Version Control for Large Files
Specification

Michael Murphy

August 2016

1 Idea

1.1 Main idea: Distributed version control for large data sets

Distributed systems tend to focus on maintaining a consistent view of a large data set, across many replicas, as it is updated from many sources. This is difficult.

By contrast, distributed version control wholly accepts inconsistency. Each replica has its own copy of the dataset, each can diverge by their own updates, and none is more valid than any other. There is no global concept of a most-recent version.

Version control makes the history of mutations explicit, and provides tools for one replica to copy changes from other replicas and incorporate them into its own history. In terms of the CAP theorem, distributed version control systems such as Git or Mercurial go all-in on availability and partition tolerance. Each replica is apart from the others by default, and each replica is always available to read and make updates on its own. Inconsistency is made explicit, and reconciliation is done manually by the user with help from difference comparison tools and merge algorithms.

Version control is a powerful tool for maintaining important data sets, usually source code. It makes it easier to keep backups, synchronize between computers, and collaborate with other users. The main limitation of current version control systems is that they are designed for source code, which as data sets go is relatively small, tens or maybe thousands of text files that are kilobytes in size. Adding larger binary files, such as media, causes existing version control systems to become sluggish and wasteful of disk space.

Our goal is to apply the distributed version control concept to data sets that are too large for existing version control systems. These data sets might:

- Contain individual files in a wide range of sizes, from text files of a few kilobytes to videos of several gigabytes
- Contain files in large quantities, perhaps millions of files

- Be too large as a whole to fit on a single conventional hard drive, up to multiple terabytes or petabytes

1.1.1 Accommodating large data sets in version control

We believe this might be achieved by starting with Git's cryptographic DAG data structure and:

- Adding facilities to break large files into smaller chunks for more efficient storage and comparison
- Relaxing the requirement that every replica store the entire history of the entire data set, allowing replicas to focus on particular subsets of the data set or particular slices of its history

Allowing each replica to only store portions of the data set will compromise availability as well as consistency. However, replicas will always be able to record updates to the data they do have. And by keeping track of what data is available at neighboring nodes, the replica can fetch and cache requested data as needed.

In much the same way that distributed version control makes consistency and inconsistency explicit, relaxing the full-history requirement makes availability explicit as well.

Replicas will be able to choose their own balance of how much data to make available locally, based on available storage space and latency to neighboring replicas.

1.2 Expected Benefits

We believe such a system could be flexible enough to be used at various scales.

- Individual users might use it to maintain a collection of important documents, photos, and media, making it easier to keep up-to-date backups and to synchronize between computers, mobile devices, and removable drives.
- Professional users that work with files too large for traditional version control, such as graphic designers, audio engineers, or maybe even video editors, might finally be able to adopt a version-control workflow.
- Corporate or government users might use it to maintain large archives of data with full history.
- Far-flung networks with high-latency or rare connectivity, such as remote wildlife sensors or Mars rovers, could use it to manage and synchronize data.

1.3 As an abstraction

In a sense, what we want to build is an abstraction for tracking a data set, its differing versions, and its history as a cohesive whole, even though it may be physically spread across many nodes.

Just as version control is a tool for managing snapshots of a codebase, this will be a tool for managing those snapshots when they become too large to store on a single disk and must be offloaded to removable drives or the cloud.

We are thinking about data across a number of dimensions:

Coverage of data set How much of the data set is available locally or in neighboring nodes?

Coverage of data history How much of the data set's history is available locally or in neighboring nodes?

Divergence of versions How many different branches has this data been forked into, and how different are they?

Number of replicas How many times is the data replicated across neighboring nodes? Is any data in danger of being permanently lost?

Availability of or distance to replicas Of the replicas available, how available are they? What is the bandwidth of the connection to the neighboring nodes? What is the latency?

Rather than strive for automated consensus or availability, we want to make the trade-offs explicit. The goal is to track and visualize the data in these dimensions for the user, so that they can make informed decisions about how to access the data they need.

Ideally, this system will be a generalized and flexible piece on infrastructure that others can use to build more automated systems for specific situations.

1.4 Main principles

- Data must never be lost accidentally.
- However, history may be deliberately truncated to save space, and sensitive data may be deliberately redacted.
- Data integrity must be verifiable: The system must be able to detect errors and, if possible, repair them.
- Changes to the dataset should be tracked, versions should be explicitly labeled, and history should be kept.
- Like with distributed version control, updates can be made independently and merged later. Different sites can have different versions. Updates (commits) and synchronization are deliberate, explicit, and manual.

