COS 316

Content Addressable Storage & Git

Amit Levy

Previously, we talked about the UNIX file system's five layers and the way they use naming to achieve particular goals. Each of these layers is an example of a location-based naming scheme, where names relate to the *location* of their value in underlying layers:

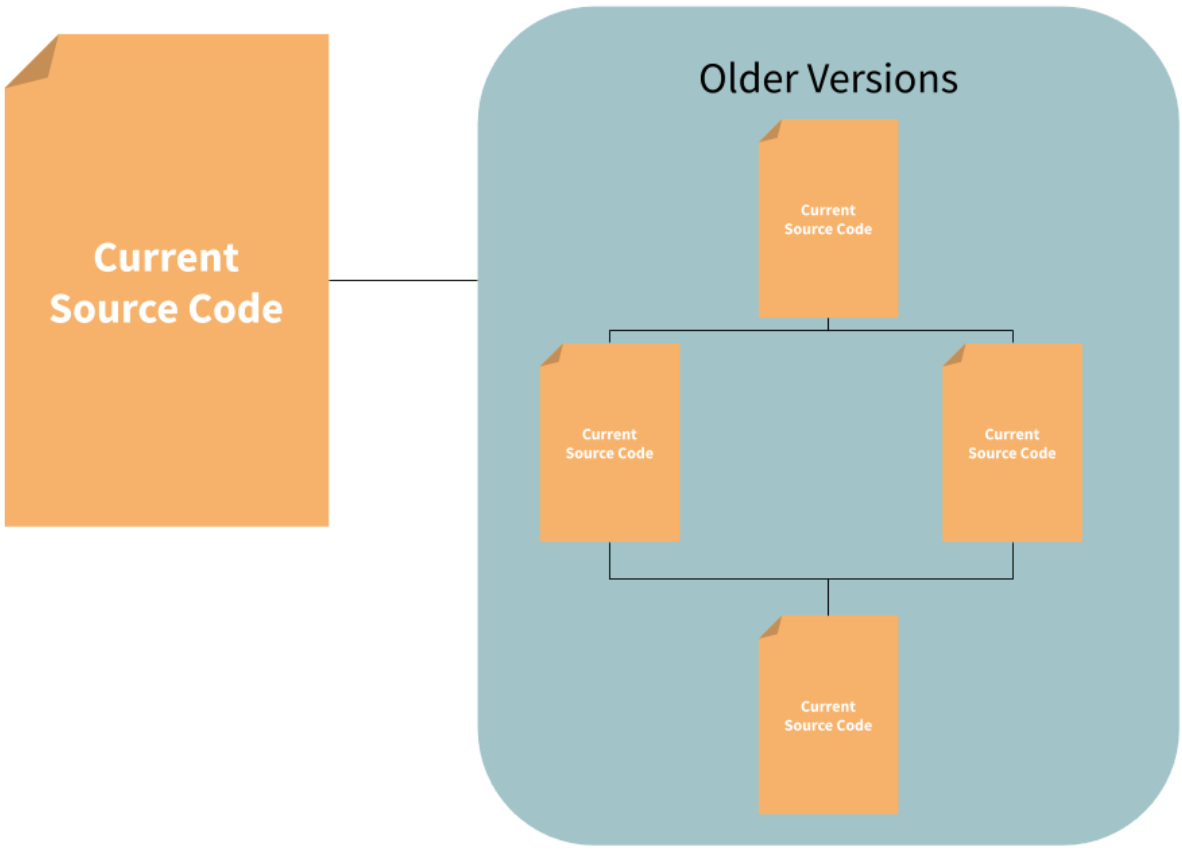
- The block layer names blocks based on the *order* in which they appear on disk
- The file layer names files based on *where* to find their blocks
- The directory layer names files (via inode numbers) based on which directory they are *located* in
- The absolute path name layer provides the *location* of the root directory

Indeed, the UNIX file system takes a location-centric view of what it stores---its role is to map logical names to the locations on disk they refer to. That's the point!

This makes sense for a local file system, but when might location-based naming be problematic?

Next, we'll look at Git to see when location-based names fall short and how we might use *content*-based names to solve those short-comings.

