Exploring the Scalability of Distributed Version Control Systems

Michael J. Murphy Otto J. Anshus John Markus Bjørndalen

November 2017

Abstract

Distributed version control is an interesting form of distributed system because it takes eventual consistency to the extreme. Every replica of a repository contains the full history in an append-only data structure, any replica may add new commits, and conflicting updates are reconciled later in a merge operation. These systems are popular, but their use is generally limited to the small text files of source code.

This paper explores the challenges of using version control to store larger binary files, with the goal of building a scalable, highly-available, distributed storage system for media files such as images, audio, and video. We developed an early prototype of such a system, which we call Distributed Media Versioning (DMV). We perform experiments with the popular version control systems Git and Mercurial, the Git-based backup tool Bup, and our DMV prototype.

We measured commit times and repository sizes when storing single files of increasing size, and when storing increasing numbers of single-kilobyte files. We find that processing files whole will limit maximum file size to what can fit in RAM. And we find that storing millions of objects loose as files with hash-based names will result in inefficient write speeds and use of disk space. We conclude that the key to storing large files is to break them into smaller chunks, and that the key to storing many small chunks is to aggregate them into larger files. We intend to incorporate these insights into future versions of DMV.

1 Introduction

Write: Distributed version control is an interesting form of distributed system because it takes eventual consistency to the extreme.

Write: Every replica of a repository contains the full history in an append-only data structure, any replica may add new commits, and conflicting updates are reconciled later in a merge operation.

Write: These systems are popular, but their use is generally limited to the small text files

This paper was presented at the NIK-2017 conference; see http://www.nik.no/.

of source code.

Distributed version control systems (DVCSs) are designed primarily to store program source code: plain text files in the range of tens of kilobytes. Checking in larger binary files such as images, sound, or video affects performance. Actions that require copying data in and out of the system slow from hundredths of a second to full seconds or minutes. And since a DVCS keeps every version of every file in every *repository*, forever, the disk space needs compound.

This has lead to a conventional wisdom that binary files should never be stored in version control, inspiring blog posts with titles such as ”Don’t ever commit binary files to Git! Or what to do if you do” [4], even as the modern software development practice of continuous delivery was commanding teams to ”keep absolutely everything in version control.” [2, p.33]

Write: This paper explores the challenges of using version control to store larger binary files, with the goal of building a scalable, highly-available, distributed storage system for media files such as images, audio, and video.

2 Distributed Media Versioning

Write: We developed an early prototype of such a system, which we call Distributed Media Versioning (DMV).

3 Experiments

Write: We perform experiments with the popular version control systems Git and Mercurial, the Git-based backup tool Bup, and our DMV prototype.

Write: We measured commit times and repository sizes when storing single files of increasing size, and when storing increasing numbers of single-kilobyte files.

Methodology

We conducted two major experiments. In order to measure the effect of file size, we would *commit* a single file of increasing size to each target *version control system (VCS)*. And to measure the effect of numbers of files, we would commit increasing number of small (1 KiB) files to each target VCS.

Write: List VCSs in a more compact way than in thesis

For each experiment, the procedure for a single trial was as follows:

1. Create an empty repository of the target VCS in a temporary directory
2. Generate target data to store, either a single file of the target size, or the target number of 1 KiB files
3. Commit the target data to the repository, measuring wall-clock time to commit
4. Verify that the first commit exists in the repository, and if there was any kind of error, run the repository’s integrity check operation

5. Measure the total repository size
6. Overwrite a fraction of each target file
7. (Number-of-files experiment only) Run the VCS's status command that lists what files have changed, and measure the wall-clock time that it takes to complete
8. Commit again, measuring wall-clock time to commit
9. Verify that the second commit exists in the repository, and if there was any kind of error, run the repository's integrity check operation
10. Measure the total repository size again
11. (File-size experiment only) Run Git's garbage collector (`git fsck`) to pack objects, then measure total repository size again
12. Delete temporary directory and all trial files

We increased file sizes exponentially by powers of two from 1 B up to 128 GiB, adding an additional step at 1.5 times the base size at each order of magnitude. For example, starting at 1 MiB, we would run trails with 1 MiB, 1.5 MiB, 2 MiB, 3 MiB, 4 MiB, 6 MiB, 8 MiB, 12 MiB, and so on.

We increased numbers of files exponentially by powers of ten from one file to ten million files, adding additional steps at 2.5, 5, and 7.5 times the base number at each order of magnitude. For example, starting at 100 files we would run trials with 100, 250, 500, 750, 1000, 2500, 5000, 7500, 10 000, and so on.

Input data files consisted of pseudorandom bytes taken from the operating system's pseudorandom number generator (`/dev/urandom` on Linux).