1.5 Important assumptions

- Contact between repositories is intermittent. Repositories may be on removable drives or mobile devices. Updates may require physical connection and reconnection. It is important to track the state of other repositories, so that the user can know what needs to be synchronized.
- Assume all actors are honest for now. No malicious components.
- However, components can and will fail. The system must discover and recover from errors (checksums, replication).

1.6 What the system should not do

We want to focus on the problem of storing file history and synchronizing files between replicas. We should be careful not to expand across the wrong abstraction boundaries or to try to do too much. In particular:

- We do not want to reinvent the filesystem. The system should place and update files on the filesystem (or offer a filesystem view, such as with FUSE) for applications to use normally. Applications such as editors should not have to be rewritten to use our system.
- We do not want to create new exotic file formats. We believe that the classic tree of files is our best chance for long-term storage.
- We hope this system could eventually be used as a piece of infrastructure on which to build useful applications. It should not incorporate functionality that would better be left to an application.
- We do not want to deal with media metadata and categorization. Metadata and categorization is best left to the applications that produce and consume those media formats. We will merely provide the storage.
- However, knowledge of media formats might be used for behind-the-scenes optimization such as more efficient compression. E.g. recognizing that only tag data has changed in an audio file.

2 Architecture

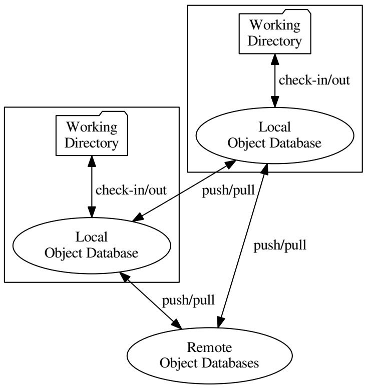
2.1 Key properties of distributed version control

2.1.1 Free-form network architecture

Because replicas are autonomous and there is no global most-recent state, there is no need for complicated network membership schemes. Replica network topologies reflect the connections between their users and their workflows.

Many small projects use a hub-and-spoke topology, designating one replica as the main replica, and others pull from it and push to it.

Figure 1: Distributed Version Control



Large projects such as the Linux kernel can form hierarchies of maintainers, each in charge of specific subsystems.

2.1.2 Working directory and plain local file access

Key advantage: applications access and edit files normally through the filesystem. Applications do not need to be rewritten to use the data.

Disadvantage: double the disk space.

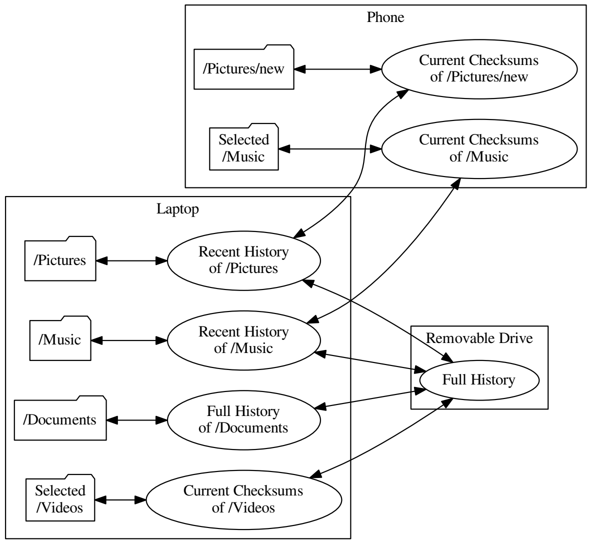
Possible solution: virtual working directory (e.g. via FUSE) as a copy-on-write snapshot of objects in the database.

2.2 Possible workflows

2.2.1 Personal Workflow

- Repository on removable drive stores current version of whole data set, and historical versions as far back as space will allow.
- Second removable drive repository configured the same way for redundancy.
- Laptop has several partial repositories + working directories:
  - Full history of `/Documents`
  - A year or so of `/Pictures`
  - Recent history of `/Music`
  - No history and selected `/Videos`
- Phone has several thin repositories + working directories:

Figure 2: Personal Workflow



- Selected `/Music` checked out with no history. Just used to sync.
- Selected `/Pictures` are checked out with no history, to be displayed on the phone.
- A `/Pictures/new` directory is checked out as the phone's new directory in which to put photos as they're taken. It stores only history that has not been synced. A new state is pushed to the laptop. The user categorizes the photos on the laptop, commits the new state, and pushes it to the phone. The selected photos show up in the categorized areas, and the `new` directory is emptied.

