects to look beyond the standardized POSIX file interface towards co-designed, domain-specific storage interfaces that offer unique opportunities for optimization. However, current systems that provide extensibility expose only primitive interface management facilities, and provide little to no isolation between application developers that evolve storage interfaces.

Consider an application that captures, stores, and analyzes log-style data—a common task in analytics platforms. Ingested logs are partitioned and written to objects within a distributed storage system. Despite being used in this case to store semi-structured data, low-level object stores typically present a single, static byte-oriented interface. Being constrained by a byte-oriented interface makes it difficult to provide fine-grained data access such as predicate-based filtering, or to evolve low-level data layout. Previous work on active storage systems has shown the benefits of co-locating processing data such as reducing data transfer and exploiting I/O and CPU parallelism [5, 7, 10, 4] However, existing systems tend to assume long-lived, *installed* interfaces having unconstrained system access, and do not provide services for managing the evolution of interfaces that are aligned with the development methodologies of applications that use custom interfaces. Addressing these concerns for the effective adoption of extensible storage services is important.

Applications and storage system interfaces are inherently co-designed, with the POSIX file interface representing a long-lived static interface assumed to exist on virtually all platforms. A storage system that allows its interfaces to be dynamically defined presents a challenge for application development because dynamic storage interfaces are not directly managed within the application run-time environment. We argue in this paper that managing the deployment, consistency, and versioning of interfaces, as well as enforcing isolation between developers and production interfaces is best handled by the storage system itself.

The development of co-designed storage system interfaces is an entirely software-based activity tightly coupled with the development of a driving application. In particular, it is very common for engineering teams to follow a branch-and-merge source-code management style using software such as Git, Perforce, or Subversion, in which feature branches are merged into a production line after some period of insulated feature development

and maturation. While application feature development can often take place using, for example small-scale deployments on developer desktops, the same is not true for storage system interface development, where access to distributed resources and the peculiarities of live data are crucial to feature development and testing correctness at scale. One option is to allow developers unconstrained access to the storage system, relying on informal team guidelines to avoid conflicts. However, branch-based feature development emphasizes developer isolation. It would be useful if the storage system also provided a development environment for storage interfaces akin to the isolated development workflows for applications.

In the remainder of this paper we present a solution based on the concept of a developer *workspace*. A workspace represents a unit of isolation within the storage system that allows for the independent evolution of interfaces that are dynamically created using a high-performance embedded scripting language. The system fully manages versioned interfaces within a workspace, ensuring a consistent view of interface versions between storage system clients and co-designed interface. Developers may merge interfaces from their workspace into production views of the system, providing an evolutionary development path aligned with common software maintenance protocols.

2 Motivation

The collection and analysis of streaming data such access logs, click streams, and sensor data require scalable, fault-tolerant systems for high-performance ingestion, and post-processing is typically applied to extract knowledge using batch-oriented analysis. Figure 1 illustrates a typical architecture in which time-ordered logs are partitioned by attributes such as user or group, and stored within objects in a distributed object-store. Shown in the same figure is a production application that interacts with the log data to produce analysis results, while an engineering team develop new features, and evolve the production deployment through standard source-code management techniques.

This architecture of decoupling storage from analysis is extremely common, however one challenge that arises is the I/O efficiency of data-intensive analysis. The object-level interfaces exposed by storage systems provide generic, byte-oriented access similar to file interfaces. Byte-oriented interfaces force applications to perform coarse-grained data access, despite analysis that may exhibit low-selectivity on the data being processed.

Alternatively, work in active storage has shown that domain-specific interfaces can be constructed within the storage system, and provide efficient, fine-grained data access. Domain-specific interfaces can provides access

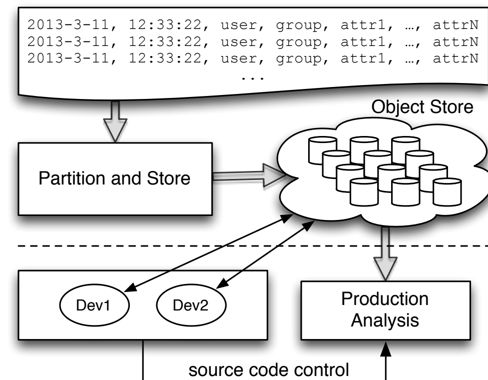


Figure 1: Log data is stored in objects that are batch analyzed while developers create new features and evolve the system.

to, for example, the arithmetic mean of a single attribute computed over the records contained in a single object. Such an interface implemented within the storage system allows applications to avoid unnecessary data transfers and recomputation by caching results, perform asynchronous work such as indexing, and reduces application complexity.

