RACS: A Case for Cloud Storage Diversity

Hussam Abu-Libdeh
Cornell University
Ithaca, NY 14853
hussam@cs.cornell.edu

Lonnie Princehouse
Cornell University
Ithaca, NY 14853
lonnie@cs.cornell.edu

Hakim Weatherspoon
Cornell University
Ithaca, NY 14853
hweather@cs.cornell.edu

ABSTRACT

The increasing popularity of cloud storage is leading organizations to consider moving data out of their own data centers and into the cloud. However, success for cloud storage providers can present a significant risk to customers; namely, it becomes very expensive to switch storage providers. In this paper, we make a case for applying RAID-like techniques used by disks and file systems, but at the cloud storage level. We argue that striping user data across multiple providers can allow customers to avoid vendor lock-in, reduce the cost of switching providers, and better tolerate provider outages or failures. We introduce RACS, a proxy that transparently spreads the storage load over many providers. We evaluate a prototype of our system and estimate the costs incurred and benefits reaped. Finally, we use trace-driven simulations to demonstrate how RACS can reduce the cost of switching storage vendors for a large organization such as the Internet Archive by seven-fold or more by varying erasure-coding parameters.

Categories and Subject Descriptors

C.2.4 [**Computer Systems Organization**]: Computer Communication Networks—*Distributed Systems*; D.4.3 [**OPERATING SYSTEMS**]: File Systems Management—*Distributed File Systems*; D.4.5 [**OPERATING SYSTEMS**]: Reliability—*Fault-tolerance*; D.0 [**SOFTWARE**]: General—Distributed wide-area storage systems

General Terms

Design, Economics, Reliability

Keywords

Cloud computing, cloud storage, distributed systems, vendor lock-in, erasure codes, fault tolerance

1. INTRODUCTION

Current trends show an increasing number of companies and organizations migrating their data to cloud storage providers [37]. Typical usage examples include storing online users' account data, off-site backup storage, and content distribution, to name a few. In fact, one impressive pilot program and motivating example is a recent announcement by the United States Library of Congress to move its digitized content to the cloud [15]. Other participants in that program include the New York Public Library and the Biodiversity Heritage Library.

However, for a service to depend solely on a particular cloud storage provider has its risks. Even though different cloud storage providers offer nearly identical service commodities, customers can experience *vendor lock-in*: It can be prohibitively expensive for clients to switch from one provider to another. Storage providers charge clients for inbound and outbound bandwidth and requests as well as for hosting the actual data. Thus, a client moving from one provider to another pays for bandwidth twice, in addition to the actual cost of online storage. This doubled cost of moving data leads to a kind of "data inertia"; the more data stored with one provider, the more difficult it becomes to move. This must be taken into consideration by consumers of cloud storage, lest they be locked into less-than-ideal vendors after entrusting them with their data. The resulting vendor lock-in gives storage providers leverage over clients with large amounts of data. These clients are vulnerable to price hikes by vendors, and will not be able to freely move to new and better options when they become available. The quickly-evolving cloud storage marketplace makes this concern more real: A customer's best decision now may leave him trapped with an obsolete provider later, held hostage by vendor lock-in.

In addition to possible increased costs, vendor lock-in subjects customers to the possibility of data loss if their provider goes out of business or suffers a catastrophe. Even though cloud operators abide by strict service-level agreements (SLAs) [3, 8, 13] with impressive up-times and response delays, failures and outages do occur [2, 12]. Further, recent incidents have shown how failures at cloud providers can result in mass data loss for customers, and that outages, though seldom, can last up to several hours [4].

Since cloud storage is for the most part a commodity, one way for customers to guard against vendor lock-in is to replicate their data to multiple providers. This achieves the goals of redundancy and market mobility, but at a very high storage and bandwidth cost. A more economical approach is to spread the data across multiple providers and introduce redundancy to tolerate possible failures or outages, but stopping short of replication. This is indeed very similar to what has been done for years at the level of disks and file systems. RAID 5, for example, stripes data across an array of disks and maintains parity data that can be used to reconstruct the contents of any individual failed disk. In this paper we argue that similar techniques can be carried over to cloud storage, although the failure model differs somewhat.

SoCC'10, June 10–11, 2010, Indianapolis, Indiana, USA.

In this paper, we describe RACS (Redundant Array of Cloud Storage), a cloud storage proxy that transparently stripes data across multiple cloud storage providers. RACS reduces the one-time cost of switching storage providers in exchange for additional operational overhead. RACS assumes a minimal storage interface (`put, get, delete, list`) and exposes that interface to its applications. In designing and evaluating this system we make several contributions. First, we show through simulation and up-to-date pricing models that, through careful design, it is possible to tolerate outages and mitigate vendor lock-in with reasonable overhead cost. Second, we show through trace-driven simulations the cost of switching cloud storage vendors for a large organization. Third, we build and evaluate a working prototype of RACS that is compatible with existing Amazon S3 clients and able to use multiple storage providers as back-ends, and we demonstrate its effectiveness on end-user traces.

In the remainder of this paper, we first motivate why clients should diversify their storage providers and give a brief overview of existing data striping and erasure coding techniques in section 2. In section 3, we describe our design followed by a discussion of our trace-based simulation results in section 4 and prototype evaluation in section 5. Section 6 gives an overview of related work. In Sections 7 and 8, we conclude with a discussion of the implications of our work and discuss how this research agenda might be further explored.

2. WHY SHOULD WE DIVERSIFY?

Hosted data storage is a commodity, and it is fungible. Certainly, cloud storage providers *do* distinguish themselves by offering services above-and-beyond basic storage. Often, these services involve integration with other cloud computing products; for example, Amazon's EC2 and Cloudfront are deeply intertwined with S3. But many consumers just want reliable, elastic, and highly available online storage. These terms apply to most cloud storage offerings, to the point that storage services are effectively interchangeable, in spite of cosmetic differences. Because of this, cloud storage providers compete aggressively on price, knowing that this will be the ultimate deciding factor for many consumers. They also compete on guarantees of uptime and availability, in the form of SLAs. These guarantees are upheld by systems meticulously designed to handle load balancing, recurring data backup, and replication to seamlessly handle component failures. So one would understandably wonder if there's really any benefit to be had by adding even more redundancy to the system and spreading data across multiple providers.

To address this, we distinguish two different kinds of failures:

Outages A series of improbable events leads to data loss in the cloud. As mentioned, cloud providers already offer strong protection against component failures, so there is a thin case for adding another failsafe on top of cloud storage systems. Still, it does happen occasionally that an outage strikes the cloud: For instance, a failure in October 2009 at a Microsoft data center resulted in data loss for many T-Mobile smart phone users [23]. Not to mention that the reliability of the cloud as a long-term storage medium is untested [35].

Economic Failures We define **economic failure** to be the situation where a change in the marketplace renders it prohibitively expensive for a cloud storage consumer to continue to use a service. There is often some advance warning of economic failure; for example, at the time of writing this paper, there is no bandwidth charge for uploading data to Amazon S3. But on June 30, 2010, Amazon will start charging fifteen cents per gigabyte. A customer who relies on a heavy volume of uploads, such as an online backup business, might find this very expensive. Other examples include a service going out of business.

In this paper, we are primarily concerned with economic failure.