2.2.2 Corporate/Scientific Workflow

- Main repository uses a DHT as a massive backing store for its object database, keeps all history.
- Users check out pieces of the data set as needed, work with it, and push their changes back.

2.2.3 Remote Sensors Workflow

- Archive on computers in office stores all data
- A directory in the hierarchy is designated for new data from each sensor

Figure 3: Corporate Workflow

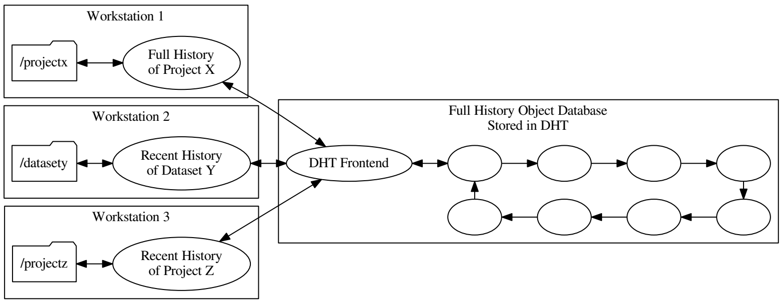
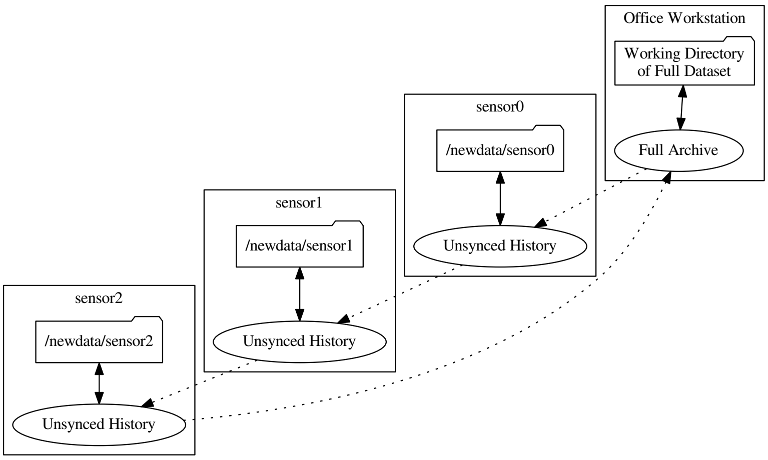


Figure 4: Remote Sensor Workflow



- The sensor has a thin repository + working directory for just its own new data directory. It commits new data.
- A courier has a thin repository on their phone, holding just the new data directories for all sensors. The courier visits a sensor and connects to it, pulling in the new data. They visit another sensor and pulls in its data too.
- The courier gets back to the office and syncs with the main archive.
- The archive now has a state with new data from all visited sensors. A process moves the new data to a permanent directory, and commits that new state.
- The courier syncs the phone again, this clears the space on their phone.

- When they visit the sensor again, it syncs and merges, deleting the data that was stored safely, and creates a new state with just new data.

3 Design

3.1 Main inspiration: Git and its DAG

3.1.1 Start with data structure

If the data structure is right, the rest should follow.

- Data structure will be based on Git's DAG: immutable blobs, trees, and commits that are stored in a content-addressable object database, and referred to by cryptographic hash.
- The immutability, cryptographic hashing, and DAG data structure make it easy to synchronize between repositories and check data integrity.
- Like Git, current state of local branches, and known state of remote repositories will be pointers to commits in the DAG.

Can we take Git's DAG and make it more efficient at handling large binary files, and can we rewrite its algorithms to be tolerant of missing blobs?

3.1.2 Differences from git

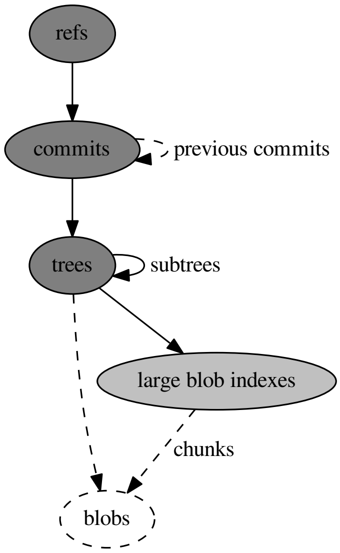
- Partial repositories: Individual repositories need not store entire history. Several repositories can work together to spread the data across many machines.
- Partial checkouts: Working directories need not check out entire data set.
- Support for large binary files (gigabytes): Large files can be split into chunks and spread across repositories.
- Repositories work more closely together to form a whole:
  - Repositories need to know not just the state of their neighbors but what data each one actually stores.
  - Should be able to visualize how complete data set storage is and how well data is replicated to protect against data loss.