Git & Distributed Version Control



Version Control History

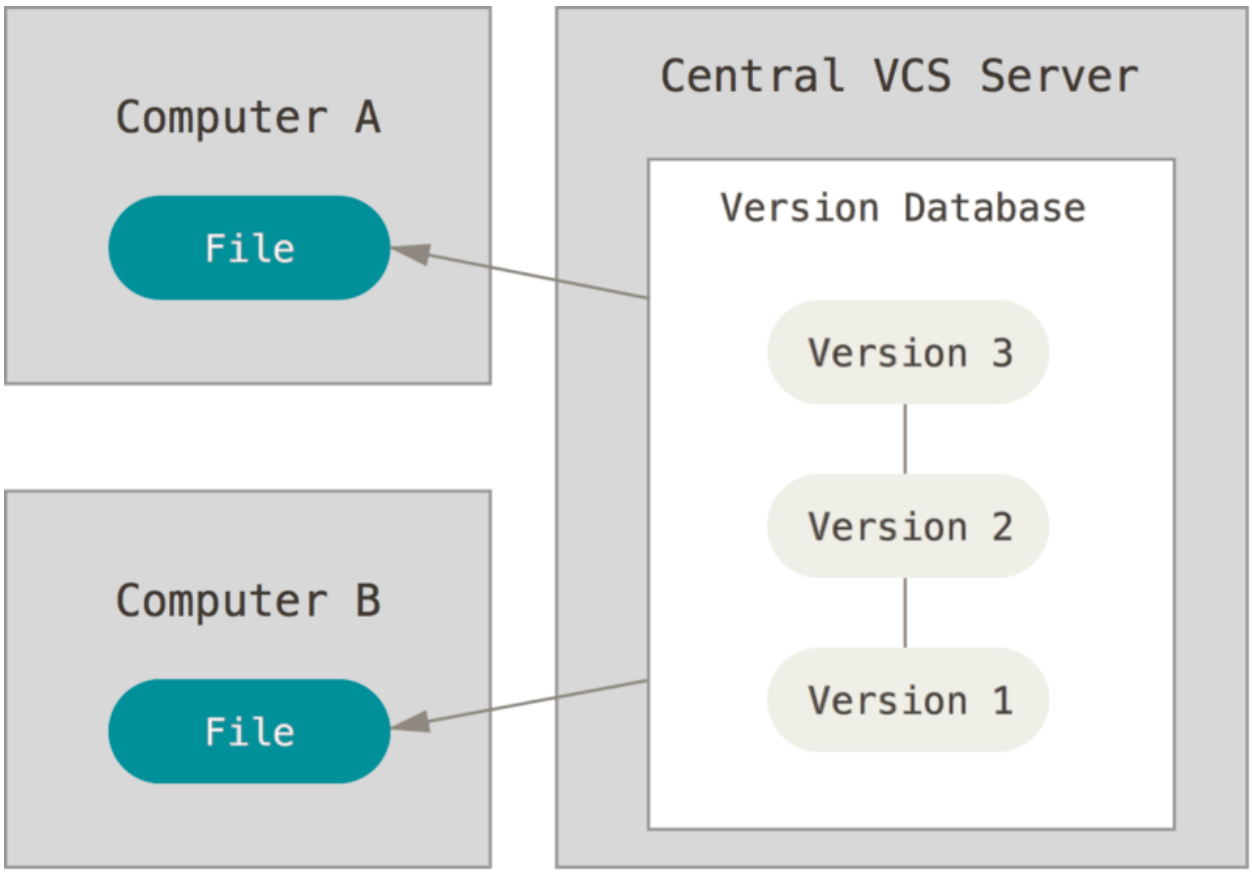
Version control has been around since the early 70s, when the first version control systems were developed in the UNIX group to organize development of UNIX and other operating systems.

- Local Only
    a. SCCS -- 1972 developed by early UNIX developers
    b. Revision Control System (RCS) -- 1982 developed by GNU project
- Client-Server
    a. Concurrent Versions System (CVS) -- 1986, originally as a front end to RCS for client-server to collaborate on ACK (Amsterdam Compiler Kit)
    b. Subversion (SVN) -- 2000, basically a rewrite of CVS

- Distributed
    a. BitKeeper --- 2000, developed to deal with Linux's development model
    b. Mercurial & Git --- 2005, open source alternative to BitKeeper for Linux kernel development when BitKeeper stopped being free

In centralized version control, a central server holds the ground truth for each file in the repository. Clients "check out" individual files to edit and "commit" their changes back to the central server to update the new versions. Importantly, it's not expected that there will be more than one checkout of the same file or that changes amongst different developers will conflict.

The main role of the version control system in this model is to efficiently store older versions of the same file, and to coordinate individual file locking and merging.