2.1 Avoiding Storage Vendor Lock-in

Cloud storage services differ in pricing scheme and performance characteristics. Some providers charge a flat monthly fee, others negotiate contracts with individual clients, and still others offer discounts for large volume or temporary promotional incentives or lower bandwidth rates for off-peak hours. Some providers may be desirable for geographic reasons (Amazon offers different zones for the United States and Europe), and others may offer extra features such as publishing files directly to the web or access via mountable file system. Changes in these features, or the emergence of new providers with more desirable characteristics, might incentivize some clients to switch from one storage service to another. However, because of storage inertia, clients may not be free to choose the optimal vendor due to prohibitively high costs in switching from one provider to another. This puts the client at a disadvantage when a vendor raises prices or negotiates a new contract.

As an example, consider a situation where a new provider with a low pricing scheme emerges on the market. A client at an old provider has to make an all-or-nothing decision about his storage service. He either moves all his data to the new provider and incurs the cost of switching to gain the benefit of lower prices, or he stays at the old provider and saves the cost of switching at the risk of incurring higher costs in the long run. The problem, of course, is that the client can not tell if and when his current provider will lower the pricing scheme to compete with the other providers'. From the client's point of view, a premature switch can be worse than no-switch, and a late switch might have lost its value. Of course, this is not limited to price changes. Clients might be stuck with underperforming providers simply because switching to better providers is deemed too costly.

One way to allow clients to be more agile in responding to provider changes is by making the switching and data placement decisions at a finer granularity than a global all-or-nothing. This can be achieved by striping the client's data across multiple providers. For example, if a client spreads his data across several providers, taking advantage of a new fast provider entails switching only a fraction of the data, which could be more economically feasible.

Of course, striping data across multiple providers entails added costs due to the difference in pricing between the different providers. However, as we'll show in our cost estimates, these differences are low and can be mitigated by choosing clever striping schemes.

2.2 Incentives for Redundant Striping

We have argued that, by striping data across multiple providers, clients can be more agile in responding to provider changes. Now we argue that by adding redundancy to data striping, clients can maintain this mobility while insuring against outages.

We use erasure coding to create this redundancy. After erasure coding, we will be able to tolerate the loss of one or more storage providers without losing data—and do so without the overhead of strict replication [18, 22, 33, 39]. Erasure-coding maps a data object broken into m equal-size original fragments (shares) onto a larger set of n fragments of the same size ($n > m$), such that the original fragments can be recovered from any m fragments. The fraction of the fragments required for reconstruction is called the rate, denoted r, with $r = \frac{m}{n} < 1$. A rate r erasure-code increases the storage cost by a factor of $\frac{1}{r}$. For example, an $r = \frac{1}{2}$ encoding might produce $n = 8$ fragments, any $m = 4$ of which are sufficient to reconstruct the object, resulting in a total overhead factor of $\frac{1}{r} = \frac{n}{m} = \frac{8}{4} = 2$. Note that $m = 1$ represents strict replication, and RAID level 5 [30] can be described by ($m = 4$, $n = 5$). Generally, there are three types of erasure-codes[1]: optimal, near optimal, and rateless. But we only consider optimal erasure-codes in this works since n is usually fairly small. Data loss occurs when all replicas or a sufficient fraction of fragments for erasure-codes (more than $n-m$ fragments) are lost due to permanent failure.

Below is a brief description of what clients can gain from erasure coding across multiple providers:

Tolerating Outages. Although storage providers offer very highly available services, experience tells us that even the best providers suffer from occasional mishaps. Inopportune outages can last for several hours [2, 4, 12] and can result in severe financial losses for the providers and consumers of the storage service. Adding data redundancies across multiple providers allows clients to mask temporary outages and get higher data availability.

Tolerating Data Loss. Storage providers implement internal redundancy and replication schemes to avoid accidental data loss. However, recent experience shows that hosted data could be lost [23]. Intuitively, storing redundant copies of the data with multiple providers can allow consumers to mask data loss and failure at individual providers.

A simple way to tolerate provider failure is by fully replicating data to multiple providers. However, this approach is costly since the redundancy overhead is 100% the size of the data. A more economically sensible approach is to lower the redundancy overhead by using erasure coding techniques and stripe the data and redundant blocks across multiple providers to tolerate the failure of one or more of them.

Adapting to Price Changes. Erasure coding across multiple providers allows clients to make migration decisions at a lower granularity. For example, clients can take advantage of a provider's price reduction by favoring the provider to serve in different quorums. If a provider becomes too expensive to use, clients can avoid reading from it and reconstruct its data from the added redundancy at other locations.

Adapting to New Providers. If a new service provider enters the market, clients can include it in future erasure codes.

Control Monetary Spending. By using a mixture of online providers, second tier storage services, and local servers, clients can control the cost of storing and serving data. Clients can bias data accesses to cheaper options to reduce the overall cost of storage.

Choice in Data Recovery. In the event of a failure of one or more online storage providers, the added redundancy allows clients to choose amongst multiple strategies for recovering the lost data. For example, clients might choose to reconstruct the missing data aggressively from other providers. Another approach is to reconstruct the data lazily by reconstructing missing data upon client requests. Or, if the degree of redundancy is sufficient, clients might even choose to ignore the failed provider and continue their operation without reconstruction.

The main idea is that if clients were to use a single provider, they would not be able to tolerate the provider's failure. Replicating data to multiple providers allows clients to tolerate failures, but comes at a higher cost. Erasure coding the data across multiple providers allows clients to tolerate provider failures and the extra redundancy allows clients to implement different strategies and optimize for different objectives in storing and accessing their data.

[1]Optimal erasure codes such as Reed-Solomon [16, 31, 34] codes produce $n = m/r$ ($r < 1$) fragments where any m fragments are sufficient to recover the original data object. Unfortunately optimal codes are costly (in terms of memory usage, CPU time, or both) when m is large, so near optimal erasure codes such as Tornado codes [25, 26] are often used. These require $(1+\varepsilon)m$ fragments to recover the data object. Reducing ε can be done at the cost of CPU time. Alternatively, rateless erasure codes such as LT [24], Online [27], or Raptor [36], codes transform a data object of m fragments into a practically infinite encoded form.

put	*bucket, key, object*
get	*bucket, key*
delete	*bucket, key*
create	*bucket*
delete	*bucket*
list	keys in *bucket*
list	all buckets

Table 1: Amazon S3 operations

3. DESIGN

RACS mimics the interface of Amazon S3, and uses the same data model. S3 stores data in named **buckets**. Each bucket is a flat namespace, containing **keys** associated with **objects**. A bucket cannot contain other buckets. Objects can be of arbitrary size, up to 5 gigabytes. Partial writes to objects are not allowed; they must be uploaded in their entirety, but partial reads are allowed. We chose Amazon S3's interface for RACS for two reasons: First, its no-frills simplicity is easy to work with, and second, its popularity means that we will be able to use existing client applications with RACS. To describe the design of RACS, we will discuss how it implements a core subset of S3's operations, shown in Table 1.

RACS presents itself as a proxy interposed between the client application and a set of n **repositories**, which are cloud storage locations ideally hosted by different providers. RACS operates transparently, allowing use of unmodified S3 clients. Upon receiving a **put** request, RACS splits the incoming object into m **data shares** of equal size (i.e. each share is $1/m$ of the original object size), where $m < n$ are configurable parameters. RACS then uses erasure coding to create an additional $(n-m)$ **redundant shares**, for a total of n shares. Redundant shares are the same size as data shares. Any subset of m shares is sufficient to reconstruct the original object. Each share is sent to a different repository.

Conversely, when a client issues a **get** request, RACS fetches m shares and reassembles the data. Metadata such as bucket and key names, modification times, and MIME types are replicated across all servers; these data are relatively small and replication is not prohibitive unless the workload is dominated by very small objects. RACS also stores a small amount of its own metadata with each share, including the size of the original object and its content hash. This allows **list** requests, which return information including size and MD5 sum for every key in a bucket, to be fulfilled without querying multiple repositories or reconstructing coded objects.