3.2 Modified Git DAG

We start with the Git DAG and modify it (Figure 5) to accommodate our desired features.

- Unlike Git, the repository is not required to store all objects in the DAG. A repository needs only to include the bare minimum of objects to record the state of its references. These include a reference to the current commit,

Figure 5: Modified Git DAG



that commit object, and all of that commit's trees. Large blob indexes should also be included. These required objects are shaded grey. Those connections that can be left dangling by not storing the referenced object are shown with dashes.

- Large blobs in the object database can be broken into *chunks* to make them easier to store, sync, and transfer. We introduce a new *large blob index* object type to point to the chunks that make up the larger blob (shown light grey). Chunks themselves are just blobs.

3.3 Possible variants of partial repositories

Commits and trees by themselves carry information about the structure and history of the data, the metadata of the repository. They are a kind of "backbone" that supports the actual data.

- A full archive could store all history, just as in Git.
- Several partial archives could work together to store the full history.

  Can configure which repository holds which data according to storage size and network topography. Old infrequently-accessed versions could be kept on larger, slower data stores.
- A shallow repository could store only a few recent versions, to be compared against the working directory or to be restored to correct mistakes.

However, it would still have the full “backbone,” so it would know what blobs would be needed to checkout different states.

- A “backbone-only” repository could store no blobs, just the working directory and the “backbone.” This would allow the working directory to detect changes, and it could create new commits, only storing new blobs until they were pushed.
- A repository could also focus on a particular subtree, storing blobs for its entire history, but none of the blobs outside. This would allow detailed work on one part of the larger data set.
- Which blobs to keep could be configurable by rules that work along dimensions of time and parts of the tree.

```
/       last 1 versions
/foo    last 5 versions
/bar    all versions
/baz    no versions
```

- We could also provide tools to recommend which blobs to store based on usage frequency, available storage space, and repository availability.
- Perhaps it is not even necessary to store the full backbone. The backbone will be tiny compared to the whole data set, but for large (millions of files) or long-lived data sets, the full backbone could be a burden on small, focused repositories.

3.4 Other deviations from the Git data model

- Trees and commits may hold more information than in Git.
  - Objects could include a measure of the cumulative size of all the objects they refer to, so that repositories could make decisions about space trade-offs, and choosing to drop unneeded blobs to save space.
- Remote pointers will hold more information than with Git.
  - Because we cannot assume that every repository has all objects, remote pointers must also keep metadata on which blobs are available at which repository. So that it knows where to look if needed.
  - This availability data will be used to gather health metrics about what parts of history and hierarchy are safely replicated over many stores, and which are in danger of being lost.
- All algorithms will have to be written around the idea that data might not be available immediately, or at all.
  - Repositories and working directories will work with what is available locally, and what is available locally will be chosen by the user based on what they need to work on now.
  - When those needs change, it will be easy to push and pull blobs to and from other repositories.

- However, if a blob is lost, it is lost. This should never happen by accident, but it may happen deliberately by dropping old history to save space, or to deliberately expunge sensitive blobs from the records. If the algorithms can deal with missing objects locally, then they should naturally also be able to handle objects that are missing completely.

4 Implementation

- Object database storage should be pluggable. Flat files by default, but should be able to use a DHT as a large highly-available object store.
- Should be able to sync with a phone, either with an on-phone app, or via USB mount of filesystem.

5 Related Works

5.1 Distributed storage and synchronization systems

5.1.1 Camlistore

Camlistore [4] is an open-source project to create a private long-term data storage system for personal users. It allows storage of diverse types of data and it synchronizes between multiple replicas of the data store. However, it eschews normal filesystems and creates its own schemas to store various media.

5.1.2 Dat Data

Dat [7] is an open-source project for publishing and sharing scientific data sets for research. This project has a lot of overlap with ours, and several of the core ideas are similar, including breaking files into smaller chunks, and tracking changes via a Git-like DAG. However, their focus is different. The Dat team is concentrating on publishing research data, and making that specific task as simple as possible for non-technical researchers who might not be familiar with version control. By contrast, our project operates at a lower level of abstraction, offering the full power of version control in a very general way, exposing and illuminating the complexities rather than trying to hide them or automate them away.

Where Dat focuses on publishing on the open internet, we focus on ad-hoc networks and data that may be private. Where Dat has components for automating peer discovery and consensus, we work at a lower level, trying to perfect and generalize the storage aspect first. Dat seems to assume that data sets will be small enough to fit on a typical disk on a workstation, while we want to scale even larger.

