Fast and Cautious Evolution of Cloud Storage

Dutch T. Meyer[†], Mohammad Shamma[†], Jake Wires[‡], Quan Zhang[†],
Norman C. Hutchinson[†], Andrew Warfield[†]

[†] *Department of Computer Science*
University of British Columbia
dmeyer,mshamma,quanz,andy,norm@cs.ubc.ca

[‡] *Citrix, Inc*
jake.wires@citrix.com

Abstract

When changing a storage system, the stakes are high. Any modification can undermine stability, causing temporary downtime, a permanent loss of data, and still worse - a loss of user confidence. This results in a cautious conservatism among storage developers. On one hand, the risks do justify taking great care with storage system changes. On the other hand, this *slow and cautious* deployment attitude is a poor match for cloud services tied closely to web-based frontends that follow an "always beta" mantra. Unlike traditional enterprise servers, cloud-based systems are still exploring what facilities should be provided by the storage layer, requiring that storage services be able to evolve as quickly as the applications that consume them. In this paper, we argue that by building support for evolution into the basic structure of a storage system, new features (and fixes) can be deployed in a *fast and cautious* manner. We summarize our experiences in developing such a system and detail its requirements and design. We also share some initial experience in deploying it on a rapidly evolving, but production, cloud hosting service that we have been building at UBC.

1 Introduction

Enterprise storage is expensive and boring. You buy it, the salesmen will explain, so that you can sleep at night. While this ethos continues to be appropriate for many storage customers, it stands in stark contrast to the approach emerging in cloud-based systems. In the cloud, applications are perpetually in beta, developers release often, and the drive for scale and application agility has led to new, rapidly-evolving storage services on a range of storage interfaces including blocks [2, 10], key-value pairs [6, 3], and database-style APIs [5]. The designers of these systems are continuously pulled in many directions as they try to meet new demands of scale, workloads, and evolving applications. While the "right" way to build storage for cloud environments is still being explored, we believe that a fundamental, common property of all storage-as-a-service systems is the requirement that they embrace change, by facilitating the frequent but safe deployment of software changes.

The work in this paper is motivated by our experience developing a new storage system *while* using it in a production environment. Recently, we deployed a virtual machine hosting service based on our storage system Parallax [10]. However, during the initial design, we could not anticipate all future uses, features, or workloads relevant to the system. As a result, we recently implemented new defragmentation and garbage collection features. We plan to implement and deploy tiering and block annotations in the next few months. In the former, different classes of blocks are stored with differentiated levels of replication, while in the latter the system stores additional per-block metadata such as usage information, taint tracking, and access patterns. Adding these features will fundamentally change the operation of the system, require new per-volume and per-block meta-data structures on disk, and modify the block lookup process. Traditionally, such upgrades would either imply a long stabilization process, or an unacceptable risk.

In the spirit of the web's "always beta" mantra, as each update to the storage system is developed, a selected subset of virtual machines will be upgraded to the new version in order to gain early experience with these features. Of course, we (and our clients) will be very disappointed if the storage system loses or corrupts any data. To that end, we introduce ***Dovetail***, a set of primitives that allow a wide range of modifications to be deployed safely and easily. The components of Dovetail are:

- Deployment and Migration Tools: Tools which enable administrators to apply upgrades selectively to specific storage consumers and their data.
- Safe Upgrade Mechanisms: Safeguards to ensure that data in a new storage system version can be

recovered from prior versions, and that flaws in an upgrade do not corrupt other versions of the system.

- Live Reconfiguration Interface: An interface which enables the modification of storage systems without noticeable service disruptions.

Together, these components provide an environment in which upgrades to a running system can be made safely, even if the upgrade itself is not safe. This has allowed us to incrementally redesign and upgrade a production cloud storage system to deploy more ambitious features, experiment with new designs, and shorten release cycles.

