In-Vivo Storage System Development

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

[1] University of California, Santa Cruz {jayhawk,carlosm,scott}@cs.ucsc.edu
[2] California State University, Chico nmtadam@gmail.com
[3] CloudFlare, Inc. ian@cloudflare.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 practice of virtually always designing for file-like byte-stream interfaces, co-designed domain-specific storage system interfaces are becoming increasingly common. However, in order for developers to evolve interfaces in high-availability storage systems, 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 ad-hoc interfaces. For maximum utility, the ability to create, evolve, and deploy *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 client applications. 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 is permitting applications and middleware developers to look beyond traditional file-like interfaces towards co-designed, domain-specific storage interfaces that offer unique opportunities for optimization. However, storage interfaces are inextricably tied to the ability to interpret and access data, elevating the criticality of their preservation and management in storage systems to that of the data artifacts themselves. Despite open-source storage systems paving the way for increased extensibility, systems currently lack any services for managing the life cycle of interface development.

Domain-specific interfaces allow applications to custom tailor data services such as I/O and co-located processing to realize optimizations not possible with generic file-like interfaces. For instance, structure and data type knowledge exposed through an interface can be used to guarantee data alignment properties and enable low-level data processing such as filtering and data transformation.

These types of services, such as filtering, are largely inefficient to perform in middleware compared to a server-side approach, but storage systems deployed today do not offer any type of service for dynamic extensibility. Previous work on active storage systems have shown the benefits of co-locating processing with data, such as reducing data transfer and exploiting I/O and CPU parallelism [1–4]. And more recently, the Rhea [5] system showed that filtering kernels could be extracted from Hadoop jobs using static analysis and applied transparently in cloud storage services such as Microsoft Azure [6]. It is also anticipated that in next-generation exascale systems, array-oriented interfaces and script-based analysis function shipping will be supported natively by storage systems in an effort to circumvent scalability challenges presented by the POSIX file interface [7]. Unfortunately, previous solutions to providing extensibility have focused on architectures in which applications must fully manage interfaces, either by statically installing additional code, or packaging functions into each request. However, applications and storage system interfaces are inherently co-designed—an interface *defines* the data, and must be preserved for continued data access and interpretation. A storage system that allows its interfaces to be dynamically defined presents a challenge for application development, portability, and archival use cases because dynamic storage interfaces tend to be directly managed within the application run-time environment completely decoupled from stored data. 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, Mercurial, 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, error prone team guidelines to avoid conflicts such as naming or data format incompatibilities. Yet another option would be to maintain a smaller development cluster, but this leads to increased costs and may not expose the development process to realistic conditions. It would be useful if a storage system provided a development environment for storage interfaces as a first class service akin to the isolated development workflows for application developers using source-code management tools.

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

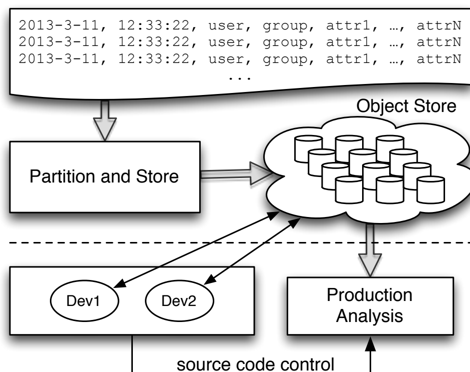


Fig. 1: Log data is stored in objects that are batch analyzed while developers create new features and evolve the system.

system fully manages versioned interfaces within a workspace, ensuring a consistent view of interface versions between storage system clients and co-designed interfaces. 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 large-scale read-mostly data such as access logs, click streams, and sensor data, as well as scientific simulation output, require scalable, fault-tolerant storage systems. Increasingly, these and other types of data commonly referred to as *big data*, are being maintained with cloud-based solutions using both object and file-based storage abstractions. Figure 1 illustrates a typical architecture in which data, such as time-ordered logs, are partitioned by attributes such as time or data source, and stored within objects in a distributed object-store such as Amazon S3. Shown in the same figure is a production application that interacts with the stored data objects by remotely reading and producing analysis results, or searching for activity patterns. Simultaneously, engineering teams may be developing and testing new features, as well as evolving the production deployment using standard source-code management techniques, workflows, and deployment operations procedures.

