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. 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 allows easy backups, synchronization between computers, and collaboration between users. The main limitation of current version control systems is that they are designed for source code, which as data sets go is relative small, tens or maybe thousands of text files that are merely 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 larger data sets. These data sets might:

- Contain individual files in a wide range of sizes, from single kilobyte text files to several gigabyte videos
- 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

We believe this might be achieved by:

- 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

1.2 Expected Benefits

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

- Single users might use it to maintain their collection of important documents, photos, and media, making it easier to keep up-to-date backups and to synchronize between computers and mobile devices.
- Professional users that work with files too large for traditional version control, such as graphic designers, audio engineers, or video editors, might finally be able to adopt a version-control workflow.
- Corporate or government users might 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 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.

1.4 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).

1.5 Main inspiration: Git

Can we take the elegance of Git's block chain and generalize and distribute it so that it works well with large files, and so that all the data does not have to be present in every repository?

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 files: Large files can be split into chunks and spread across repositories.
- Repositories work more closely together to form a whole:
  - Repositories should be aware of each other and what pieces of data each one stores.
  - Should know where to find data when needed.
  - Should be able to visualize how complete data set storage is and how well data is replicated to protect against data loss.

Other considerations:

- Stored data set should be a normal file tree, with history, like any normal version control system.
  - We do not want to reinvent the filesystem.
  - We do not want to deal with media metadata and categorization. That is a whole other problem.
  - This is meant for long-term storage. Classic files are the best chance for that. New data schemas are also a whole other problem.
  - However: it may be possible to generalize, and allow flexible aggregation besides trees.
- 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 Core data structure: DAG

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

Again, the main inspiration is Git.

- 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.

2.2 Like version control, interactions based on files and check-ins

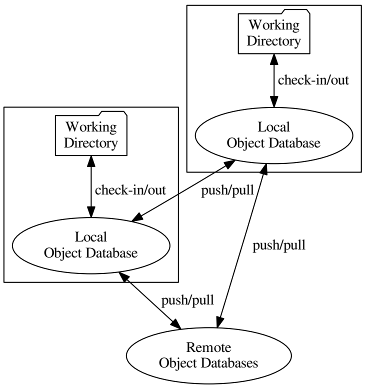
- Like Git, there will be an object database that stores the DAG, and a working directory where files are checked out for normal use and editing.
- We prefer a version-control style interaction, checking files in and out explicitly to the filesystem where applications can work with them as normal files. Again, we do not want to reinvent the filesystem.

  However, to save space when working with large collections, perhaps a FUSE filesystem could present a virtual working directory that is actually a copy-on-write snapshot of files in the object database.
- Like Git, no repository is inherently more important or more central than any other. Importance given to a certain repository or specific push-pull flows will be the result of the user's workflow, not of the underlying system.

2.3 Unlike Git, repositories will not be required to store all objects

- 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 carries the actual data.
- We could have all repositories keep this "backbone," but allow them to selectively store only certain blobs. This will allow repositories to balance space vs completeness of history.
  - A full archive could store all history, just as in Git.
  - Several partial archives could work together to store the full history.

Figure 1: Distributed Version Control



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.

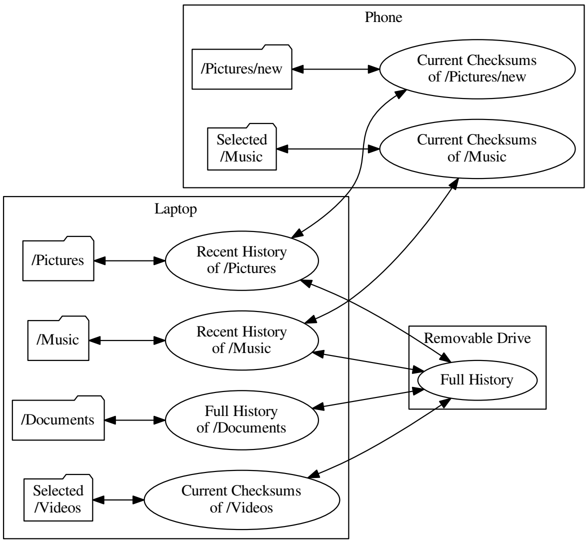
- Trees and commits may hold more information than in Git.
  - Objects could include a measure of the cumulative size of all the objects they refer to, so that repositories could make decisions about space trade-offs, and choosing to drop unneeded blobs to save space.
- Remote pointers will hold more 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 permanently.