Allowing application developers to dynamically construct co-designed storage interfaces as part of the normal development process is a powerful construct for building distributed applications. However, the tight coupling between storage interfaces and applications require that both components can *evolve together* through a standard software development life-cycle.

2.1 Storage Interface Evolution

Dynamically created storage interfaces pose a challenge for software development because application software may evolve independently from the deployed storage interfaces, but still require strong version consistency. Recall from Figure 1 that multiple developers evolve a production analysis application using feature branch services provided by a source-code control library. If each developer may now co-design alternative storage interfaces within a single cluster, isolating the effects of each developer and from each other and from live data become important.

Consider the application life cycle depicted in Figure 2. Developers *Dev1* and *Dev2* are responsible for developing independent, domain-specific interfaces—arithmetic average, and minimum—that will replace the same computation computed remotely by the production analysis application. Each developer must now evolve the storage-level interfaces, as well as change application-level code to take advantage of the new the

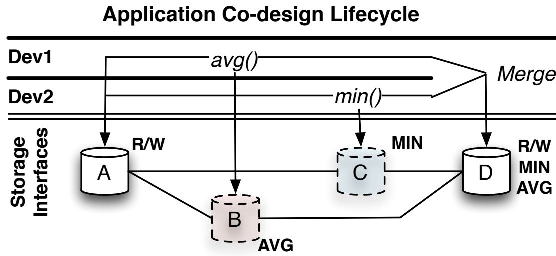


Figure 2: Developers evolve application software and storage interfaces through a co-design process.

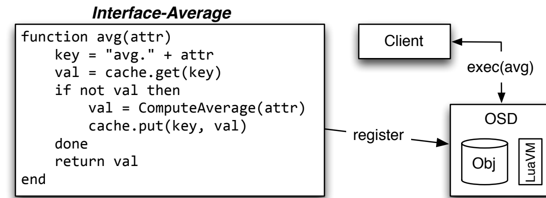


Figure 3: An interface defining an *average* function is registered with an OSD after which point a client may remotely invoke the method on an object.

features. For instance, both developers begin with a base storage interface exposing the standard byte-oriented interface (ver. A). Each developer evolves the application and storage interfaces with their respective features (ver. B, C). Once the features are complete, they are merged to expose the new interfaces to the production application (ver. D).

Next we discuss dynamic interfaces, the low-level building for our system.

3 Extensible Object Interfaces

A core building block of our system is a service that allows the dynamic creation of low-level object interfaces using a high-performance embedded scripting language. We consider our design in the context of the Ceph distributed storage system, and begin our discussion with a brief overview of Ceph and its object-based storage system.

3.1 The RADOS Object Store

The RADOS object store is a highly scalable, fault-tolerant storage service that forms the basis for high-level Ceph services such as the Ceph File System [9, 8]. A RADOS cluster consists of a set object-storage devices that expose a rich object interface including byte-oriented access methods as well extended attributes, indexing, and snapshots. Clients access objects through a library that hides the cluster layout, network, and fault recovery. In addition to its natively supported interfaces, objects in RADOS can be extended by constructing C++-based plugins that defines new methods on objects, analogous to creating a sub-class in an object-oriented language. A method is invoked against a target object by a client and is transparently executed within the storage server process responsible for the object.

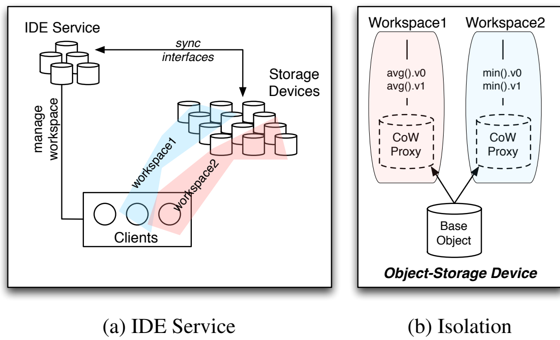
The extensibility of RADOS objects is very powerful, and can be used to construct interfaces like to those discussed in Section 2. Unfortunately it is non-trivial to deploy statically compiled, architecture dependent interfaces within a high-availability cluster, making it difficult to integrate rapid interface evolution with the iterative development of applications. What is needed is a mechanism for dynamically constructing new object methods.

