Cloudifying Source Code Repositories: How Much Does it Cost?

Michael Siegenthaler
Dept. of Computer Science
Cornell University
msiegen@cs.cornell.edu

Hakim Weatherspoon
Dept. of Computer Science
Cornell University
hweather@cs.cornell.edu

***Abstract*—Cloud computing provides us with general purpose storage and server hosting platforms at a reasonable price. We explore the possibility of tapping these resources for the purpose of hosting source code repositories for individual projects as well as entire open source communities. An analysis of storage costs is presented, and a complete hosting solution is built and evaluated as a proof-of-concept.**

I. Introduction

The advent of cloud computing has brought us a dazzling array of public computing services that can be instantly tapped by anyone with a credit card number. Users are spared from having to invest in expensive infrastructure such as servers, disks, and cooling equipment because the service provider takes care of these and amortizes the cost across many clients, achieving efficiency through economies of scale. Companies are realizing that it no longer makes sense to build and manage all of their own infrastructure, and services "in the cloud" are quickly becoming popular.

It seems inevitable, then, that software development projects will turn to cloud computing to store their master code repositories, either on a project-by-project basis or as part of a larger migration of a SourceForge-like community. Even small code repositories represent a huge investment of developer-hours, so the need to store this data durably and reliably is obvious. Less obvious are the shortcomings of traditional storage systems: RAID arrays, off-site replicas, and tape backup, properly managed, protect against data loss, but they are neither cheap nor simple, especially when developers and server administrators are geographically spread thin.

In this paper, we focus on the costs of moving source code repositories to the cloud as an example of moving services in general to the cloud, especially collaborative open source projects. Such an endeavor includes many costs, the most critical of which is storage since that is the simplest and likely first component to be moved. We set an agenda for demonstrating the financial storage and computing costs of moving source code repositories to the cloud.

In section II we explain what it means to store a code repository in the cloud and why there are cost advantages to doing so. Section III is a case study on using Amazon's S3 to host some popular open source communities, and includes a cost analysis. In section IV we present an implementation that ties Subversion to S3, with front-end servers running on Amazon's EC2 and using Yahoo's ZooKeeper for consistency. In section V we evaluate the performance of this solution, and in section VI we address related work.

II. Cloudifying Source Repositories

In a revision control system, a master copy of the source code (or other data) is stored in a logically centralized *repository*. Each developer *checks out* and then keeps a *working copy* on his machine that mirrors the repository. The developer edits files in his working copy and periodically *commits* the changes back to the repository, and *updates* his working copy to reflect the changes made by other developers. Each commit is assigned a unique, sequential *version number*. The repository maintains complete history so at any point in time it is possible to check out a working copy for any specified version number.

Storing a repository in the cloud eliminates worries of data loss due to hardware failure, but issues of access control and consistency must still be addressed. Authorized users should be able to commit new versions of files to the repository, but not edit existing history. Users expect the repository to be consistent and for any changes they make not to be pre-empted later on, even in the face of cloud services that offer lesser guarantees. For these reasons we do not expect that clients will be directly using the cloud storage API anytime soon, but that they will contact one of a set of front-end servers that are responsible for enforcing access control, ensuring consistency, and pushing the data into the cloud. These might consist of virtualized server instances in the cloud, or traditional physical machines owned by the community, but in either case their local storage systems are allowed to be cheap and unresilient against hardware failure.

Another consideration with any hosting solution is resource provisioning. Open source communities with limited budgets and private enterprises that are increasingly cost-sensitive may well prefer to pay just for the resources they use, rather than trying to budget in advance what they are going to need. Cloud computing makes this a possibility, and increased competition among providers of commodity services will ensure that prices are reasonable.

III. Case Study: S3/EC2

By far the most popular general purpose cloud storage service today is Amazon's S3. We chose to use this as a basis

for cost studies and for the implementation of our system. S3 is an appealing choice because Amazon also offers the EC2 virtualized computing service, so it is possible to use their services as a complete hosting solution with low latency access to storage.