When updating data files for the second commit, we would overwrite a single contiguous section of each file with new pseudorandom bytes. We would start one-quarter of the way into the file, and overwrite 1/1024th of the file's size (or 1 byte if the file was smaller than 1024 KiB). So a 1 MiB file would have 1 KiB overwritten, a 1 GiB file would have 1 MiB overwritten, and so on.

Experiment Platform

We ran the trials on four dedicated computers with no other load. Each was a typical office desktop with a 3.16 GHz 64-bit dual-core processor and 8 GiB of RAM, running Debian version 8.6 ("Jessie"). Each computer had one normal SATA hard disk (spinning platter, not solid-state), and trials were conducted on a dedicated 197 GiB LVM partition formatted with the ext4 filesystem. All came from the same manufacturer with the same specifications and were, for practical purposes, identical.

Include platform table?

We ran every trial four times, once on each of the experiment computers, and took the mean and standard deviation of each time and disk space measurement. However, because the experiment computers are practically identical, there was little real variation.

4 Results

File Size

Write: We find that processing files whole will limit maximum file size to what can fit in RAM.

In our experiments, both Git and Mercurial had file size limits that were related to RAM. Mercurial would refuse to commit a file 2 GiB or larger. It would exit with an error code and print an error message saying "up to 6442 MB of RAM may be required

to manage this file." The commit would not be stored, and the repository would be left unchanged. This suggests that Mercurial needs to be able to fit the file into memory three times over in order to commit it.

Git's commit operation would appear to fail with files 12 GiB and larger. It would exit with an error code and print an error message saying "fatal: Out of memory, malloc failed (tried to allocate 12884901889 bytes)." However, the commit would be written to the repository, and git's `fsck` operation would report no errors. So the commit operation completes successfully, even though an error is reported.

With files 24 GiB and larger, Git's `fsck` operation itself would fail. The `fsck` command would exit with an error code and give a similar "fatal ... malloc" error. However, the file could still be checked out from the repository without error. So we continued the trials assuming that these were also false alarms.

These findings are summarized in Table 1 and Table 2.

[Table 1 about here.]

[Table 2 about here.]

Write: Give file size result timing

Number of Files

Write: And we find that storing millions of objects loose as files with hash-based names will result in inefficient write speeds and use of disk space.

5 Discussion

Write: Discuss implementation details that lead to RAM limits

6 Conclusion

Write: We conclude that the key to storing large files is to break them into smaller chunks, and that the key to storing many small chunks is to aggregate them into larger files.

Write: We intend to incorporate these insights into future versions of DMV.

References

[1] CHACON, S. and STRAUB, B. *Pro Git*. 2nd. Berkely, CA, USA: Apress, 2014. ISBN: 1484200772, 9781484200773. URL: `https://git-scm.com/book/en/v2` (visited on Apr. 27, 2017).

[2] HUMBLE, J. and FARLEY, D. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader)*. Pearson Education, 2010.

[3] TORVALDS, L. *Git - the stupid content tracker*. Git source code README file. From the initial commit of Git's source code into Git itself (revision e83c516). Apr. 8, 2005. URL: `https://github.com/git/git/blob/e83c5163316f89bfbde7d9ab23ca2e25604af290/README` (visited on Apr. 25, 2017).

[4] WINSLOW, R. *Don't ever commit binary files to Git! Or what to do if you do.* The Blog of Robin. June 11, 2013. URL: https://robinwinslow.uk/2013/06/11/dont-ever-commit-binary-files-to-git/ (visited on May 11, 2017).

Todo list

Scratch Pad

This section contains text that I have written, but decided to set aside for now.

From the overlong abstract

Intro about multiple devices and not the cloud

A typical computer user has multiple devices holding an increasing amount of data. Most users will have at least a computer and a mobile phone. Many will also have a work computer, tablet, or other devices. These devices have varying resources, including processing, memory, and storage. They may also be in different locations, on different networks, or turned off at any time. The user's data will be in files of varying sizes and media types, from kilobyte text documents to multi-gigabyte videos and beyond. The volume of data is also always increasing as data is authored, collected from the internet, or gathered from mobile sensors. This data is strewn across these devices in an ad-hoc fashion, according to where it is produced and consumed. When the user needs a particular file, they must either remember where it is or perform a frustrating, manual, multi-device search. Also, copies of data on different devices will diverge if updates are made separately and not reconciled.

Cloud computing eases these problems by centralizing storage, searching, and update reconciliation. However, the user's access to their data depends on the reliability of their network connection and the reliability and longevity of the cloud service. Handing data

over to a third party also raises concerns about privacy. The cloud service may also charge a recurring subscription fee.