Figure 1 illustrates the RACS architecture with a single proxy.

3.1 Distributed RACS

Because all data must pass through a RACS proxy to be encoded and decoded, a single RACS proxy could easily become a bottleneck. To avoid this, RACS is intended to be run as a distributed system, with many proxies concurrently communicating with the same set of repositories. RACS proxies store a limited amount of

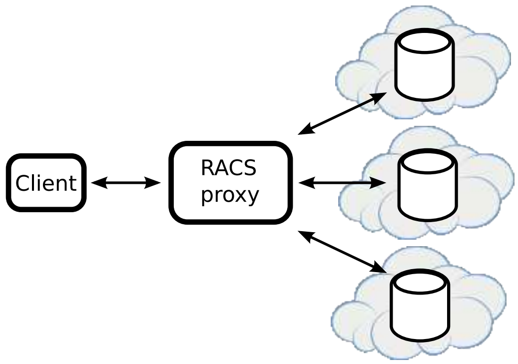


Figure 1: RACS single-proxy architecture

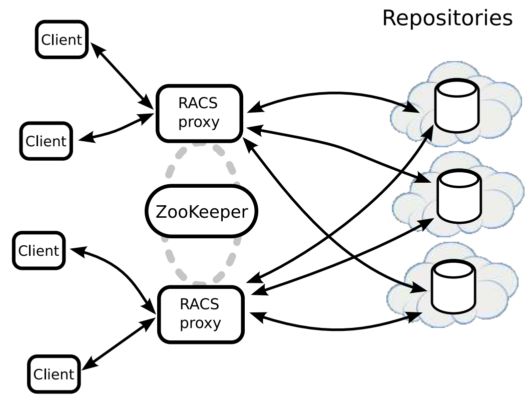


Figure 2: Multiple RACS proxies coordinate their actions using ZooKeeper

state: User authentication information and the location and credentials for each repository. Changes to this state are broadcast to each proxy. Distributing shares of data to multiple repositories introduces a kind of data race that is not native to S3 semantics. If two clients simultaneously write to the same key using Amazon S3, Amazon will process the writes in the (arbitrary) order it receives them, and the later write will trump the earlier. If the same two writes are issued to two different RACS proxies, this race will be played out at each of the n repositories, with possibly catastrophic results. Any two repositories for which the outcome of the race differs will become inconsistent, and RACS would have a difficult time determining which of the two objects is the correct one. Worse, there may not be m shares of either object in the system, leading to data loss. A related problem occurs when a read and write happen concurrently: The read may return some combination of old and new shares. To prevent these races, RACS proxies coordinate their actions with one-writer, many-reader synchronization for each *(bucket,key)* pair. RACS relies on Apache ZooKeeper [14] for distributed synchronization primitives. ZooKeeper is a distributed system that provides atomic operations for manipulating distributed tree structures; it can be used to implement synchronization primitives and abstract data types (ADTs). Such synchronization primitives cannot be built using only S3, as it lacks strong consistency guarantees.

Figure 2 illustrates the RACS architecture with multiple distributed proxies.

3.2 Failure Recovery

Economic failures such as price increases are often known ahead of time. Administrators of a RACS system can begin a **migration** away from a repository even before the failure occurs, in a sort of pre-emptive failure recovery. During a migration, RACS moves the redundant shares from the soon-to-fail repository to a fresh repository. While migrating, RACS does not use the failed repository to serve **get** requests unless other failures have occurred. **put** requests are redirected from the failed repository to the new repository. This entails download and then upload of $1/m$ of the total object data, a major improvement over replication. Unpredictable failures carry a higher price of recovery: RACS must reconstruct the redundant shares from other repositories. This is still preferable to replication, though, as only $1/m$ of the total object data is uploaded. Depending on its configuration, RACS may interpret unpredictable failures as transient, and continue normal operation in the hope that failed repository returns. Operations such as **put** and **delete** that modify data during such an outage will cause the failed repository to become out-of-sync with the others; we leave the question of how to recover a repository from a transient outage for future work.

3.3 Policy Hints

RACS is compatible with unmodified client applications. However, a RACS-aware client can include in its requests hints about how RACS should behave. Many RACS operations involve a choice about which repositories to communicate with, and in which order. Using *policy hints*, a client can specify preferred repositories. This has many potential uses: Exploitation of geographic proximity, navigation of variable pricing schemes (such as off-peak discounts), load-balancing, or even load-*un*balancing to favor inexpensive repositories. Further, trade-offs may be made to favor bandwidth over latency or vice versa. For example, a **list** operation normally queries only a single repository, but clients who value latency above all else might choose to concurrently query all repositories and take the quickest answer.

3.4 Repository Adapters

It would be unrealistic to expect cloud storage providers other than Amazon to offer the same REST interface as S3. RACS does not require this. Instead, adapters are written that wrap a storage provider's API in S3-like semantics that can be used by RACS. For some providers, such as Rackspace, there is a direct correspondence. For others, such as network mounted filesystems, a more complicated mapping must be devised.

3.5 Performance Overhead

The primary goal of RACS is to mitigate the cost of vendor lock-in by reducing the importance of individual storage providers. In exchange, RACS incurs higher overhead costs as follows:

Storage RACS uses a factor of n/m more storage, plus some additional overhead for metadata associated with each share. Since storage is split among different providers, some of whom may have higher rates, the total price paid for storage may be even greater than n/m times the cost of using only the least expensive provider.

Number of requests RACS issues a factor of n more requests to repositories for **put** , **create** , and **delete** operations than the client would using S3 directly. **get** requests are multiplied by m. **list** operations do not use any additional requests by default.

Bandwidth RACS increases the bandwidth used by **put** operations by a factor of $\frac{n}{m}$, due to the redundant shares. **get** requests do not suffer a commensurate bandwidth increase. **list** uses no additional bandwidth by default. However, each operation does incur a slight increase in bandwidth due to the extra number of requests per operation and the size of the header for each request; this may be significant if stored objects are relatively small.

Latency RACS may introduce slightly more latency for **put** requests, as **put** operations must wait for the slowest of the repositories to complete the request. On the other hand, operations such as **list** that require only one answer can optionally query all repositories in parallel and return as soon as the first response is received. However, there is an inherent tradeoff between bandwidth and latency. **get** occupies a middle-ground: Depending on which subset of m repositories is queried, latency could be better than the average of all repositories. Erasure coding also introduces latency, as the proxy must buffer blocks of data to encode or decode. Coordination with ZooKeeper is another source of latency, although it is expected to be little more than the round-trip time to ZooKeeper except in the presence of object contention.

3.6 RACS Prototype Implementation

Our RACS prototype is implemented in approximately 4000 lines of Python code. We implement Amazon S3's REST interface, which exposes S3 as a web-based service. It is simpler and more popular than the more powerful S3 SOAP interface. Most existing S3 REST clients can be configured to communicate with S3 through an HTTP proxy; this is generally used to bypass firewalls. By setting RACS as a proxy, S3 clients can take advantage of RACS without modification. It is relatively easy to map a basic data model onto others; we might have instead chosen to have RACS present itself as a network file system, but using file system semantics would have added considerable complexity to the implementation. Our prototype does not yet implement the complete S3 interface: It has no provisions for authentication or payment-related queries. It also does not implement RACS policy hints, although a static configuration option can be set to prioritize bandwidth or latency. ZooKeeper is used to implement readers-writer locks at per-key granularity for distributed RACS proxies.