While this architecture of decoupling storage from analysis is extremely common, one challenge that arises is the I/O efficiency for data-intensive analysis tasks. For instance, simple filtering or computing statistical summaries are relatively inexpensive to perform, yet generally require transferring all dependent data across the network for analysis. The byte-oriented interfaces exposed by storage systems are a major impediment to offering new extensibility services because structural and type information at a low-level is needed perform these types of semantically rich operations. 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 provide access to, for example, the arithmetic mean of a single attribute computed over the records contained in a single object, apply a predicate derived from a high-level query, or reorganize data for more efficient access. Such an interface implemented within the storage system allows applications to avoid unnecessary data transfers, complex domain-specific middleware, and allows service providers an opportunity to offer a broad range of services such as offline, best-effort indexing and compression.

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, and that storage systems provide services for preserving installed interfaces within a large-scale cluster.

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 and compatibility between the application and deployed interfaces. Additionally, recall from Figure 1 that multiple developers may evolve an application by first developing and testing features, then integrating the changes into a production deployment. In order for each developer to work on features independently, conflicts that result from customized interfaces must be isolated and handled transparently.

Consider the application life cycle depicted in Figure 2. Developers *Dev1* and *Dev2* are responsible for developing independent, domain-specific interfaces to indvidual objects—arithmetic average, and minimum—that will replace the same per-object operation performed 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 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). Two interfaces can conflict if local object resources are not partitioned. For instance, if two interfaces implementing distinct statistical calculations (e.g. *mean* vs *median*) cache their result in a local object attribute to avoid recomputation, but use the same attribute name (e.g. *avg*), data corruption may lead to silent errors and unexpected results. Thus, providing transparent isolation between interfaces is important in order to avoid the type of ad-hoc coordination among developers that would otherwise be required. Next we discuss dynamic interfaces, the low-level building block for our system.

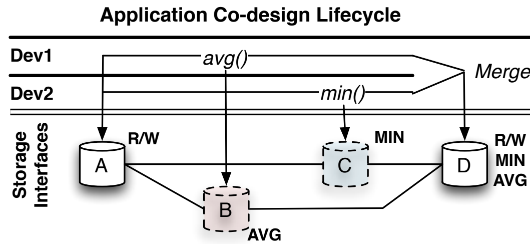


Fig. 2: Developers evolve application software and storage interfaces through a co-design process.

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. Note that while our framework assumes the existence of extensible object interfaces, many approaches to providing extensibility should be compatible. In this paper we consider for context a design based on the Ceph distributed storage system, and begin our discussion with a brief overview of Ceph and its object-based storage system called RADOS.

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 [8, 9]. 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 define new methods on objects, analogous to creating a sub-class in an object-oriented language. A method is invoked against a target object remotely by a client and is transparently executed within the storage server process responsible for the object.

The extensibility of RADOS objects is very powerful, can be used to construct a variety of interfaces such as those discussed in Section 2, and is used by Ceph internally, and in several products built on top of Ceph. 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.

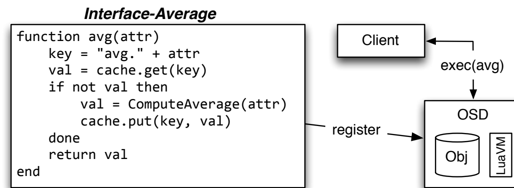


Fig. 3: An interface defining an *average* function is registered with an OSD after which point a client may remotely invoke the method on an object.

3.2 Dynamic Interfaces

We have extended the object-storage devices in RADOS to support dynamically defined object interfaces using the Lua language [10], specifically designed as an embedded language for 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 library.

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 a single object. Notice that before computing the average a cached attribute is queried to avoid recomputing results. A client that invokes this method on an object will trigger the method within the OSD process and the results will be returned to the client, potentially avoiding recomputation. Scripts may be preregistered, or sent along side a client request for completely dynamic behavior.