2 Storage Architecture Assumptions

Several architectural assumptions must hold for any storage service that would use Dovetail. Figure 1 shows an environment we consider typical. It divides the storage system into three abstract layers. At the top of the stack are applications and OSes that consume storage over some interface: block, file, key-values, or something else. At the bottom is a block layer that manages the allocation and assignment of physical storage resources in the spirit of systems such as Petal [9]. Within this model, we assume the ability to interpose *Tee* and *Isolation Layer* modules (shown shaded in Figure 1) on either side of the storage service to be upgraded.

Our environment, which motivated our interest in storage system upgrades, is a virtual hosting service. We currently support Parallax [10], which exports virtual disks transparently to VMs, and CouchDB [3], a scalable key-value store.

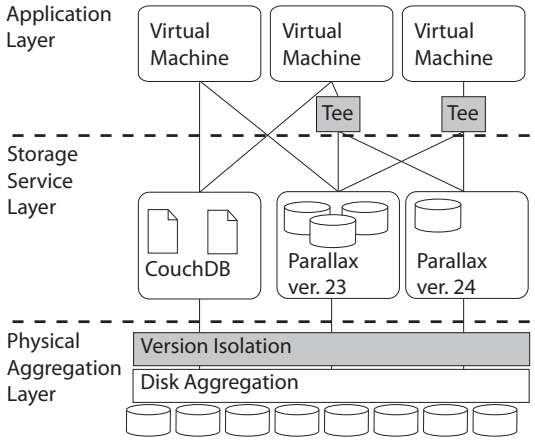

Figure 1: Our example cloud storage environment.

2.1 Supported Upgrades

The Tee and Isolation layer are thin shims that provide safe upgrade mechanisms to any storage service between them. Upgrades can consist of any code modification, ranging from a small patch to a complete re-write of the storage service. They can modify any aspect of the operation or on-disk layout of the system. Upgrades can also be applied to a subset of the on-disk data, leaving the remainder of the content to be migrated over time.

We assume that the interfaces above and below these shims stay reasonably fixed. While this assumption has been shown to hold for block and file interfaces, it represents a limitation for the introduction of interface-changing features. We briefly discuss how this class of upgrade (such as extended attributes on files or block device snapshots) might be incorporated in Section 5.

Dovetail does not attempt to support upgrades to the lower layers of the storage stack, including device firmware, drivers, or the physical aggregation layer. Our belief is that these layers are less likely to be a source of rapid evolution in cloud environments, and should be serviceable with conventional upgrade and maintenance techniques.

3 Architecture

A flow chart of Dovetail's upgrade process is provided in Figure 2. Each panel highlights one of three areas of concern: We must address which users see an upgrade and how their data is updated to new versions of the system (*Deployment and Migration*). We must ensure that the upgrade process itself is non-disruptive (*Live Reconfiguration*). Finally we must safeguard against software errors in each upgrade (*Safe Upgrade*). For the purposes of illustration, we will discuss each problem in the context of adding support for data tiering to Parallax, starting with the concern for safety.

3.1 Safe Upgrade

Data tiering, as mentioned in Section 1, allows critical on-disk structures to be replicated. While this is intended to increase reliability, the change also carries some risk. Since it modifies the block indexing code, a software error could cause structures that are intended to be replicated to instead be lost.

Safety in Dovetail is provided by two modules shown in Figure 1. *Tees* detect errors and recover by returning the service to its original state. Modifications made after the upgrade are preserved. *Version Isolation* disallows writes that might otherwise cause corruption.

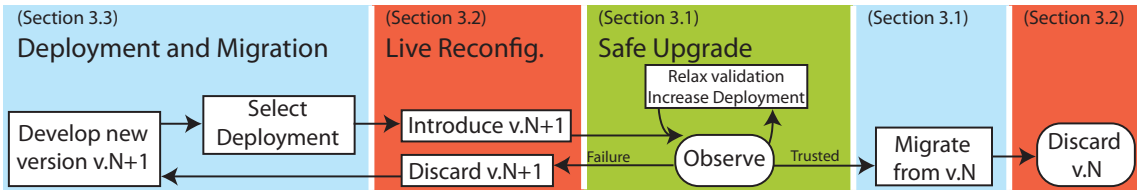

Figure 2: Our work-flow for performing storage service upgrades, categorized by the relevant sections in the text.