A. How much does it cost?

The cost analysis is based on real-world traces taken from the Subversion repositories of popular open source projects. Subversion represents each revision in a repository's history, regardless of how many changes it contains, as two files, the first for data, as a diff against previous revisions, and the second for meta-data such as the author, timestamp, and other revision properties. Our cost analysis is based on the sizes of these files and the time at which each revision was committed.

Looking up the size of these special files is only possible if one has filesystem level access to the disk on which the repository is stored, so we had to use Subversion's mirroring capability to fetch revisions from the network-accessible repository and replay them against a local copy. Doing this also implicitly gives us the log of timestamps indicating when each revision was committed. Thus it is possible to calculate the bandwidth, storage, and per-transaction costs of pushing the two files for each revision into S3 over time, based on Amazon's current pricing structure, shown in table I.

TABLE I
AMAZON'S S3 PRICING STRUCTURE

Description	Price
Monthly storage	$0.15 per GiB
Bandwidth in	$0.10 per GiB
Bandwidth out	$0.17 per GiB
Per 1000 writes	$0.01
Per 10,000 reads	$0.01

Not included in the analysis is the cost of fetching data out of S3 to be served to clients. This cost will vary depending on how much caching is done on the front-end servers, but with EC2 providing well over 100 GiB of per-instance storage, and dedicated servers potentially having much more due to inexpensive SATA disks, it is not unreasonable to assume that a cache hit rate of close to 100% is possible. As an example, the 506 public Subversion repositories of the Debian Linux community amount to a total of only 33 GiB. The only outgoing bandwidth costs are then to to replace failed front-end servers or to synchronize replicas if more than one is in use. In the case of EC2, the bandwidth costs are actually waived and the user then pays only for the traffic between the front-end servers and their clients.

Table II shows the cost of using S3 for a number of individual open source projects, as well as an aggregate for the 506 repositories of the Debian community. Also shown is an estimate for the Apache Software Foundation. Apache has taken the unusual approach of using a single repository for all of its projects, both public and restricted; due to access control restrictions on some paths, Subversion's mirroring tool was unable to create local copy. The complete log of timestamps, however, was available, so we based our analysis on that along with the assumption each revision data file would be 37031 KiB and each revision property file 185 B, the averages observed for the other repositories in table II.

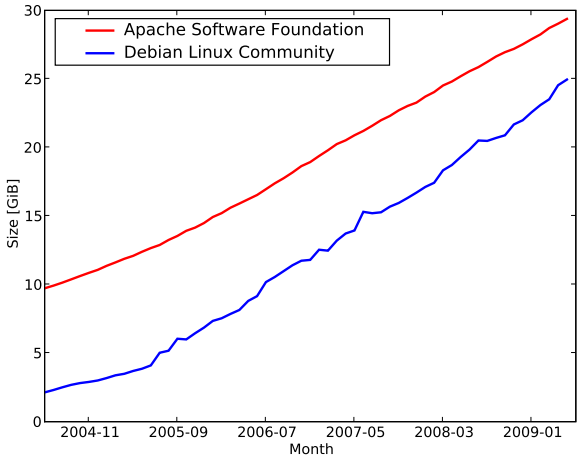

Fig. 1. Size of repository stored in S3

TABLE II
MOST RECENT MONTHLY COST OF STORING REPOSITORIES IN S3 FOR INDIVIDUAL PROJECTS AND ENTIRE COMMUNITIES

Software Project	Monthly Cost
SquirrelMail	$0.03
phpMyAdmin	$0.04
Subversion	$0.08
Mono	$0.57
KDE	$7.35
Hosting Community	**Monthly Cost**
Debian Linux Community	$3.89
Apache Software Foundation	$4.58

Even for the fairly large Apache Software Foundation, the current cost of using S3 for storage is less than $5 per month. It is very unlikely that any vendor could provide a traditional storage solution consisting of SCSI disks and tape backup at this price. The amount of S3 storage required of course increases each month as the repository grows, but as shown in figure 1, the increase is roughly linear, as developer productivity remains constant. The cost of storage is declining exponentially [1], so if Amazon's pricing stays competitive, the long-term trend is towards lower costs.