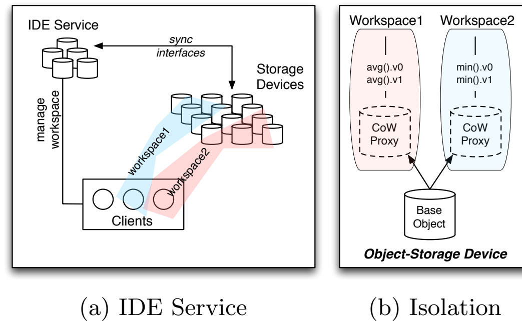
This basic mechanism of constructing dynamic interfaces using small code fragments allows applications to easily evolve storage interfaces at a fine-granularity. However, two major issues arise. First, when working within a live system, developers should be able to work independently without worrying about causing conflicts. For instance, the author of the interfaced shown in Figure 3 should not be affected by other developers that also cache data with the same name; the system should provide this isolation. 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. To solve these challenges we propose that storage systems provide developers with logically isolated workspaces for interface development, and native services for managing installed interfaces across a cluster.

4 Interface Development Environment

We propose an *interface developer environment* (IDE) for constructing new, native storage interfaces that consist of an isolated *workspace* abstraction that is well-aligned to common software development workflows, and integrates with provisioning and tiering abstractions already present within storage systems.

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 a development branch forms the basic unit of isolation. It is expected that the use of a workspace will resemble a developers *working copy* in the traditional sense of source-code control systems, providing a safe environment to construct, test, and refine a line of feature development. The key difference being that a workspace exists within and is managed entirely by the storage system for the purpose of providing server-side interfaces.



(a) IDE Service (b) Isolation

Fig. 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.

Isolation. Providing efficient isolation among workspaces is a challenge. In order to avoid expensive data duplication, the system should allow interfaces to share as much data as possible. For instance, read operations performed by an interface should require no special handling, and be satisfied by base data. However, writes must be carefully handled as to not interfere with state created by interfaces in other workspaces, or blob data associated with the object. For instance, the interface shown in Figure 3 caches a computed *average* value by constructing a key and saving it in an object-local cache. The framework must ensure that other interfaces are unaffected, a problem that could arise if, for example, two interface developers chose identical key names. Write operations that access object services, such as a local index or raw payload data, must be intercepted by the storage device and isolated. Efficient techniques for isolation depend on the type of service. For instance, isolation can be achieved using namespacing when storing key/value pairs, while copy-on-write techniques can be used for data transformations on blob data. Performance isolation is also an important aspect to consider. It is not unreasonable to use existing performance

isolation techniques in a system, but we have not considered the need for an entirely new method for performance isolation.

Partitioning. While logical isolation is important to ensure correctness, an organization may want to physically partition its storage in such a way that development workspaces reside on distinct hardware. The logical workspace entity should integrate with existing facilities within the storage system for custom data placement and tiering policies, allowing subsets of data to be placed onto specific sets of nodes. Workspaces can be linked to these physical partitions through existing system abstractions (e.g. a *pool* in Ceph) 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 source-code management systems, the primary responsibility of the storage system is to provide feedback to developers about the changes they are making through the interface development service, and help manage conflict resolution.

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 using new data layouts, migrating all interfaces to a use a new layout may be necessary. In order to make format migration easier, workspace merging should optionally specify a transformation routine that the system ensures is applied prior to invoking any interface following a merge operation. Finally, the removal of workspaces will result in lazy deletion of all unmerged interface state created during the lifetime of the workspace.

5 The IDE Service

Interfaces and workspaces are cluster-wide entities that must be managed by the storage system in the face of cluster failures, expansion, and policy changes. For instance, the storage system must ensure that a new storage node has the required interfaces present before it can service requests from applications requiring these interfaces. Further, newly registered versions of interfaces must be propagated to nodes within the system, and properly synchronized with applications expecting the latest version.

Luckily, existing services within distributed storage systems solve similar problems. For instance, 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. Such existing services are closely related to the requirements of the IDE service, and should be capable of being reused to provide an interface synchronization service within the cluster.