2.4 Possible workflows

2.4.1 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:
  - Selected `/Music` checked out with no history. Just used to sync.

Figure 2: Personal Workflow



Phone
/Pictures/new
Current Checksums of /Pictures/new
Selected /Music
Current Checksums of /Music
Laptop
/Pictures
Recent History of /Pictures
/Music
Recent History of /Music
/Documents
Full History of /Documents
Selected /Videos
Current Checksums of /Videos
Removable Drive
Full History

  - 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.

2.4.2 Corporate/Scientific Workflow

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

2.4.3 Remote Sensors Workflow

- Archive on computers in office stores all data
- A directory in the hierarchy is designated for new data from each sensor
- The sensor has a thin repository + working directory for just its own new data directory. It commits new data.

Figure 3: Corporate Workflow

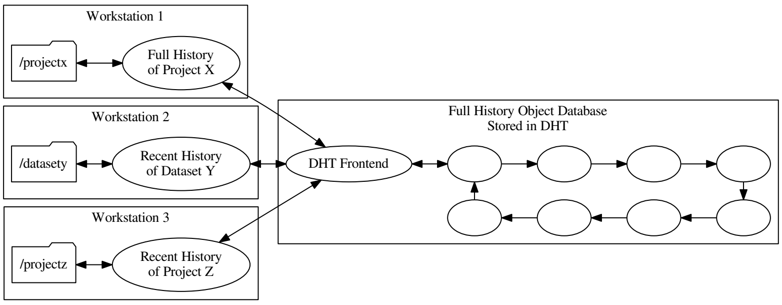
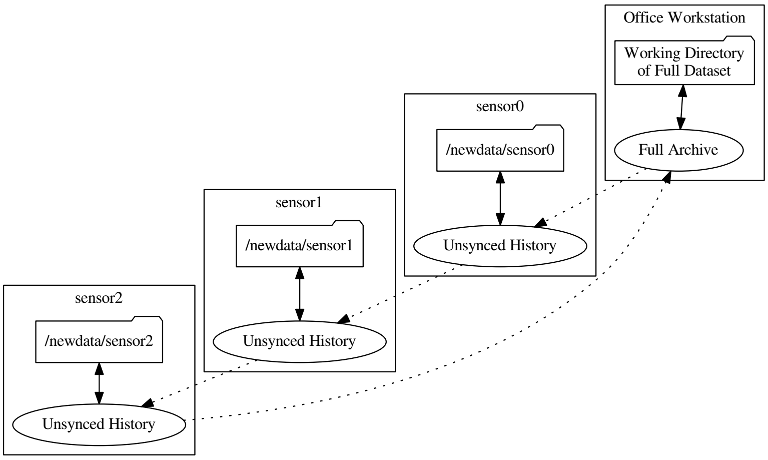

Workstation 1
/projectx
Full History of Project X
Workstation 2
/datasety
Recent History of Dataset Y
Workstation 3
/projectz
Recent History of Project Z
Full History Object Database Stored in DHT
DHT Frontend

Figure 4: Remote Sensor Workflow

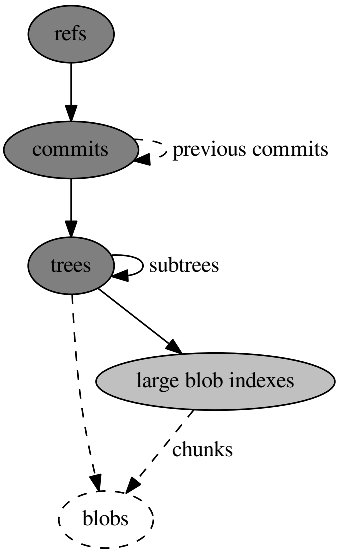


- A courier has a thin repository on their phone, holding just the new data directories for all sensors. The courier visits a sensor and connects to it, pulling in the new data. They visit another sensor and pulls in its data too.
- The courier gets back to the office and syncs with the main archive.
- 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 Modified Git DAG

Figure 5: 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, that commit object, and all of that commit's trees. Large blob indexes should also be included. These required objects are shaded grey in Figure 5. Those connections that can be left dangling by not storing the referenced object are shown with dashes in Figure 5.
- 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 in Figure 5. Chunks themselves are just blobs.

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.

References

[1] 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.

[2] 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.