The design for RACS calls for repositories backed by many different storage providers. Our implementation supports three kinds of repositories: S3-like (including S3 and Eucalyptus [28], an open-source Amazon Web Services clone), Rackspace Cloud Files [11], and a file system mapping that can be used for mounted network file systems. The Rackspace repository adapter is 258 lines of Python code, and the file system adapter is 243. Both were written in a matter of hours; we expect this to be typical of the ease of adding new kinds of repositories.

Before evaluating our prototype, we will first estimate the vendor lock-in cost associated with switching storage providers. We will then estimate the cost of avoiding vendor lock-in by using RACS.

4. AN INTERNET ARCHIVE IN THE CLOUD

One of our primary motivating examples is the recent decision by the Library of Congress and a handful of other American public libraries to move their digitized content to the cloud [15]. A pilot program of this initiative is being developed by the non-profit DuraSpace organization [5]. Publicly released documents, along with personal conversations with the program directors, reveal that the project (titled DuraCloud) entails replicating the libraries' data across multiple cloud providers to safeguard against sudden failure. We believe that RACS' data striping approach is more fit for the task because it minimizes the cost of redundancy and allows a large organization to switch cloud storage providers without incurring high costs.

We used a trace-driven simulation to understand the costs associated with hosting large digital libraries in the cloud. Our trace covers 18 months of activity on the Internet Archive [9] (IA) servers. The trace represents HTTP and FTP interactions to read and write various documents and media files (images, sounds, videos) stored at the Internet Archive and served to users. We believe this trace is a good reflection of the type of workloads induced on online digital library systems, both in terms of the file sizes and request patterns.

Our goal from using this trace is to answer the following high-level questions:

- What are the estimated costs associated with storing library-type content such as the Library of Congress and the Internet Archive on the cloud?
- What is the cost of changing storage providers for large organizations? What is the added cost of a DuraCloud-like scheme?
- What is the cost of avoiding vendor lock-in using RACS? And what is the added cost overhead of using RACS?

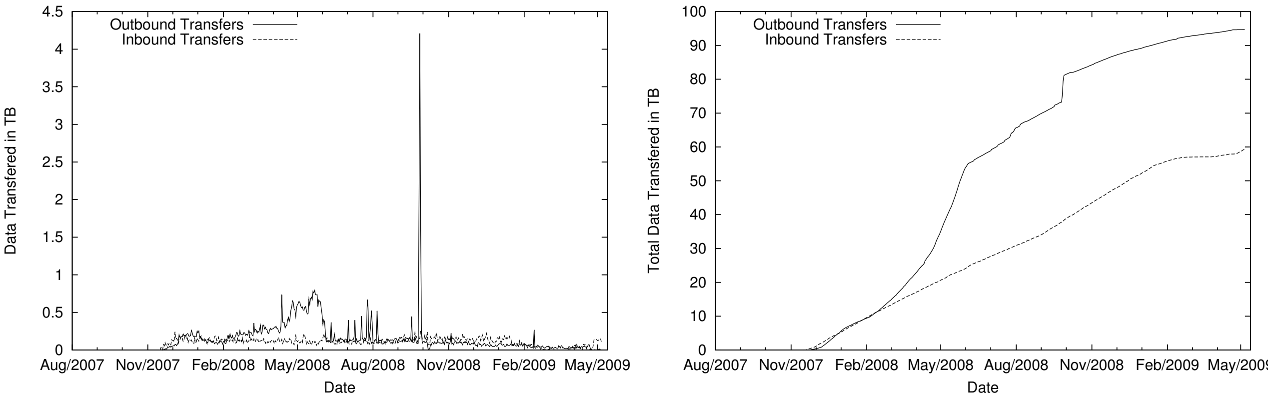
4.1 Trace Characteristics

The Internet Archive trace covers the period from November of 2007 to May of 2009. Figure 3 shows the amounts of data written to the Internet Archive servers and read back from it. The volume of data transfers is dominated 1.6:1 by reads to writes. Figure 4 shows the number of read and write requests issued to the Internet Archive servers during that period. Read requests are issued at a 2.8:1 ratio compared to write requests. In our simulation, we assume that the cloud starts empty; that is, it is not preloaded with any data before the simulation begins.

4.2 Cost of Moving to The Cloud

We estimated the monetary cost of moving the Internet Archive data to the cloud using up-to-date pricing schemes of the leading public cloud storage providers and the trace described in Section 4.1. Storage providers offer bracket-based pricing schemes depending on the amount of data stored and bandwidth consumed in transfers. Table 2 lists a simplified view of the pricing scheme for leading cloud storage providers [1, 6, 7, 10, 11]. In addition to bandwidth and storage, providers exposing a REST API also charge per operation. The full details of the different pricing schemes are not shown in the table for brevity, however we used a detailed model in our simulation.

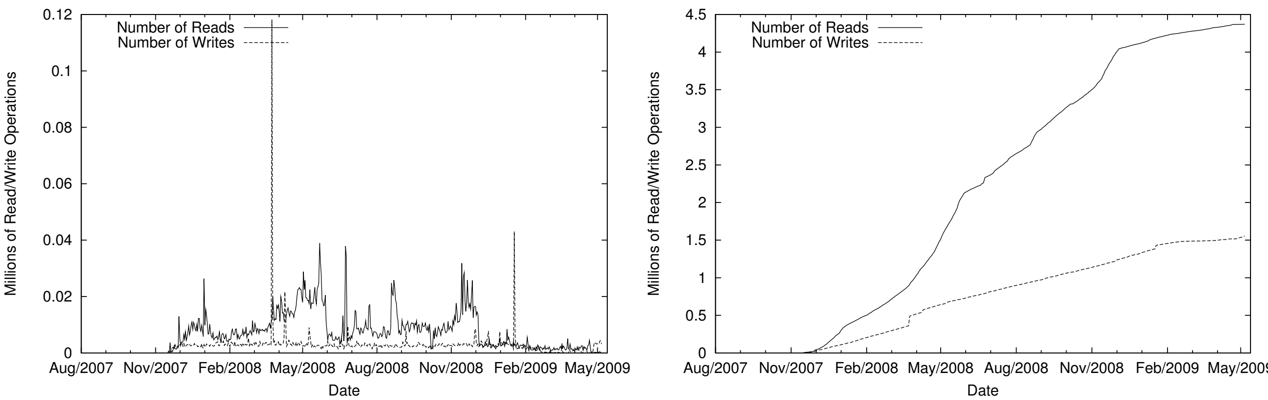
In Figure 5, we estimated the cost of servicing the Internet Archive by using a single storage provider, the DuraCloud scheme of full replication to two providers, and RACS using multiple erasure coding configurations. There are three key takeaways. First, the pricing schemes of the different storage providers are very comparable with one another. Hosting Internet Archive in the cloud with a single provider costs between \$9.2K to \$10.4K per month. Second, using full-replication with this workload roughly doubles the hosting costs and is less ideal than using an erasure coding striping technique as with RACS. Finally, the added overhead cost of RACS depends on the coding configuration. Competitive cloud storage providers have SLAs with 99% and 99.9% monthly up-time percentages, thus we believe that adding enough redundancy



(a) Individual IA data transfers

(b) Cumulative IA data transfers

Figure 3: Inbound and outbound data transfers in the Internet Archive trace



(a) Individual IA read/write requests

(b) Cumulative IA read/write requests

Figure 4: Read/write requests in the Internet Archive trace

to tolerate one provider outage or failure at a time will be sufficient in most cases. In such situations, it is ideal to spread the stripes across many providers to lower the overhead costs. In our calculations, we stripe data across up to nine providers while tolerating one provider's failure. While this might not be possible today due to the limited number of public cloud storage providers, we believe it is a valid estimate in anticipation of expected growth in this popular market. However, we also calculated the associated costs with modest striping schemes with 7 and 5 providers which are possible in today's market.