3.2 Dynamic Interfaces

We have extended the object-storage devices in RADOS to support dynamically defined object interfaces using the Lua language, specifically designed to be embedded in high-performance applications. New interfaces are created by sending to a storage device a Lua script that defines any number of methods, after which point the interfaces are made available through any existing RADOS client libraries.

Figure 3 illustrates how dynamic interfaces are used. First, a developer authors a Lua script that defines a new object method. Shown in the figure is a script that computes the arithmetic mean of an attribute over the records in an object. Notice that before computing the average a cache is queried to avoid recomputing results. A client that invokes this method on an object will trigger the method automatically within the OSD process and the results will be returned to the client, potentially avoiding recomputation. Scripts may be pre-registered, or sent along side a client request for completely dynamic behavior.

This basic mechanism of constructing dynamic interfaces using small code fragments allows applications to easily evolve storage interfaces at a fine-granularity. However, two major issues arise. First, when working within a live system, developers should be able to work independently without worrying about causing conflicts. And second, in a large, elastic system developers should not have to be involved in the details of ensuring that a consistent view of their deployed interfaces are present on all system devices. We propose that the storage system expose a specialized development environment that abstracts away these concerns.



(a) IDE Service (b) Isolation

Figure 4: In (a) clients use an IDE service to create workspaces that form a context within the storage system. In (b) base data is not duplicated, and CoW provides isolation for interface private data.

4 Interface Development Environment

We propose an environment for developing new interfaces consisting of one or more *workspaces* that, at a high-level, provide isolation between between interfaces.

4.1 Workspaces

A workspace is an entity managed by the storage system which provides isolation between dynamic interfaces. Workspaces can be created, destroyed, and merged through the use of the *interface development environment* (IDE) service, illustrated in Figure 4a. The IDE service exposes an interface similar to that of Git, Mercurial, or Subversion in which branches form the basic unit of isolation. A typical use of a workspace by a developer is to provide a safe environment to construct, test, and refine storage interfaces as part of a larger application development process. Once a developer creates a workspace, storage clients interact with object devices in the context of the workspace—analogous to authentication or user contexts—allowing the system to enforce isolation guarantees on a per-workspace basis.

Isolation. Workspace contexts are used within storage devices to enforce isolation between interfaces that interact with objects. Interfaces that perform only read operations on an object require no special handling—reads are directed to the base object. However, writes must be carefully handled as to not interfere with state created by interfaces in other workspaces, or data in the base object. For instance, the interface shown in Figure 3 caches a computed *average* value by constructing a key and saving it in a object-local cache. Write operations that access the cache, indexes, extended attributes, or other object services are intercepted by the storage device and isolated by transparently applying a namespace to keyed data accesses. Some interfaces will reformat object payloads in order to test the performance impacts on alternative data layouts. The system employs copy-on-write (CoW) techniques to make this efficient, and internally addresses CoW partitions using namespacing.

Partitioning. By default a client operating within the context of a workspace has access to all objects on any object device (subject to the standard privilege restrictions imposed by the system). However it may be the desirable to partition a cluster between development and production to allow extra protection, such as providing physical I/O and CPU isolation. Existing facilities within the RADOS storage system for custom replication and tiering policies allow subsets of data to be placed onto specific sets of nodes. Workspaces can be linked to these physical partitions which ensure that the space of addressed objects is constrained by the physical partitioning.

4.2 Workspace Management

Ultimately interfaces defined within workspaces as part of application development will be migrated into a production environment. For instance, the interfaces defined in separate workspaces shown in Figure 2 can be merged into production, providing access to the union of the interfaces to applications accessing the storage system in the context of the production workspace.

There are several issues that may arise when merging workspaces. First, at a high-level merging changes the visibility of interfaces, and as a result interface naming conflicts may arise. For instance, two workspaces may define the same interface. These types of conflicts are largely application-specific and must be handled explicitly by developers. Like SCM systems, the primary responsibility of the storage system is to provide feedback to developers about the changes they are making through the IDE service.

Interfaces that utilize private data can be merged without low-level conflicts by migrating the same isolation parameters (e.g. namespacing) used to prevent conflicts between workspaces. However, for interfaces that perform heavy-weight data transforms such as new data layouts, migrating all interfaces to a use a new layout may be necessary. In order to make format migration easier, workspace merging can optionally specify a transformation routine that the system ensures is applied prior to invoking any interface following the merge. Finally, the removal of workspaces results in lazy deletion of all unmerged interface state created during the lifetime of the workspace.