Additional costs will be incurred for front-end servers. For the case of EC2, a standard machine instance is billed at $0.10 per hour, plus data transfer of $0.10 per GiB in and $0.17 per GiB out. Discounts are available if data transfer exceeds 10 TiB per month, and the instance cost may be reduced to $0.03 per hour by paying a $500 three-year reservation fee in advance. This gives an amortized monthly cost of $35.80 plus bandwidth. As we show in the next section, one instance should be enough for almost any individual project or moderately sized community.

B. Usage Patterns

In addition to getting a grasp of the costs involved in moving a repository to S3, it is important to understand the usage patterns, especially the rate at which commits take place. Since achieving the consistency properties that developers expect will require a consistency layer to be built in front of S3, it is crucial that any such layer be able to handle the load of commits.

The critical statistic to consider is the number of simultaneous commits. For centralized revision control system such as Subversion, each commit is assigned a unique, sequential number, and any change to a versioned file is stored as a diff against its previous version. A commit must be rejected if any of the versioned files that it touches have been changed in an earlier revision that the developer performing the commit was unaware of. This ensures that every conflict gets resolved by a human before becoming part of the repository's state. Thus, exclusive locking is required on commits.

Taking a loose definition of simultaneous to be "within one minute", the Apache repository had a maximum of 7 simultaneous commits and the Debian community, ignoring for now that their use of 506 separate repositories allows for finer-grained locking, an aggregate maximum of 6. In determining these numbers we filtered out any sequences of multiple commits by the same author during a one minute period since those were likely sequential rather than simultaneous and do nor represent the common case. The average rates were 1.10 and 1.12, respectively, so exclusive locking for commits should not pose any scalability problems in a typical environment.

We did not consider the rate of read operations because clients updating their working copies, or reading from the repository, do not require a lock. The Debian community today uses only a single Subversion server, and the Apache foundation has a master server plus a European mirror, primarily for latency reasons. As such, we expect that most communities will have at most a handful of front-end servers.

C. Achieving Consistency

Amazon's infrastructure is built on the principle of *eventual consistency* [2], and does not directly support the locking required for revision control. Originally developed to run the company's own online store, the system preferred availability over consistency because downtime translated directly into lost revenue. Customers may opt to shop elsewhere or to simply forgo impulse purchases that they didn't really need anyway. An inconsistent shopping cart, however, could be resolved by heuristics or user-intervention at checkout time.

It is well known that consistency and availability cannot both be achieved simultaneously in any real network where hosts or entire subnetworks are sometimes unreachable due to connectivity losses [3]. If a cloud service is designed to provide high availability but an application instead requires perfect consistency, additional software infrastructure is required to bridge the gap.

For revision control it makes sense to adopt eventual consistency for read operations, since at worst an earlier revision will be returned. If the user is aware, through some other channel, that a newer version should exist, s/he can retry and expect that version to be available within a short timeframe. For commits, as detailed earlier, perfect consistency is required and a locking layer must be built to support this. This may result in a commit being rejected if consensus cannot be reached, but shouldn't be a problem because code changes are usually not impulse decisions and the commit can be retried later.

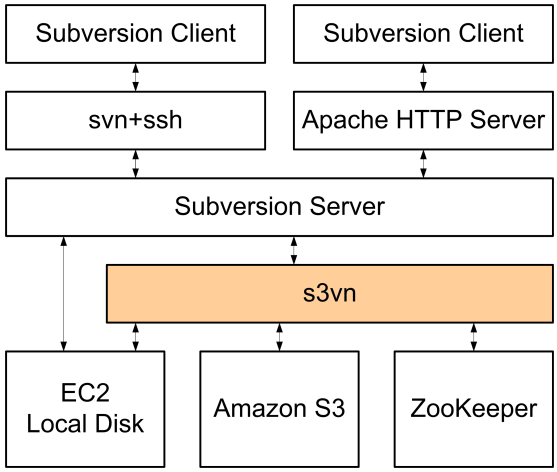

Fig. 2. System architecture

IV. Design

As a proof-of-concept, we built s3vn, a tool for integrating Subversion with S3. s3vn is colocated with Subversion and inserts a layer between Subversion and S3, as shown in figure 2. For simplicity we did not modify the Subversion server in any way; s3vn is responsible for receiving event notifications from Subversion and transferring data between the local disk on the EC2 instance and S3. Subversion calls s3vn at the start and end of each commit, and s3vn acquires and releases locks using Yahoo's open source ZooKeeper lock service.