Striping data across 9 providers while tolerating 1 provider's failure has an added overhead cost of roughly $1000 a month, just under 11% of the original monthly cost. Figure 6 shows the breakdown of the cloud storage cost in four configurations, a single provider, the DuraCloud system, and with RACS using 5 and 9 repositories. In all cases, the monthly bill is dominated by the storage costs. Also, outbound traffic costs are higher than inbound traffic costs. The dominance of storage costs explains why a simple full replication scheme as proposed in DuraCloud is very costly. It also explains why striping cloud storage across a large number of providers so as to limit the overhead added to each individual provider's share is preferable.

4.3 Cost of Vendor Lock-in

One of our main arguments in this paper is that cloud storage clients might be locked-in to particular vendors due to prohibitive switching costs. To quantify the cost of vendor lock-in, we assumed that the Internet Archive is currently hosted on a single reasonably priced provider from the list in Table 2 (in this case S3 Northern California) and that they wish to switch to another leading competitor that is slightly cheaper than the original (Rackspace).

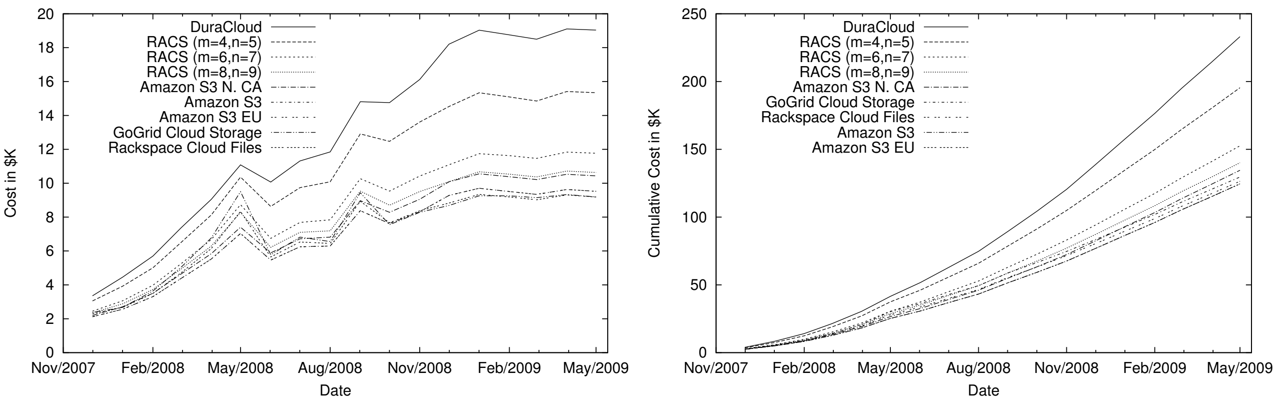
Figure 7(a) shows the long term financial gain if the switching were to happen at different months during the trace period. Comparing purely based on the pricing schemes of the two competitors, it would seem that switching to the cheaper provider is always profitable, and that switching from the onset would yield a maximum benefit of $10K (solid line). However, when we account for the actual cost of switching the potential financial benefit is quickly eroded, and at the end of the trace, changing vendors becomes more than twice as costly as the maximum benefit that would have been attainable in the beginning (dashed line).

We calculated the estimated cost of making the switch at the end of each month during the trace period. For comparison, we also simulated a RACS setup with three striping configurations using providers with comparable prices, and we estimated the cost of migrating between our choice of providers from within RACS. The cost of switching vendors was estimated by calculating the cost of

	S3 USA,EU	S3 N. CA	Rackspace	GoGrid	Nirvanix	EMC Atmos
Data transfer in (GB)	$0.10	$0.10	$0.08	–	$0.18	$0.10
Data transfer out (GB)	$0.15	$0.15	$0.22	$0.29	$0.18	$0.25
Storage (GB/month)	$0.15	$0.165	$0.15	$0.15	$0.25	$0.15
put and **list** requests (per 1000)	$0.01	$0.011	**	–	–	–
get and other requests (per 10000)	$0.01	$0.011	–	–	–	–

(– no charge. **put requests free for objects above 250kB. **delete** requests are free)

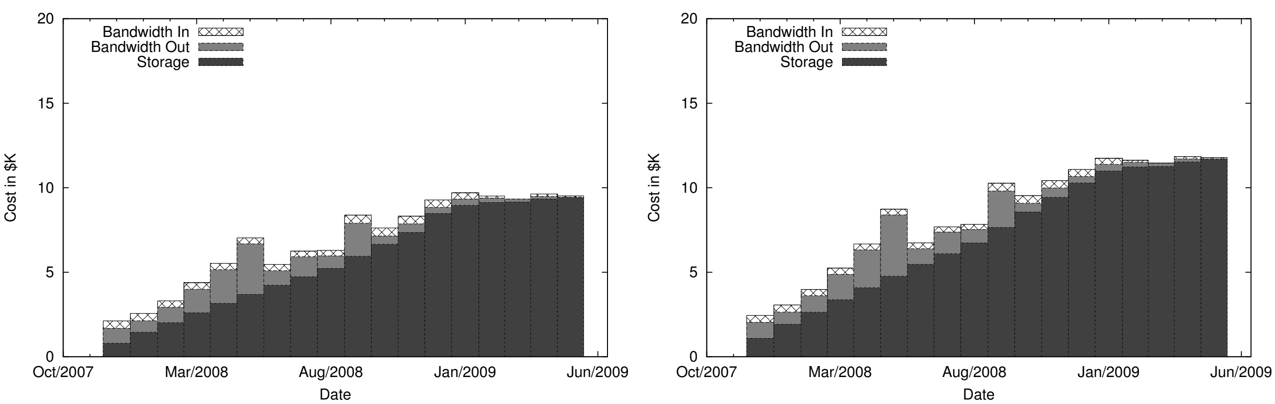
Table 2: Simplified pricing schemes of different cloud storage providers.



(a) Monthly costs with different storage providers

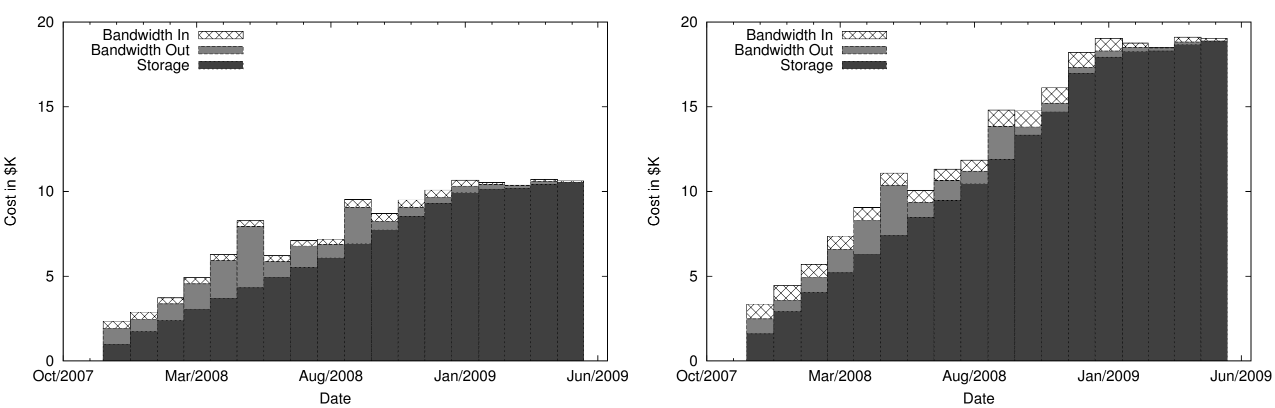
(b) Cumulative costs with different storage providers

