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

[Figure 1 about here.]

[Figure 2 about here.]

[Figure 3 about here.]

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

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] 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).

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 [2].

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].

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

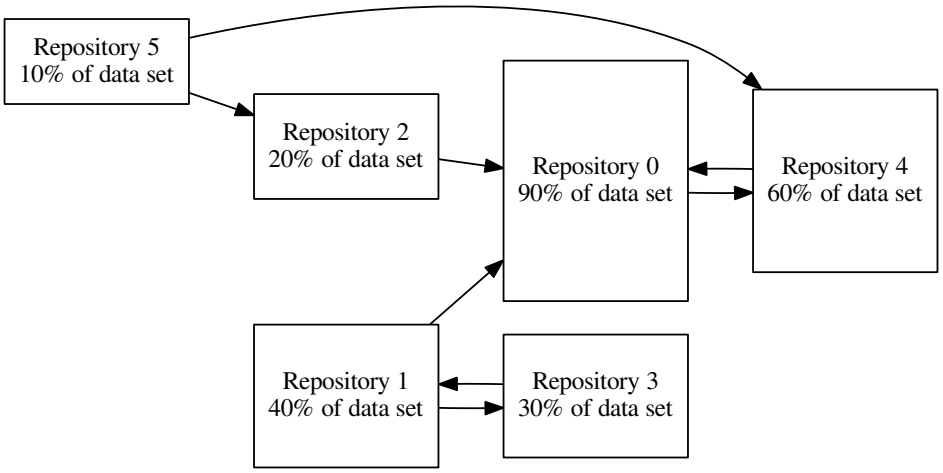


Figure 1: Repositories in an ad-hoc network

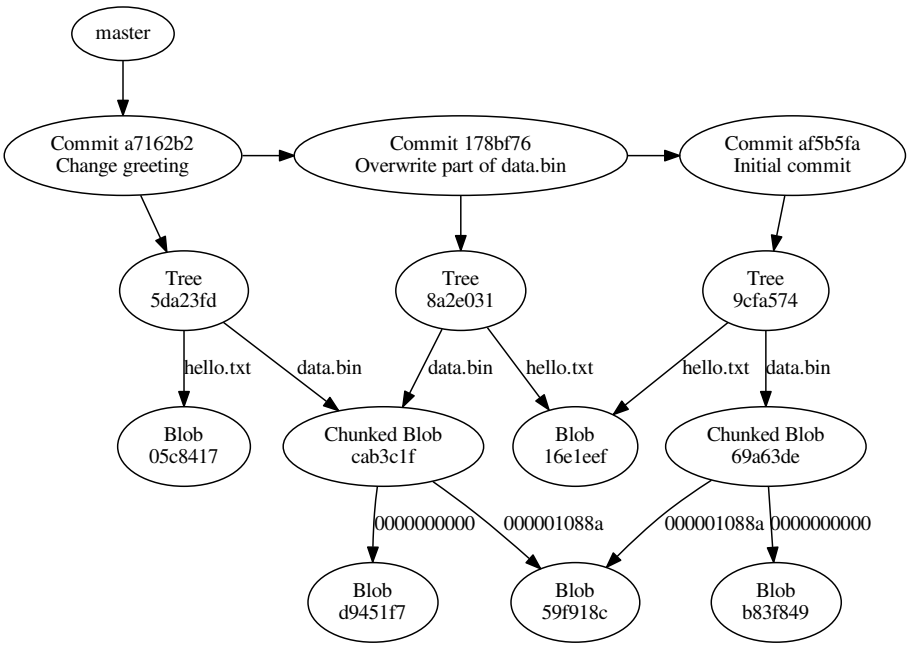
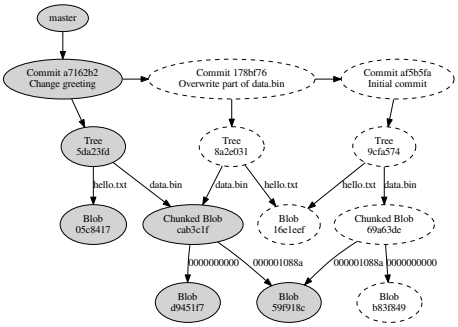
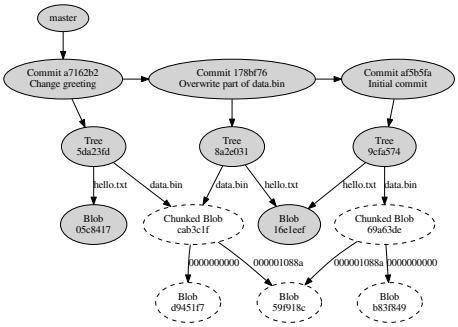


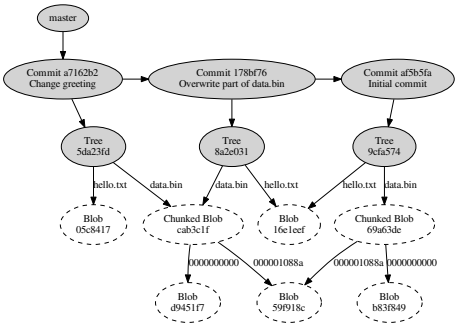
Figure 2: A simple DMV DAG with three commits



(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