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” [3], even as the modern software development practice of continuous delivery was commanding teams to ”keep absolutely everything in version control.” [1, 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

File Size Limits: RAM, Time, Disk Space

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.

The *Distributed Media Versioning (DMV)* prototype was able to store a file up to 64 GiB in size, but time became a limiting factor as file size increased. At 96 GiB, our experiment script timed out and terminated the commit after five and a half hours.

Our experiment environment itself limited the largest file stored by any VCS to 96 GiB. Any larger and it was simply impossible to store a second copy of the file on our 197 GiB test partition. Bup was able to store a 96 GiB file with no errors in just under two hours. Git could also store such a large file, but one must ignore the false-alarm ”fatal” errors being reported by the user interface.

These findings are summarized in Table 1 and Table 2.

[Table 1 about here.]

[Table 2 about here.]

Commit Times for Increasing File Sizes

Write: Give file size result timing

Figure 1 shows the wall-clock time required for the initial commit, adding a single file of the given size to a fresh repository. Over all, the trend is clear and unsurprising: commit time increases with file size. It increases linearly for Git, Mercurial, and Bup. DMV's commit times increase in a more parabolic fashion, which is most apparent in Figure 1e.

[Figure 1 about here.]

Number of Files

File Quantity Limits: inodes

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.

Git, Mercurial, DMV, and the copy operation all failed when trying to store 7.5 million files or more, reporting that the disk was full. However, the disk was not actually out of space — it was out of inodes.

Unix filesystems, ext4 included, store file and directory metadata in a data structure called an *inode*, which reside in a fixed-length table [2]. When all of the inodes in the table are allocated, the filesystem cannot store any more files or directories.

Bup avoided the inode limit. Bup trials could continue until the input data itself exhausted the system's inodes attempting to generate 25 million input files.

Commit Times for Increasing Numbers of Files
Figure 2

[Figure 2 about here.]

Figure 2 shows the time required for the initial commit, storing all files into a fresh empty repository. Here we see the commit times for Git and DMV increasing quadratically with the number of files, while Mercurial, Bup, and the copy increase linearly.

5 Discussion

Write: Discuss implementation details that lead to size limits

Write: Mercurial: Diff during commit

Write: Git: Compress during gc

DMV's parabolic increase is due to the way it breaks the large file into chunks and stores objects as individual files on the filesystem. While it is reading one large file, it is writing many small files, which incurs filesystem overhead. So its performance characteristic for storing a large file is closer to that of storing many files (**??**). Bup also breaks the file into many chunks, but it avoids the filesystem overhead by recombining the chunks into *pack files*. We investigate the filesystem overhead further in **??**.

This sluggishness is due to the way DMV stores chunks of the file as individual files on the filesystem, turning the problem of storing one large file into the problem of storing many small files. Storing many small files in this way incurs filesystem overhead, as we discovered in the results of the number-of-files experiment (**??**), and later performed more experiments to examine in detail (**??**).

File Quantity Limits

Git, Mercurial, DMV, and the copy all create one file in their *object stores* for each input file. So to store 7.5 million files, they will create 7.5 million more, resulting in 15 million files on the filesystem, plus directories. However, the 197 GiB experiment partition has 13 107 200 total inodes, so storing 15 million files is impossible.

Bup is able to store more files because it does not write a separate object file for each input file. Bup aggregates its DAG objects into pack files, writing several large files instead many small files. As such, it does not exhaust the disk's inodes, and can continue until the experiment itself exhausts the system's inodes when it tries to go up from 10 million files to the next step and run a trial with 25 million files.

Hash-Based Directory Names Cause Disk Seeking

Write: Discuss Git and DMV slowing down due to random writes

We saw in the file-size commit times (**??**) that DMV's time increased quadratically, and we suspected that was because it was creating many small files and incurring filesystem overhead. This effect would explain why both Git and DMV do so poorly here while Bup would fare much better. But why then would Mercurial and the copy also have a linear increase instead of an quadratic one?