Integration. Finally, a mechanism is needed to associate interface versions managed by the storage system with the versions that applications expect. We are considering two possible solutions to this problem. First, some source-code control systems provide the ability to inject external context information into the managed content (e.g. CVS tags expansion). Extending or exploiting this feature may allow us to automatically generate version macros used to provide context when clients access the system. Similarly, systems such as Git allow external repositories to be seamlessly integrated into existing repositories. By allowing the storage system to export its own virtual Git repository, we can enable the system to present previously registered, versioned code automatically into a higher-level project repository. Providing an easy-to-use and robust integration solution is important for usability, and utilizing other techniques from RPC stub generation may prove to be very valuable.

6 Related Work

Oasis [3] is an active storage framework based on extensions to the T10 standard. Primitive script management associates scripts with *function objects*, but the project does not address high-level management challenges. Runde, et al. [11] use sandboxing virtualization technology to isolate server-side computations, but do not address any type of logical isolation necessary to build the workspace abstraction we have proposed.

GlusterFS [12] translators provide a rich mechanism for adding functionality at different levels of the file system. Translators are statically defined and designed to be a long-lived *permanent* extension.

Building new pNFS striping strategies using the Lua scripting language have been proposed [13]. The script defining a new strategy is embedded in a file inode where script versioning is aligned with inode consistency mechanisms.

The Elephant File System is a versioning file system allowing data to be managed using checkpoints, tags, and other protocols [14]. While this file system is only concerned with payload data, the scalability lessons from this body of work dealing with interface 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 [15]. 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, but are interested in mechanisms useful in distributed storage systems.

7 Conclusion

Providing alternative storage interfaces to the traditional byte-stream oriented interface 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. This management is important to address as interest in dynamic interfaces increases. In this paper we have proposed that storage systems provide a service based on developer workspaces that enforce isolation within the storage system, allowing storage interfaces to safely evolve from development into production, and integrate with existing provisioning abstractions and developer workflow protocols.

References

1. Piernas, J., Nieplocha, J., Felix, E.J.: Evaluation of active storage strategies for the lustre parallel file system. In: SC '07. (2007)
2. Son, S.W., Lang, S., Carns, P., Ross, R., Thakur, R., Ozisikyilmaz, B., Kumar, P., Liao, W.K., Choudhary, A.: Enabling active storage on parallel i/o software stacks. In: MSST '10. (2010)
3. Xie, Y., Muniswamy-Reddy, K.K., Feng, D., Long, D.D.E., Kang, Y., Niu, Z., Tan, Z.: Design and evaluation of oasis: An active storage framework based on t10 osd standard. In: MSST '11. (2011)
4. Lim, H., Kapoor, V., Wighe, C., Du, D.H.C.: Active disk file system: A distributed, scalable file system. In: MSST '08. (2008)
5. Gkantsidis, C., Vytiniotis, D., Hodson, O., Narayanan, D., Dinu, F., Rowstron, A.: Rhea: automatic filtering for unstructured cloud storage. In: NSDI '13. (2013)
6. Brad Calder, e.a.: Windows azure storage: A highly available cloud storage service with strong consistency. In: SOSP '11. (2011)
7. Acceleration, D.E.S.T.: Fastforward
8. Weil, S., Leung, A., Brandt, S.A., Maltzahn, C.: Rados: A fast, scalable, and reliable storage service for petabyte-scale storage clusters. In: PDSW '07. (2007)
9. Weil, S., Brandt, S.A., Miller, E.L., Long, D.D.E., Maltzahn, C.: Ceph: A scalable, high-performance distributed file system. In: OSDI '06. (2006)
10. : Lua language
11. Runde, M.T., Stevens, W.G., Wortman, P.A., Chandy, J.A.: An active storage framework for object storage devices. In: MSST '12. (2012)
12. : Glusterfs clustered file system. http://www.gluster.org
13. Grawinkel, M., Suß, T., Best, G., Popov, I., Brinkmann, A.: Towards dynamic scripted pnfs layouts. In: PDSW '12. (2012)
14. Santry, D.S., Feeley, M.J., Hutchinson, N.C., Veitch, A.C., Carton, R.W., Ofir, J.: Deciding when to forget in the elephant file system. In: SOSP '99. (1999)
15. Grant, R.: Filesystem interface for the git version control system. Technical report, University of Pennsylvania (2009)