Detailed Description of Git's DAG

Git stores its data in a directed acyclic graph (DAG) structure. Blob objects contain file data; tree objects store lists of blobs, representing directories; and a commit objects each associate a particular tree state with metadata such as time, author, and previous commit state, placing that tree state into a history. Each object is stored in an content-addressed object database, indexed by a cryptographic hash of its contents [3].

Objects, once stored, are immutable. Updating an object would change its hash and thus its ID, creating a new object. Because objects refer to other objects by hash ID, a new object can only refer to a pre-existing object with known content. The graph is directed because these links flow in one direction, and it is acyclic because links cannot be created to objects that do not exist yet, and existing objects cannot be updated to point to newer objects. The DAG is append-only.

Such a DAG structure has several interesting properties for data storage.

De-duplication Identical objects are de-duplicated because they will have the same ID and naturally collapse into a single object in the data store. This results in a natural compression of redundant objects.

A record of causality Copies of the DAG can be distributed and updated independently. Concurrent updates will result in multiple branches of history, but references from child commit to parent commit establish a happens-before relationship and give a chain of causality. Branches can be merged by manually reconciling the changes, and then creating a merge commit that refers to both parent commits. When transferring updates from one copy to another, only new objects need to be transferred.

Atomic updates When a new commit is added, all objects are added the database first, then finally the reference to the current commit is updated. This reference is a 160-byte SHA-1 hash value, and which can be updated atomically.

Verifiability Because every object is identified by its cryptographic hash, the data integrity of each object can be verified at any time by re-computing and checking its hash. And because objects refer to other objects by hash, the graph is a form of blockchain. All objects can be verified by checking hashes and following references from the most recent commits down to the initial commits and the blob leaves of the graph.

Compression Depends on Mapping files to DAG objects

The efficiency of de-duplication depends on how well identical pieces of data map to identical objects. In Git, the redundant objects are the files and directories that do not change between commits. De-duplication of redundant data within files is accomplished by aggregating objects together into pack files and compressing them with zlib [1, Section 10.4].

Some Images

[Figure 1 about here.]

[Figure 2 about here.]

[Figure 3 about here.]

Alternate outline

```
- Problem: many devices, more data, difficult to follow what is where

- Cloud not solution. Relies on third party. Connection, privacy, etc.

- DVCS: An alternate approach

    - Extreme availability
    - Version history gives chain of causality for later reconciliation

- Interesting properties of DAG

    - DAG gives de-duplication
    - Content addressing gives tampering/bitrot protection
    - DAG also gives convenient ways to shard data

- Problem 1: dealing with larger files: chunking

    - Bup has chunking but locked into backup workflow

- Problem 2: dealing with many files: packing

    - Git has packing but in separate step that fails for large files

- Problem 3: increasing data: sharding

- In-between: de-duplication

- DMV prototype

- Experiments

    - File size and number of files
    - Random writes

- Results

- Conclusion

    - chunk, content-address, re-pack
```