5 The IDE Service

Interfaces and workspaces are inherently cluster-wide entities that must be managed by the storage system. For instance, workspace isolation may be implemented by constructing a globally unique namespace prefix, and source code defining interfaces must be versioned, stored, and deployed to cluster nodes where interface versions are matched to client requests and workspace context. In an elastic cluster that experiences failures and maintenance, automation is essential.

A core service often found in distributed systems is a highly available versioned data store commonly implemented using a consensus algorithm, such as Paxos. For instance, Ceph uses *monitor services*, built upon Paxos, to manage cluster membership, service discovery, replicated logs, and authentication. A monitor provides a consistent view of the system state, and clients and OSDs can contact a monitor to synchronize their states. The IDE service we purpose can be built within a specialized monitor to provide remote access to the service, as well as providing a mechanism for OSDs to synchronize their view of interface versions.

Integration. Finally, a mechanism is needed to relate interface versions managed by the storage system with application development. Many source-code control systems provide the ability to inject external information into the revision history. For instance, Git allows external repositories to be seamlessly integrated, and CVS tags allow the expansion of macros within source code. We envision a similar form of integration that allows macro-like expansion of interface versions to be included in application development source. However, simpler alternatives such as providing a tagging service to developers may also be sufficient.

6 Related Work

In Oasis [10] the T10 standard is extended to allow new active storage scripts to be injected into an object storage device. Primitive script management allows creation, listing, and removal of new application functions, but does not address higher-level management challenges such as versioning or isolating contexts.

GlusterFS [1] translators provide a rich mechanism for adding functionality at different levels of the storage system. Translators are statically defined and designed to be a long-lived *installed* component.

Building new pNFS striping strategies using the Lua scripting language have been proposed [3]. The script defining a new strategy is embedded in a file inode where script versioning is aligned with inode consistency mechanisms.

A variety of versioning file systems allow data to be managed using checkpoints, version tags, and other protocols [6]. While these are concerned with user-level payload data, the scalability lessons from this body of work dealing with versioning may prove useful.

The Git source-code control library has been integrated into a FUSE-based file system to extend its versioning features to files and directories [2]. We are also considering making use of the Git library for its rich, embeddable interface for managing and versioning textual data such as Lua code snippets.

7 Conclusion

Providing non-POSIX storage interfaces has been the topic of much research. However, little has been done to address the management of co-designed storage interfaces with application development processes. In this paper we have proposed that storage systems provide a service based on developer workspaces that enforce isolation within the storage system, and allow storage interfaces to safely evolve from development into production.

References

[1] Glusterfs clustered file system. http://www.gluster.org.

[2] GRANT, R. Filesystem interface for the git version control system. Tech. rep., University of Pennsylvania, 2009.

[3] GRAWINKEL, M., SUSS, T., BEST, G., POPOV, I., AND BRINKMANN, A. Towards dynamic scripted pnfs layouts. In *PDSW '12* (2012).

[4] LIM, H., KAPOOR, V., WIGHE, C., AND DU, D. H.-C. Active disk file system: A distributed, scalable file system. In *MSST '08* (2008).

[5] PIERNAS, J., NIEPLOCHA, J., AND FELIX, E. J. Evaluation of active storage strategies for the lustre parallel file system. In *SC '07* (2007).

[6] SANTRY, D. S., FEELEY, M. J., HUTCHINSON, N. C., VEITCH, A. C., CARTON, R. W., AND OFIR, J. Deciding when to forget in the elephant file system. In *SOSP '99* (1999).

[7] SON, S. W., LANG, S., CARNS, P., ROSS, R., THAKUR, R., OZISIKYILMAZ, B., KUMAR, P., LIAO, W.-K., AND CHOUDHARY, A. Enabling active storage on parallel i/o software stacks. In *MSST '10* (2010).

[8] WEIL, S., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *OSDI '06* (2006).

[9] WEIL, S., LEUNG, A., BRANDT, S. A., AND MALTZAHN, C. Rados: A fast, scalable, and reliable storage service for petabyte-scale storage clusters. In *PDSW '07* (2007).

[10] XIE, Y., MUNISWAMY-REDDY, K.-K., FENG, D., LONG, D. D. E., KANG, Y., NIU, Z., AND TAN, Z. Design and evaluation of oasis: An active storage framework based on t10 osd standard. In *MSST '11* (2011).