Figure 5: Estimated monthly and cumulative costs of hosting on the cloud



(a) Monthly costs breakdown with Amazon S3

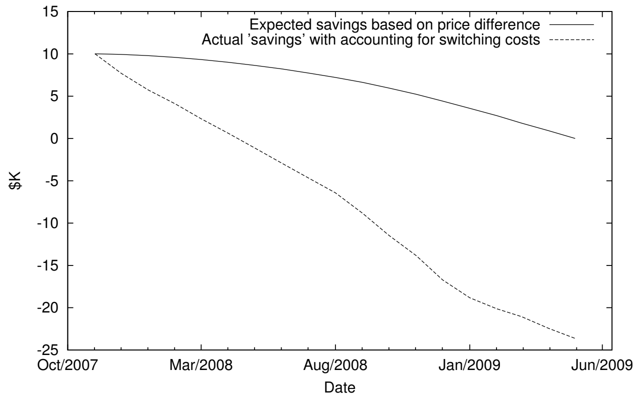
(b) Monthly costs breakdown with RACS(m=4,n=5)



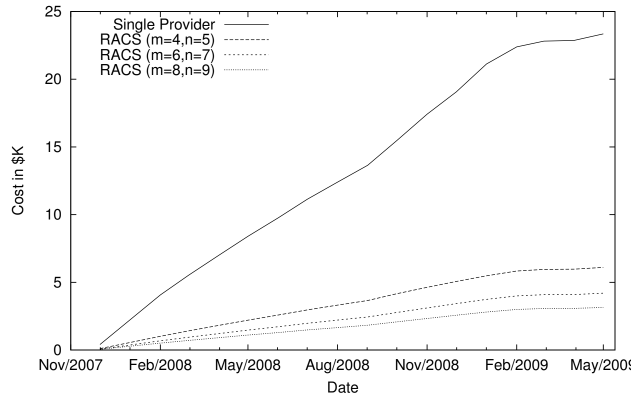
(c) Monthly costs breakdown with RACS(m=8,n=9)

(d) Monthly costs breakdown with DuraCloud

Figure 6: Breakdown of cloud storage costs



(a) Month-by-month switching benefit (non-RACs solution)



(b) Month-by-month switching costs for various configurations

Figure 7: The cost of switching the Internet Archive's storage provider

taking the data out of the old provider and writing it to the new provider. Figure 7(b) shows the switching costs at each month for all the configurations.

Figure 7 highlights the cost of switching vendors, which are prohibitive if using a single provider, but manageable with RACS. Even though Figure 5 shows that the monthly cost using cloud storage for this trace to be just below \$10K, the cost of switching vendors continuously grows to exceed \$23K at the end of the trace period for a single provider (Figure 7(b)). The cost of switching increases as time goes by due to the increase in the size of the data stored at the provider as shown in Figure 6. This implies that the longer an organization stays with a more expensive cloud provider, the more costly it will be for them to switch to another vendor down the road. Alternatively, Figure 7(b) shows that by striping data across multiple vendors, the cost of switching providers in RACS is severely reduced to just below \$3K at the end of the trace period, a factor of seven reduction in the cost to switch.

4.4 Tolerating Price Hikes

In this section, we analyze tolerating increases in price without switching storage providers. We assume a hypothetical scenario where a single provider doubles its pricing scheme halfway through the trace period. In such a scenario, RACS can be used to serve read operations from cheaper providers.

Figure 8 shows the effects of a price hike on hosting on a single provider and using multiple RACS configurations. As expected, striping data across a larger number of providers results in the greatest dampening of the hike. Even an $(m=4, n=5)$ coding, which is possible today, reduces the effects of a potential drastic price hike. Coupled with the low cost of switching vendors as shown in Figure 7(b), clients can use RACS to insure themselves against potential mischievous behavior from their storage providers. As a result, RACS gives more control back to the client to switch providers when they see fit, and protects them from (economic) failures.

5. PROTOTYPE EVALUATION

To evaluate our RACS prototype implementation, we ran several benchmarks with various (m, n) values to determine the operational overhead incurred by RACS. We also tested the response time of RACS against Rackspace to confirm that RACS does not introduce unacceptable latency.

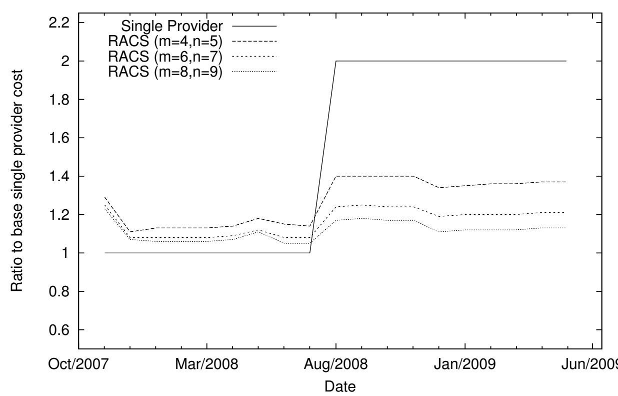


Figure 8: Tolerating a vendor price hike

5.1 Benchmarks

Our benchmarks use the backup tool Cumulus [38] to back up a user's home directory to the cloud. They were run using a single RACS proxy running on the client machine. All repositories were backed by Amazon S3 to obtain an apples-to-apples comparison between operation counts (in real-world usage, of course, using the same storage provider for every repository negates the purpose of RACS). ZooKeeper was disabled, since there was only a single client. Although we used S3 for the benchmarks, we also estimate the cost of using Rackspace repositories instead, based on measured bandwidth from S3. Note that, although S3 objects can be copied inside of S3 without incurring bandwidth charges, we did not do this—to move objects between repositories, we downloaded and re-uploaded them.

The benchmarks are as follows:

Upload snapshot The Cumulus backup tool stores backup snapshots on Amazon S3. Cumulus packs files into large segments to reduce the number of requests sent to S3, and compresses the segments before uploading them. Our experiment used Cumulus to take a snapshot of workstation home directories totaling about four gigabytes. After compression, the snapshot consisted of 238 segments ranging from four to six megabytes each, for a total of 1.2 gigabytes. Cumulus uploads each segment

	No RACS	(2,3)	(4,5)	(5,7)
put and **list** requests	241	731	1209	1747
get requests	240	485	485	487
Data Transfer In (MB)	1199	1811	1486	1656
Data Transfer Out (MB)	20	34	34	28
S3 One-time cost (USD)	$0.12	$0.19	$0.16	$0.18
RS One-time cost (USD)	$0.10	$0.15	$0.12	$0.14
Monthly cost (USD)	$0.18	$0.27	$0.22	$0.24
One-time cost (rel.)	1	1.55	1.33	1.5
Monthly cost (rel.)	1	1.51	1.24	1.38

Table 3: Upload Snapshot Benchmark. Amazon S3 and estimated Rackspace (RS) costs. Monthly cost is the same for both.

	No RACS	(2,3)	(4,5)	(5,7)
put and **list** requests	4	4	4	4
get requests	243	482	972	1215
Data Transfer In (MB)	28	30	30	31
Data Transfer Out (MB)	1191	1235	1263	1210
S3 Cost (USD)	$0.20	$0.21	$0.21	$0.21
RS Cost (USD)	$0.26	$0.27	$0.27	$0.26
S3 Cost (rel.)	1	1.04	1.06	1.02

Table 4: Restore Snapshot Benchmark. Amazon S3 and estimated Rackspace (RS) costs.