- DMV not yet viable, but it’s a start

List of Figures

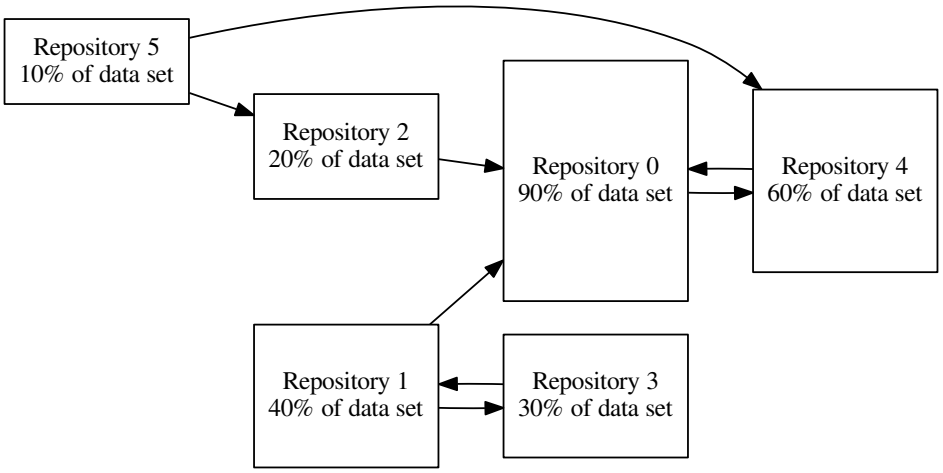


Figure 1: Repositories in an ad-hoc network

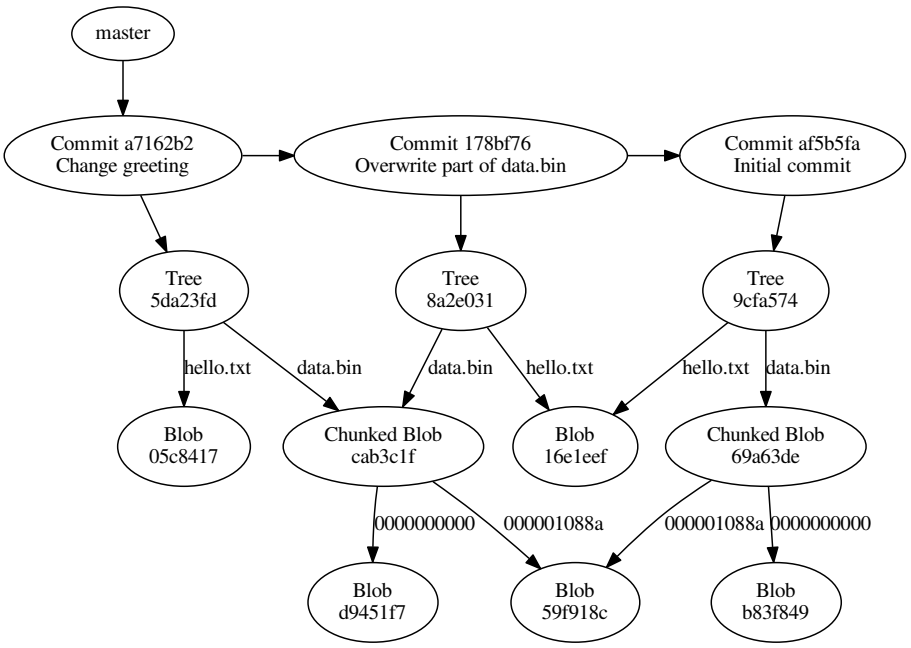
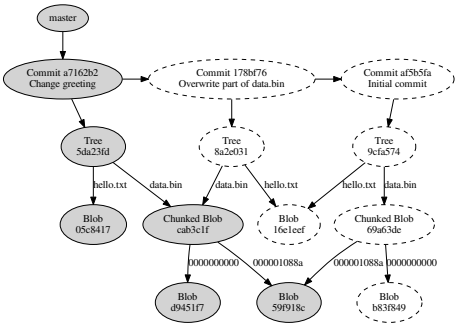
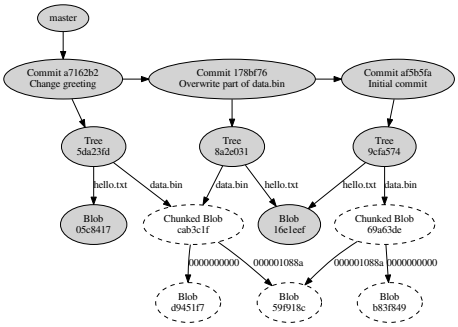


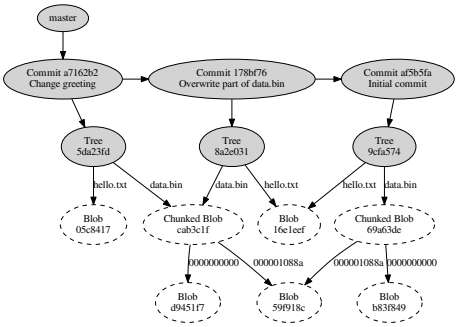
Figure 2: A simple DMV DAG with three commits



(a) Partial history of full data set



(b) Full history of part of data set



(c) Full history of metadata

Figure 3: A DMV DAG, sliced in different dimensions

List of Tables

Table 1: Observations as file size increases

Size	Observation
1.5 GiB	Largest successful commit with Mercurial
2 GiB	Mercurial commit rejected
8 GiB	Largest successful commit with Git
12 GiB	Git false-alarm errors begin, but commit still intact
16 GiB	Largest successful Git fsck command
24 GiB	Git false-alarm errors begin during fsck, but commit still intact
64 GiB	Largest successful DMV commit
96 GiB	DMV timeout after 5.5 h
96 GiB	Last successful commit with Bup (and Git, ignoring false-alarm errors)
128 GiB	All fail due to size of test partition

Table 2: Effective size limits for VCSs evaluated

VCS	Effective limit
Git	Commit intact at all sizes, UI reports errors at 12 GiB and larger
Mercurial	Commit rejected at 2 GiB and larger
Bup	Successful commits at all sizes tried, up to 96 GiB
DMV	Successful commits up to 64 GiB, timeout at 5.5 h during 96 GiB trial