3.1.1 Tees

To ensure recoverability after a flawed upgrade, we deploy the upgraded version of the system alongside the existing version and replicate the request stream to both. This mechanism is provided by an interface-specific Tee. In contrast to more conventional N-Version systems like EnvyFS [4] and RAIF [8], Tees compare the results of a trusted system to those of a system under test.

Our block device Tee is implemented in 508 lines of C code as a blktap [14] module. It operates at disk granularity and is capable of supporting any block-based service. We have also implemented a Tee for CouchDB in 904 lines of C that operates at database granularity. Tee modules maintain consistency between versions by mirroring write requests and validate correctness by comparing the data returned from reads. Requests received through auxiliary interfaces (e.g., volume snapshot requests) can also be mirrored by the Tee, if appropriate.

A Similar Tee abstraction has been used in the past to verify heterogeneous implementations of an NFS server [13]. In that work, use in a live setting was discussed but not attempted. Our system is focused entirely on production environments and has been deployed in practice. We extend the technique to allow both the trusted system and the system under test to share a common storage substrate, and allow configurable tradeoffs between the fidelity of integrity checks and the overheads associated with such validation. The Tee is also logically similar to the "parallel universe" created in Imago [7] by mirroring a website to Amazon's cloud-compute service.

3.1.2 Tee Policies

Initially, we will want to observe our upgraded system closely for errors. As we gain confidence in the upgrade, we would then prefer to relax validation efforts in exchange for performance, as shown in panel three of Figure 2. For this reason, a Tee can follow a variety of validation policies. The *Synchronous cross checks* policy stipulates that read requests are mirrored across both versions of the storage system and validated against each other, while write requests are re-read after completion. In the case of a mismatch, the VM can continue under the original version and the error can be reported.

Other protection levels offer progressively better performance with more relaxed validation, as summarized in Table 1. In the *Barrier writes asynchronous* policy, writes to the more trusted disk are delayed until all prior reads are verified. This means that an application may receive bad data, but it will not write to disk as a result. In *Asynchronous with new version primary*, on-disk data could be modified in response to a bug in an upgrade. In *Lazy validate hashes*, hashes of large regions of the disk are periodically computed to catch errors in the background.

Policy	Error Exposure
Synchronous cross checks$^{\checkmark\ \oplus}$ Synchronous read checks$^{\checkmark}$ Async. w/ old version primary$^{\checkmark}$	No errors exposed
Barrier writes async.$^{\checkmark}$	To app. level only
Async. w/ new version primary$^{\checkmark}$ Lazy validate hashes	On-disk and to app.

Table 1: Tee protection levels, $^{\checkmark}$ denotes policies currently implemented for the block-level, $^{\oplus}$ for CouchDB.

3.1.3 Isolation Layer

In our environment, all storage services use a shared block store. This presents a problem, as a flawed software upgrade could follow an invalid block pointer and corrupt data in another service. To address this, we added an access control mechanism similar in spirit, though lighter-weight than Snapdragon [1].

Permissions in our system are <location, key> tuples applied to large extents of the block store. Each version of the system, as identified by the key, works in isolated extents to contain errors. Tuples provide permission to write to the associated extent. They are granted during extent allocation, cached, and checked for every write. Read permissions (to ensure data privacy) are not implemented. Even if such a mechanism were provided, it could not protect privacy within a opaque virtual disk. Therefore, we use validation policies that validate all reads at the Tee level until such a time as we are satisfied with the privacy provided.

3.2 Live Reconfiguration

Safe upgrades would be far less attractive if systems had to be taken off-line for deployment. To remove this concern, both our Tees are capable of reconfiguring their underlying storage services by pausing the request stream, letting outstanding requests complete, and adding or removing services as desired. This works in part because Parallax and CouchDB are largely stateless. The same would not be true of all filesystems. Restartable filesystem support has been considered in Membrane [12], and we consider this to be interesting future work.