Vendor Migration With the backup snapshot already loaded into the cloud, we instruct the RACS server to migrate the entire contents of one repository to a new repository. This simulates the scenario of leaving a vendor in the case of economic failure or new opportunity.

Restore snapshot Retreives all segments for a particular snapshot from the cloud storage repository (i.e. use RACS to **get** each segment), then unpacks each segment locally into files to give a user a file system view of the snapshot.

For each run, we collect the metrics used to determine prices for Amazon S3 USA (see Table 2), and estimate the cost of the trial.

Table 3 shows the relative cost of running RACS for a Cumulus backup compared to a baseline without RACS. RACS issues many more requests, but the cost of this benchmark is dominated by large uploads. There are no great surprises with this benchmark; we expect RACS uploads to use a factor of n/m greater bandwidth and storage than the baseline, and the total cost reflects this. It is of interest, though, to see how much more expensive the three RACS trials are compared to the baseline, with $(m = 4, n = 5)$ being the most feasible. Obviously it is desirable to bring n/m as close to one as possible. Using only Rackspace repositories would have been less expensive for this trial because Rackspace charges $0.08 per GB upload as opposed to Amazon's $0.10. We did not collect data for incremental backup snapshots, but we would expect them to adhere to the same costs relative to the baseline as the full snapshot upload. One side-effect of RACS appears to be improved bandwidth saturation of a client that issues serial **put** requests. We observed this with Cumulus, which had dips in network utilization between sending objects.

In Table 4, we see the results of downloading the backup snapshot from the cloud. This consists almost exclusively of **get** requests, which we expect to consume roughly the same bandwidth as the baseline plus additional overhead due to the multiplicative increase in the number of requests. It is clear that RACS is much more flexible for applications that favor **get** requests.

	No RACS	(2,3)	(4,5)	(5,7)
put and **list** requests	247	247	247	247
get requests	243	241	243	243
Data Transfer In (MB)	1211	610	306	244
Data Transfer Out (MB)	1214	618	316	242
S3 Cost (USD)	$0.32	$0.16	$0.09	$0.07
RS Cost (USD)	$0.36	$0.18	$0.09	$0.07
S3 Cost (rel.)	1	0.51	0.26	0.21

Table 5: Vendor Migration Benchmark. Amazon S3 and estimated Rackspace (RS) costs.

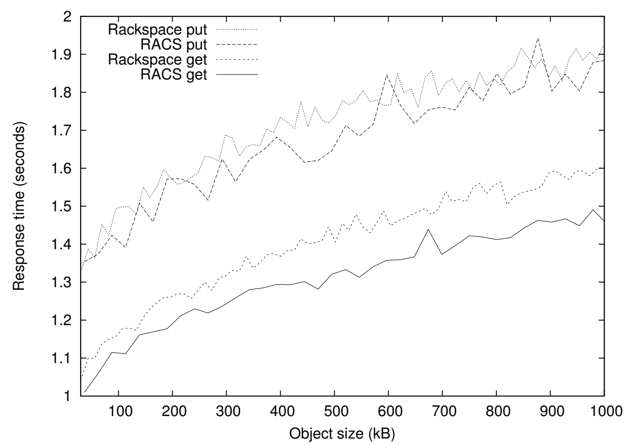


Figure 9: How long it takes RACS and Rackspace to handle object put and get requests, as a function of object size. All response times averaged over four runs.

Table 5 confirms that RACS does indeed reduce the cost of vendor migration, giving consumers greater mobility in the marketplace and more leverage with their providers. The cost of switching providers under RACS is roughly $1/m$ that of the baseline. Note that the number of requests is not increased because RACS is simply copying data from one repository to another.

5.2 Performance

Figure 9 compares the responsiveness of RACS to that of a direct Rackspace connection. For this experiment, RACS was using a (2,3) configuration backed by three Rackspace repositories. RACS and the client were on the same machine at Cornell. Perhaps surprisingly, RACS does not lag behind. We attribute this to parallelism: Even though RACS is uploading 50% more data for **put** requests after erasure coding, it saves time by running repository queries concurrently.

RACS' major CPU expense is erasure coding. The prototype uses the Zfec [29] Reed-Solomon erasure coding library. On 2 GHz Core 2 Duo machine, we clocked Zfec encoding at a rate of 95 MB/sec, and decoding 151 MB/sec, using only one CPU core. We conclude that erasure coding is not likely to become a bottleneck until gigabit ethernet speeds are involved. Our RACS proxy's CPU usage hovered at around 25% during the benchmarks. Beyond these speeds, Distributed RACS can be used to scale beyond the limitations of one proxy. Alternately, we might choose to use more efficient erasure codes, such as those used by RAID 6, to increase erasure coding throughput—albeit at a loss of configurable (m,n) parameters.

Distributed RACS has been tested with two concurrent RACS proxies sharing a single remote ZooKeeper server. The extra overhead was negligible, except when contention forced requests to the same key to be serialized. The contention overhead could be reduced by using an even finer-grained locking system able to control access to individual repositories while guaranteeing external consistency, although the potential for long waits and serialized access is fundamental. RACS could also hypothetically buffer waiting events on the proxy and return immediately to the client, giving the client an illusion of not needing to wait, at the risk of data loss if the RACS proxy crashes.

6. RELATED WORK

As more (and larger) organizations look to the clouds to offload their storage needs, it is imperative that these organizations be able to change storage providers, for any reason, without prohibitive costs (monetary, time, etc). RACS attempts to reduce the cost of switching storage providers.

The main underlying technique that RACS employs to provide its flexibility is RAID at the cloud storage level, which is increasingly common. HAIL [17] uses RAID-like techniques across storage vendors to ensure high-availability and integrity of data stored in the clouds. Peer-to-peer storage [19–21, 32] have employed RAID-like techniques to ensure the availability and durability of data. The difference between systems like HAIL, peer-to-peer storage systems, and RACS is that RACS focuses on economic failures and how to prevent them without excessive overheads, while still benefiting from greater availability and durability of other RAID-like systems.

For storage providers, there are many services for a system like RACS to choose from. We have divided these services into three broad classes. First, high-end cloud storage services such as Amazon S3, Rackspace Cloud Files, GoGrid Cloud Storage, EMC Atmos Online, Nirvanix SDN, and IronMountain, to name a few, are all geared towards hosting and serving storage requests for other services. For example, Twitter and Smugmug use S3. Second, file storage and sharing services targeted at companies to store and share files internally, such as Google Docs Premium and Box.net. These providers are not meant to support other external services or act as the building block for a web application like Twitter or Smugmug, but are intended to be an online repository for people to share their work files. Third, personal file backup services like Mozy and Microsoft SkyDrive that are aimed at individuals who want to back up their files online. The current design of RACS targets the first of these categories. For example, our trace of the Internet Archive shows that they serve approximately 300GB a month of outgoing FTP GETs (outgoing bandwidth being served to clients). However, one can imagine that in a future work we can extend RACS to support multi-tier storage.

Finally, there are an increasing number of services, such as DuraCloud and Nasuni, that lie between cloud storage providers and clients. Such services use diversity and redundancy to provide flexibility, availability, and durability to their clients.