The difference is the naming schemes of stored files. Git and DMV name each object file according to the SHA-1 hash of the object's contents, while Mercurial, like the copy, uses the original input file's name. This means that Git and DMV write files in a random order with respect to their names, jumping between different object store subdirectories, while Mercurial and the copy can write files in the order they read them, one subdirectory at a time. The filesystem is most likely optimized for that kind of sequential write.

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

[2] RITCHIE, O. M. and THOMPSON, K. "The UNIX time-sharing system". In: *The Bell System Technical Journal* 57.6 (1978), pp. 1905–1929. ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1978.tb02136.x.

[3] 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. It should disappear if the `final` option is applied to the document.

List of Figures

Figure 1: Wall-clock time to commit one large file to a fresh repository

Simplify to just largest sizes, convert to black & white

Subfigure (a) shows the full range on a logarithmic scale, while the others are linear-scale for specific ranges and include error bars.

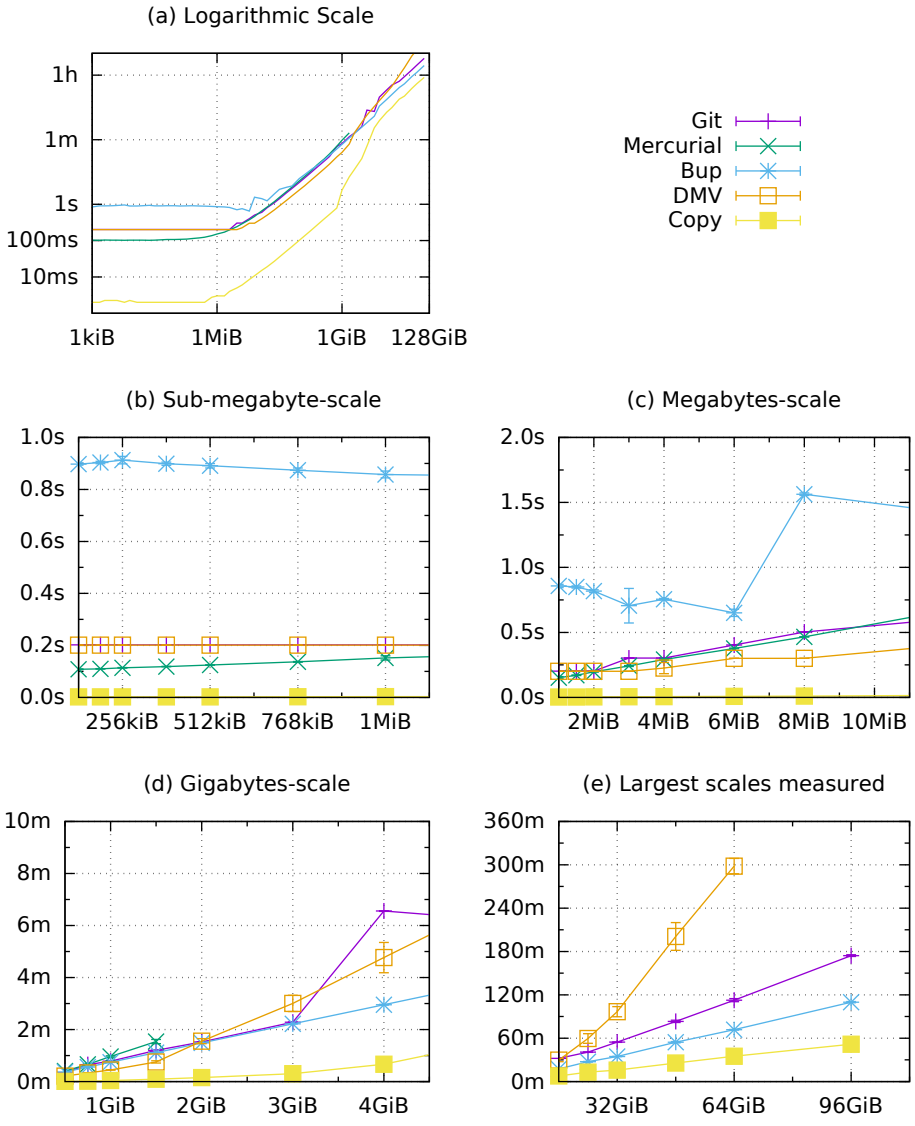
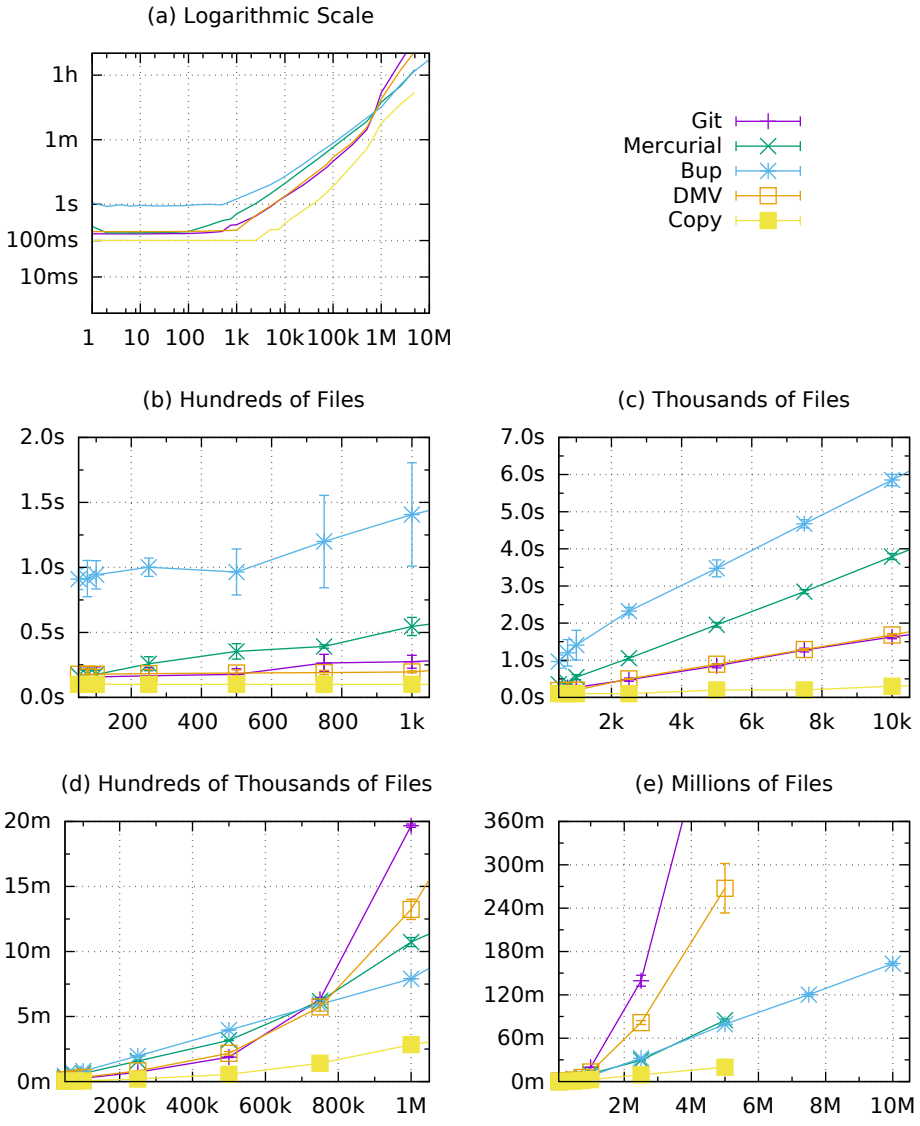


Figure 2: Wall-clock time to commit many 1KiB files to a fresh repository

Simplify to just largest sizes, convert to black & white

Subfigure (a) shows the full range on a logarithmic scale, while the others are linear-scale for specific ranges and include error bars.



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

Remove this table?

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