The difficulty achieving consistency with a service such as Amazon's S3 stems from the fact that files pushed into the storage cloud do not simultaneously become available on all service endpoints. If a file is overwritten, different clients may read back different versions, and even the same client may see the old version if it suddenly switches to speaking with a different S3 endpoint. The file will always be internally consistent, since put and get operations are atomic, but its contents may not reflect expectations that the client formed based on other files and out of band communication.

s3vn works around the consistency problem by storing the number of the latest revision into ZooKeeper. A revision, even if multiple files were changed by the client, is represented by Subversion has a single file containing binary diffs against earlier revisions. A revision is never changed after the fact, so a front-end server attempting to fetch a revision i from S3 will receive either the one true, consistent revision i, or a missing file error if i was posted so recently that it has not

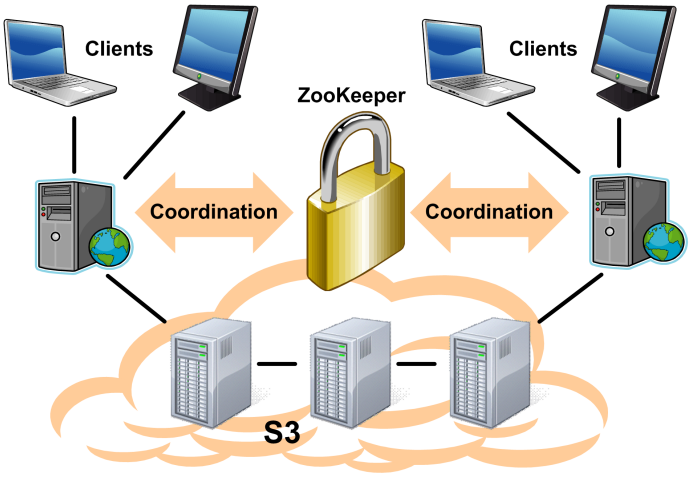

Fig. 3. System overview, showing that front-end servers are equivalent and clients may interact with any of them

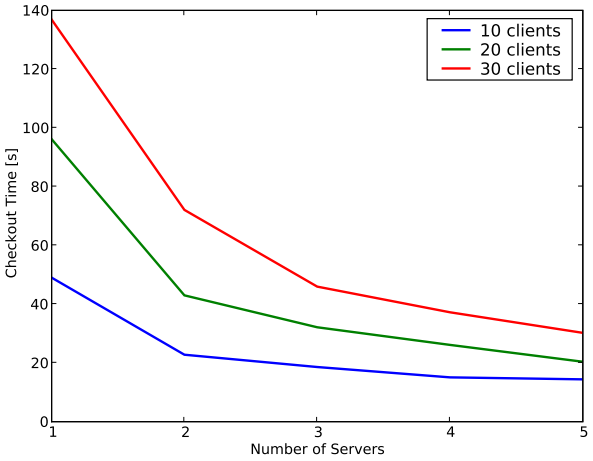

Fig. 4. Performance of simultaneous checkouts

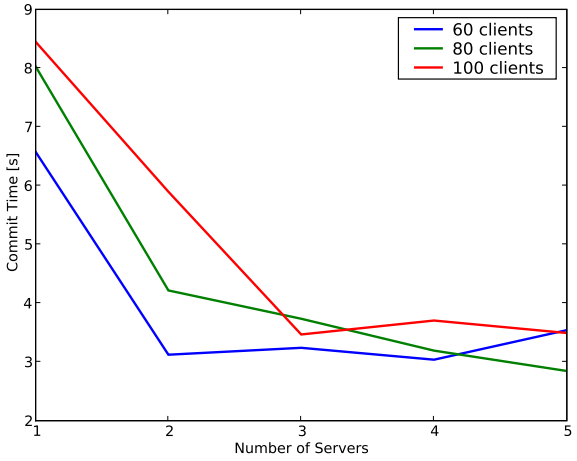

Fig. 5. Performance of simultaneous commits

yet propagated through S3. In the latter case, the server retries indefinitely until the file is available.