3.3 Deployment and Migration

The final concerns we must address are highlighted in the first and fourth panels of Figure 2. Here we wish to deploy our tiering implementation. To fully upgrade we must also migrate existing data to the new system.

A full deployment of data tiering to the entire cluster is possible, but it is likely better to start with a subset of the cluster and gain experience and confidence in the upgrade first. Upgrades operate at the service-level unit of storage (likely disk or database, though we make no hard assumptions) and could be made based on their tolerance of, or need for, the feature. Alternatively they could be sampled randomly. Over time we will expand this group, as shown in the third panel of Figure 2.

After some time under successful observation, the new version of the system will be fully trusted and the old version will be retired. Before this point we will need to migrate existing data from the old version of the service to the new. A copy-out operation provided through the Tee can be invoked at any time to make migration safe and recoverable. Naturally, developers could instead use an in place upgrade tool once the new version is trusted.

4 Evaluation

In this section we establish the overhead of the Tee module and isolation system. We also discuss our experiences with deploying Dovetail on our live storage system and recovering from real errors.

4.1 Handling errors

Our primary concern in deploying Dovetail was to ensure that our in-development system was safe to deploy at the necessary release cycle. Since January 20th, 2010, we have been using Dovetail to provide safe upgrades for a virtual machine hosting service. Our first users were comfortable with placing their data on our prototype because it was mirrored (using Dovetail) onto Ext3 on Linux. We deploy a new set of upgrades from our source-code repository, on average, every 3 weeks, though nothing would prevent us from doing so more frequently. By convention developers run a regression suite prior to any checkin. Of the 46 patches deployed in this manner, 2 have contained flaws that would have affected users, but were instead caught by our Tee validation system.

On January 29th, an upgrade attempt failed quickly after installation. The code defect would have been caught by the test suite, but one of the developers failed to follow the testing policy. On March 4th another software defect caused a failure on one user's disk. In this case it was a rarely seen race condition that depended on an unusual interleaving of requests. It was not triggered until the system had been in use for nearly a month. Rather than rushing a hotfix to correct this issue immediately, we left the flawed version running on all other disks, out of convenience, until the next scheduled upgrade.

4.2 Performance Overhead

Maintaining multiple versions of a storage service necessarily incurs extra I/O operations. The primary goal of this section is to show how validation policies mitigate system load increases. We focus on a worst-case configuration by targeting a single disk and sending every request through our block level Tee.

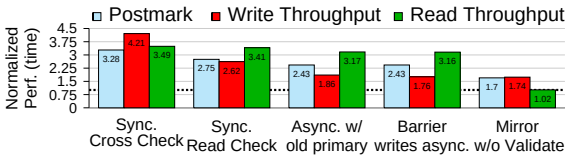

Figure 3: Tee policy overheads

Figure 3 shows the test time for each of Postmark, sequential write, and sequential read workloads under five Tee policies. Results are normalized against a baseline with no Tee or Isolation layer (i.e., a score of 2 means it took twice as long to complete the workload). The first four policies operate as described in Table 1 and show progressively better write performance. The *Mirror without validate* policy shows the overhead of mirroring writes without validating reads. Read performance is dominated by seek overhead, which may be less pronounced on a multi-disk array. The overheads of isolation are minimal. Modifications to access controls are rare (one I/O per 2GB of new write requests) and permission checks can be cached.

These measurements clearly indicate that the increased validation provided by Dovetail is not free. However, we expect these increases will be applied to an

admin-controlled subset of the system, and can be absorbed with appropriate disk provisioning. Further performance gains could be made by buffering writes [11] or using Single Instance mechanisms [4]. Finally, once confidence in a new version has been established, the Tee and all of its attendant overhead can be removed.

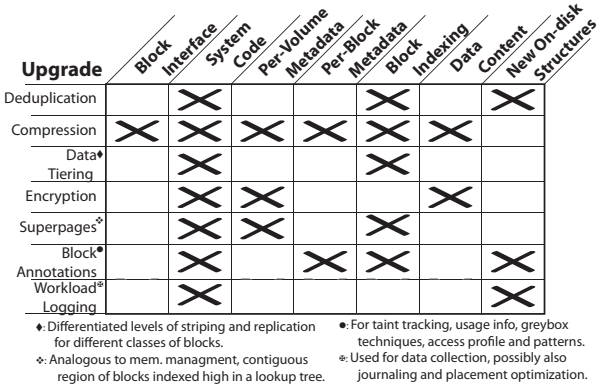

Figure 4: Future Parallax upgrades and their impacts.