7. CONCLUSIONS

The cloud services marketplace is in its infancy. And while it may be at the forefront of technology, as a market—an *economic entity*—it is not so unique. The commoditization of cloud services has brought with it the characteristics of an economy, good and bad. In cloud computing, it is fitting that technological devices should be used to address economic problems, and that is what RACS is: A simple application of technology to change the structure of a market. In RACS, consumers have a tool that can be used to adjust the trade-off between overhead expense and vendor mobility. In the design of RACS, we have applied erasure coding to a different type of failure (economic) than it is usually used for in storage systems. We have built a working implementation and run microbenchmarks, and we have simulated larger experiments on real-world traces. We can conclude from these simulations and experiments that RACS enables cloud storage customers to explore trade-offs between overhead and mobility, and to better exploit new developments in the fast-paced cloud storage marketplace.

8. FUTURE WORK

We have considered cloud storage as a stand-alone product in this paper. But the cloud is an ecosystem: It is increasingly common for providers to offer a whole range of complementary services such as computing and content distribution. Naturally, providers arrange these services to work well with one another; for example, the virtual compute nodes of Amazon EC2 can read from and write to Amazon S3 storage with low latency and no bandwidth charges. This trend creates a different kind of vendor lock-in that is much harder to break out of. After all, what good is mobility in the storage arena when you are tied to a provider's virtual machine offerings? This line of reasoning is one direction that future work on RACS could take: Making other kinds of offerings more mobile, and moving multiple services together.

Another direction is the path of heterogeneous repositories: Can RACS make use of a desktop PC as one repository, a cloud providers as second, and a cluster as a third? Thus far, we have made the implicit assumption that cloud providers are infinitely provisioned. Or, at least, that their capacities exceed those of the clients they serve. But if capacity and limits were assigned to repositories, RACS could possibly facilitate policies such as, "Serve from the local repositories up to their capacities, and then use the cloud when demand is high".

9. AVAILABILITY

The RACS source code is published under the BSD license and is freely available http://www.cs.cornell.edu/projects/racs

10. ACKNOWLEDGMENTS

We would like to thank the Internet Archive (Brewster Kahle), Fedora Commons/DuraCloud (Sandy Payette), and Amazon (Werner Vogels) for access to their system, traces and initial discussions about RACS. This work was funded by an AFRL and NSF TRUST grants.

11. REFERENCES

[1] Amazon S3. `http://aws.amazon.com/s3`.
[2] Amazon S3 july 2008 outage. `http://www.networkworld.com/news/2008/072108-amazon-outages.html`.
[3] Amazon S3 SLA. `http://aws.amazon.com/s3-sla`.
[4] Cloud services outage report. `http://bit.ly/cloud_outage`.
[5] DuraCloud Project. `http://www.duraspace.org/duracloud.php`.
[6] EMC Atmos Online Storage. `http://www.atmosonline.com/?page_id=7`.

[7] GoGrid Cloud Storage. http://www.gogrid.com/cloud-hosting.

[8] GoGrid SLA. http://www.gogrid.com/legal/sla.php.

[9] Internet Archive. http://www.archive.org/.

[10] Nirvanix Storage Deliver Network. http://www.nirvanix.com/how-to-buy/self-service-pricing.aspx.

[11] Rackspace Cloud Files. http://www.rackspacecloud.com/cloud_hosting_products/files.

[12] Rackspace june 2009 outage. http://www.bbc.co.uk/blogs/technology/2009/10/the_sidekick_cloud_disaster.html.

[13] Rackspace SLA. http://www.rackspacecloud.com/legal/cloudfilessla.

[14] The ZooKeeper project. http://hadoop.apache.org/zookeeper.

[15] E. Allen and C. M. Morris. Library of Congress and DuraCloud Launch Pilot Program Using Cloud Technologies to Test Perpetual Access to Digital Content. In *Library of Congress, News Release*, July 14 2009. http://www.loc.gov/today/pr/2009/09-140.html.

[16] J. Bloemer, M. Kalfane, M. Karpinski, R. Karp, M. Luby, and D. Zuckerman. An XOR-based erasure-resilient coding scheme. Technical Report TR-95-048, The International Computer Science Institute, Berkeley, CA, 1995.

[17] K. Bowers, A. Juels, and A. Oprea. HAIL: A high-availability and integrity layer for cloud storage. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, Nov. 2003.

[18] Y. Chen, J. Edler, A. Goldberg, A. Gottlieb, S. Sobti, and P. Yianilos. Prototype implementation of archival intermemory. In *Proc. of IEEE ICDE*, pages 485–495, Feb. 1996.

[19] B. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient replica maintenance for distributed storage systems. In *Proc. of USENIX NSDI*, San Jose, CA, May 2006.

[20] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. of ACM SOSP*, Oct. 2001.

[21] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris. Designing a DHT for low latency and high throughput. In *Proc. of USENIX NSDI*, Mar. 2004.

[22] R. Dingledine, M. Freedman, and D. Molnar. The freehaven project: Distributed anonymous storage service. In *Proc. of the Workshop on Design Issues in Anonymity and Unobservability*, July 2000.

[23] D. Ionescu. Microsoft red-faced after massive sidekick data loss. *PCWorld*, Oct. 2009.

[24] M. Luby. Lt codes. In *Proc. of FOCS Symp.*, pages 271–282, Nov. 2002.

[25] M. Luby, M. Mitzenmacher, M. Shokrollahi, D. Spielman, and V. Stemann. Practical loss-resilient codes. In *Proc. of ACM STOC*, pages 150–159, 1997.

[26] M. Luby, M. Mitzenmacher, M. Shokrollahi, D. Spielman, and V. Stemann. Analysis of low density codes and improved designs using irregular graphs. In *Proc. of ACM STOC*, May 1998.

[27] P. Maymounkov. Online codes. Technical Report TR2002-833, New York University, New York, NY, Nov. 2002.

[28] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus open-source cloud-computing system. In *CCGRID '09: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 124–131, Washington, DC, USA, 2009. IEEE Computer Society.

[29] Z. O'Whielacronx. zfec forward error correction library. http://allmydata.org/trac/zfec, 2009.

[30] D. Patterson, G. Gibson, and R. Katz. The case for RAID: Redundant arrays of inexpensive disks. In *Proc. of ACM SIGMOD Conf.*, pages 106–113, May 1988.

[31] J. Plank. A tutorial on reed-solomon coding for fault-tolerance in RAID-like systems. *Software Practice and Experience*, 27(9):995–1012, Sept. 1997.

[32] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the OceanStore prototype. In *Proc. of USENIX FAST*, 2003.

[33] S. Rhea, C. Wells, P. Eaton, D. Geels, B. Zhao, H. Weatherspoon, and J. Kubiatowicz. Maintenance free global storage in oceanstore. In *Proc. of IEEE Internet Computing*. IEEE, Sept. 2001.

[34] L. Rizzo and L. Vicisano. A reliable multicast data distribution protocol based on software fec. In *Proc. of HPCS*, Greece, June 1997.

[35] M. A. Shah, M. Baker, J. C. Mogul, and R. Swaminathan. Auditing to keep online storage services honest. In *HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, pages 1–6, Berkeley, CA, USA, 2007. USENIX Association.

[36] A. Shokrollahi. Raptor codes. Technical Report DF2003-06-01, Digital Fountain, Inc., Fremont, CA, June 2003.

[37] H. Stevens and C. Pettey. Gartner Says Cloud Computing Will Be As Influential As E-business. In *Gartner Newsroom, Online Ed.*, June 26 2008. http://www.gartner.com/it/page.jsp?id=707508.

[38] M. Vrable, S. Savage, and G. M. Voelker. Cumulus: Filesystem backup to the cloud. *Trans. Storage*, 5(4):1–28, 2009.

[39] H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Proc. of IPTPS*, Mar. 2002.