ZooKeeper ensures that the latest revision number is incremented atomically. ZooKeeper maintains a simple file-system like tree of nodes; nodes may store a small amount of data and can have children. s3vn stores the latest revision number in /s3vn/<repo>/current, supporting multiple named repositories in a single ZooKeeper tree. Before pushing a new revision, a front-end server must acquire a lock by creating a *sequence* node /s3vn/<repo>/lock/lock-, to which ZooKeeper will append a unique, monotonically increasing sequence number. The front-end server then lists the children of /s3vn/<repo>/lock; if its own lock node has the lowest number, it may proceed with the commit. Otherwise it *watches* the node with the next lower number in order to be notified when that node and its associated lock go away. After comitting the revision to S3 and updating /s3vn/<repo>/current, it releases its lock by deleting the lock node. Lock nodes are marked with ZooKeeper's *ephemeral* flag to ensure that the lock is forcibly released if the front-end server fails.

ZooKeeper runs as a replicated service, so it remains available as long as a majority of the hosts are up and reachable. A client only speaks to one ZooKeeper server at a time (though it may fail-over to another server if necessary), but the server ensures that the relevant state has been replicated before responding to a client's request.

In general multiple front-end servers may be run, each on its own EC2 instance. The system is organized as in figure 3. Unlike the traditional replicated Subversion setups that are used today, no single server acts as a master; with s3vn all are equivalent.

V. Evaluation

We observe that running multiple front-end servers, which cloud computing makes easy to do, increases the throughput of read operations. We tested s3vn by running a fixed number of clients, each repeatedly checking out about 14 MiB of source code from an EC2/S3-hosted repository, and varying the number of servers over which the load was distributed. Figure 4 shows the results.

Write performance was measured by observing the latency of simultaneous commits from different clients. Since simultaneous commits to a single repository would not be a typical case, 1000 individual s3vn repositories were used, all sharing the same set of front-end servers and same set of three ZooKeeper servers. Each client checked out a random repository from a random front-end server, and then repeatedly committed small amounts of data. Changes were propgated in the background to the other front-end servers, via S3. Figure 5 shows that adding front-end servers can indeed alleviate latency problems caused by high load, and that the overhead of propagating data in the backgound is not significant enough to negatively affect performance.

VI. Related Works

Moving services to the cloud has been published on in other contexts. Cumulus [4] is a backup application that implements a custom block-based file system to store multiple versions of backup data on S3. The authors make the distinction between *thin-clouds* that provide a low-level API and *thick-clouds* that are designed for a specific application. Thick clouds for a variety of purposes, including backup and source code repository hosting, already exist, with SourceForge and Google Code being examples of the latter. The authors of Cumulus and we show that thin-cloud solutions can be a cost-effective alternative.

Another example of moving a service to the cloud is MetaCDN [5], a content distribution network. The work evaluates the latency of various cloud storage services from several locations and provides an abstraction to integrate the different offerings into a single system.

ElasTraS [6] provides a database-like transactional data store backed by S3, and faced similar issues as s3vn due to its need for high consistency. ElasTraS assigns update priviledges for different areas of the data store to individual front-end servers, using the lock service to elect an owner for each partition, much in the style described by Google's Chubby [7]. Chubby, a lock service based on Paxos [8], defers fine-grained locking to the application in order not to burden the global lock service with high traffic. For s3vn we opted to use the lock service, ZooKeeper, for fine-grained locking instead of just leader election, since the latter would have required duplicating much of ZooKeeper's functionality to replicate the leader's state. Scalability is not an obstacle because there is no need for global locking across multiple repositories; the load can be partitioned across as many ZooKeeper instances as necessary.