5 Conclusion and Future Work

Dovetail can recover from a failed software upgrade of a storage system. Unlike OS-level upgrade recovery mechanisms [15], it can protect against the resulting destruction of on-disk data. Live reconfiguration and migration services ensure that moving between versions is simple and non-disruptive. This framework has helped us shorten our own release cycles, providing a much better match for an in-development storage service.

Our successes with Dovetail have encouraged us to continue to develop and use the system. Figure 4 shows our current upgrade plans for Parallax and their associated impacts. The majority of these changes can be fully protected. However, changes to the storage aggregation or Tee interface require careful consideration. In providing a Tee between Parallax and ext3, we chose to ignore the snapshot requests against the trusted system. In the event of a failure, these snapshots would be lost. Alternately, we could have suffered a full volume copy-out during snapshot.

Much of the run-time overhead of Dovetail could be eliminated by removing redundant I/O with a data deduplication layer [12]. The costs of migration could be amortized by supporting an Upgrade-on-Write mode where modifications to the trusted system would call programmer provided upgrade code and copy the data to the newer system. In practice, this is already a requirement of storage developers looking to modify on-disk formats. Exposing block placement information at the disk aggregation layer would help support firmware and hardware upgrades, by ensuring that modifications only affect a subset of data replicas.

Finally, identifying the sufficient subset of users to evaluate an upgrade, and the correct length of time to observe the system are important areas of future work. As we gain experience with the approach, we hope to develop guidelines and best practices for performing aggressive upgrades of production cloud storage systems.

References

[1] M. K. Aguilera, M. Ji, M. Lillibridge, J. MacCormick, E. Oertli, D. Andersen, M. Burrows, T. Mann, and C. A. Thekkath. Block-Level Security for Network-Attached Disks. In *FAST'03*.

[2] Amazon.com. Amazon simple storage service (Amazon s3). http://aws.amazon.com/s3.

[3] J. C. Anderson, J. Lehnardt, and N. Slater. *CouchDB: The Definitive Guide*. O'Reilly Media, 2010.

[4] L. N. Bairavasundaram, S. Sundararaman, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Tolerating File-System Mistakes with EnvyFS. In *ATEC'09*, San Diego, CA.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI'06*, pages 205–218, 2006.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*

[7] T. Dumitraş and P. Narasimhan. Toward upgrades-as-a-service in distributed systems. In *Middleware'09*.

[8] N. Joukov, A. Rai, and E. Zadok. Increasing distributed storage survivability with a stackable raid-like file system. In *CCGRID'05*, pages 82–89, 2005.

[9] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *ASPLOS '96*, pages 84–92, Cambridge, MA.

[10] D. T. Meyer, G. Aggarwal, B. Cully, G. Lefebvre, M. J. Feeley, N. C. Hutchinson, and A. Warfield. Parallax: virtual disks for virtual machines. In *Eurosys'08*.

[11] H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara. Snapmirror: File-system-based asynchronous mirroring for disaster recovery. In *FAST'02*.

[12] S. Sundararaman, S. Subramanian, A. Rajimwale, A. C. Arpaci-dusseau, R. H. Arpaci-dusseau, and M. M. Swift. Membrane: Operating system support for restartable file systems. In *FAST'10*.

[13] Y.-L. Tan, T. Wong, J. D. Strunk, and G. R. Ganger. Comparison-based file server verification. In *ATEC '05*.

[14] A. Warfield, S. Hand, K. Fraser, and T. Deegan. Facilitating the development of soft devices. In *ATEC'05*.

[15] M. Wise. Windows xp system restore. http://technet.microsoft.com/en-us/library/bb490854.aspx.