We hope that our system could be used as a base to build something like Dat, but we intend to create something even more general than the Dat core.

5.1.3 Eyo

Eyo [9] is system for storing personal media and synchronizing it between devices. It utilizes a Git-like content-addressed object database behind the scenes, but it works more like a networked filesystem than version control. It focuses on organizing media by metadata, which requires agreement on metadata formats, and it requires applications to be rewritten to access files via Eyo rather than the filesystem, both of which are thorny and ambitious problems. We prefer to focus purely on storage and synchronization.

5.1.4 git-annex and git-media

Git-annex [5] and git-media [2] are open-source projects that extend Git with special handling for larger files. Both store the metadata of larger files in the normal Git repository and then store the files themselves in a separate location. Git-media stores all the larger files in a separate data store which may be remote. Git-annex is more flexible. Annex files may be spread across several different remote repository clones or data stores, and git-annex has features for tracking the locations of annex files in different remote repositories and moving them from one repository to another. These tracking and distribution features are very similar to our goals. However, git-annex is not quite as flexible as we aim for in our system. It considers the large files atomic units, and it does not break them into smaller chunks for de-duplication. Also, because metadata is processed by Git, it has the same limitations that Git does. All repositories must have all metadata, and performance suffers when metadata is too large to fit into RAM.

5.1.5 IPFS: The Interplanetary Filesystem

IPFS [1] is an open-source project to create a global content-addressed filesystem. By its global nature, all files are stored together, publicly, in a global network of nodes with global addressing. IPFS should be an excellent resource for storing published information, but we wanted to work on a smaller, more private scale with discrete data sets. We want individuals and organizations to be able manage their own data stores privately on their own hardware.

It should be noted that IPFS does have support for storing private objects by way of object-level encryption. However, this seems wasteful of disk space, since small changes in the plain text of a file would completely change the ciphertext, leaving no way to compress the redundancy.

5.1.6 Kademlia

Kademlia [6] is an advanced distributed hash table system that updates its network topology information as part of normal lookups. It is an advanced piece of infrastructure, but like other distributed hash tables, it focuses on system-wide consistency, rather than the version-control paradigm we are trying to achieve.

5.2 Content-Addressed Storage and Backup

5.2.1 Boar

Boar [3] is an open-source project to create a version control system for large binary files. It is one of the main inspirations for our project. It stores file versions in a content-addressed way, and provides de-duplication for large files that only change in small pieces, and it can truncate history to reclaim disk space. However, Boar retreats to a centralized version control paradigm, with a central repository that working directories must connect to to check files in or out. We want to provide the advantages of Boar in a flexible distributed version control model. Boar also has practical limitations on repository size and number of files. Repositories are assumed to fit on one disk volume, and file metadata is assumed to fit into Ram. We aim to overcome both of those limitations.

5.2.2 Bup

Bup [8] is an open-source file backup system that is based on Git's repository format. A Bup backup is a valid Git repository and it can be read by Git, but Bup is a separate program written from scratch to read and write files to Git's pack file format directly, skipping Git's separate store and pack steps that use double the disk space. It has many features that we want for our low-level storage of the object database. It breaks files into chunks by rolling checksum, and it has considerations for metadata that is larger than RAM. However, it is locked into a backup-based workflow. History is linear and based on clock time of backup. And it assumes that the whole data set and the whole repository can fit onto one filesystem.

References

[1] Juan Benet et al. IPFS: The interplanetary filesystem. GitHub, 2014. `https://github.com/ipfs/ipfs`.

[2] Scott Chacon, Alex Lebedev, et al. git-media. `https://github.com/alebedev/git-media`.

[3] Mats Ekberg et al. Boar. `http://www.boarvcs.org/`.

[4] Brad Fitzpatrick et al. Camlistore is your personal storage system for life. `https://camlistore.org/`.

[5] Joey Hess et al. git-annex, 2015. `http://git-annex.branchable.com/`.

[6] Petar Maymounkov and David Mazières. *Kademlia: A Peer-to-Peer Information System Based on the XOR Metric*, pages 53–65. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[7] Max Ogden, Mathias Buus, Karissa McKelvey, et al. Dat Data. `http://dat-data.com/`.

[8] Avery Pennarun, Rob Browning, et al. bup, it backs things up. `https://bup.github.io/`.

[9] Jacob Strauss, Justin Mazzola Paluska, Chris Lesniewski-Laas, Bryan Ford, Robert Morris, and Frans Kaashoek. Eyo: Device-transparent personal storage. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 35–35, Berkeley, CA, USA, 2011. USENIX Association.