Replication is not without its dangers [9], and it has been shown that replicating too eagerly leads quickly to degraded performance. The solution proposed is to use master copy replication, where a transaction does not immediately update all replicas. s3vn treats S3 as the master copy, and only the lock service, which deals with simple, low-bandwidth operations that may be concentrated on a small number of servers, must be eagerly replicated.

Also relevant is SUNDR, the Secure Untrusted Data Repository [10]. This file system allows clients to detect against malicious or compromised storage servers or hosting platforms by providing fork consistency, a property which ensures that clients can detect integrity failures as long as they see each other's file modifications. Similar techniques could be used to recover data from client working copies in the event of a catastrophic cloud failure.

Once code repositories are stored in the cloud, one might imagine enabling mashups in ways not previously possible. For example, web based code viewers, search tools, and cross reference viewers might be built by third-parties, pulling data from the repositories of several distinct communities. CloudViews [11] seeks to enable such applications by granting direct access of cloud storage to third parties, subject to the data owner's security requirements.

A question that may naturally arise is, why not use a general purpose file system interface to S3, such as s3fs, and store a repository on that? This is indeed possible to do, but would entail pushing temporary files such as transactions-in-process into S3 and incurring additional monetary costs due to the increased number of S3 requests. There would also likely be performance problems, since file append and rename operations do not map efficiently to S3's simple get/put API. A specialized s3fs that is aware of Subversion's file naming and use scenario could of course overcome these limitations by pushing only what is actually required into S3, but we believe that such specialized tools are better built on top of a file system abstraction than pushed underneath it.

VII. Conclusion

We have shown that the cost of using a cloud computing storage service for source code repository hosting is low, both for individual projects and moderately sized communities. Considering the costs of a resilient local storage system of SCSI disks and tape backup, cloud computing is a very attractive solution for this application. Our implementation of s3vn brings this concept a step closer to becoming reality, and provides evidence that performance will be acceptable for typical use scenarios.

References

[1] E. Grochowski and R. D. Halem, "Technological impact of magnetic hard disk drives on storage systems," *IBM Syst. J.*, vol. 42, no. 2, pp. 338–346, 2003.

[2] W. Vogels, "Eventually consistent," *ACM Queue*, 2009. [Online]. Available: http://queue.acm.org/detail.cfm?id=1466448

[3] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent available partition-tolerant web services," in *In ACM SIGACT News*, 2002, p. 2002.

[4] M. Vrable, S. Savage, and G. M. Voelker, "Cumulus: Filesystem backup to the cloud." in *FAST*, M. I. Seltzer and R. Wheeler, Eds. USENIX, 2009, pp. 225–238.

[5] J. Broberg and Z. Tari, "Metacdn: Harnessing storage clouds for high performance content delivery," in *ICSOC '08: Proceedings of the 6th International Conference on Service-Oriented Computing*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 730–731.

[6] S. Das, D. Agrawal, and A. E. Abbadi, "Elastras: An elastic transactional data store in the cloud," in *HotCloud*. USENIX, 2009.

[7] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *USENIX'06: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2006, p. 24. [Online]. Available: http://portal.acm.org/citation.cfm?id=1267308.1267332

[8] L. Lamport, "Paxos made simple," *ACM SIGACT News*, vol. 32, no. 4, pp. 18–25, December 2001.

[9] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," in *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, vol. 25, no. 2. New York, NY, USA: ACM Press, June 1996, pp. 173–182. [Online]. Available: http://dx.doi.org/10.1145/233269.233330

[10] J. Li, M. Krohn, D. Mazières, and D. Shasha, "Secure untrusted data repository (sundr)," in *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*. Berkeley, CA, USA: USENIX Association, 2004, pp. 9–9.

[11] R. Geambasu, S. D. Gribble, and H. M. Levy, "Cloudviews: Communal data sharing in public clouds," in *HotCloud*. USENIX, 2009.