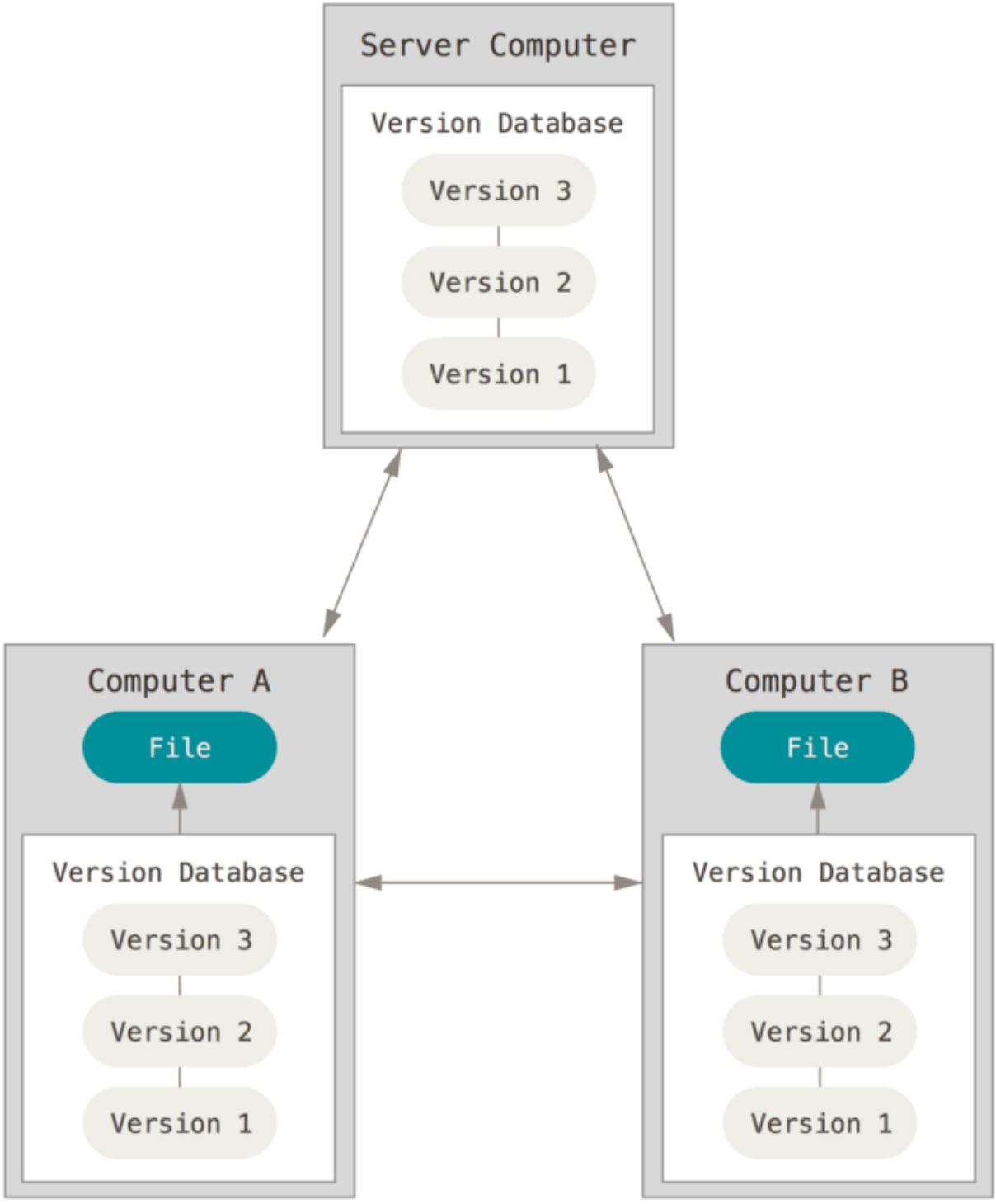


Distributed Version Control

Centralized version control has several shortcomings:

- Are the set of files in the central repository collectively valid? What if two developers change two files, concurrently, that rely on each other?
- In a non-centralized software development community (e.g. the Linux ecosystem Git was developed for), using a centralized repository limits how developers can collaborate.
- What happens if data on the single canonical version is corrupted? How do we recover?

Distributed version control systems (DVCSs), such as Git, address these shortcomings. As we'll see, one of the mechanisms that allow DVCSs to do so is a content-based naming scheme.

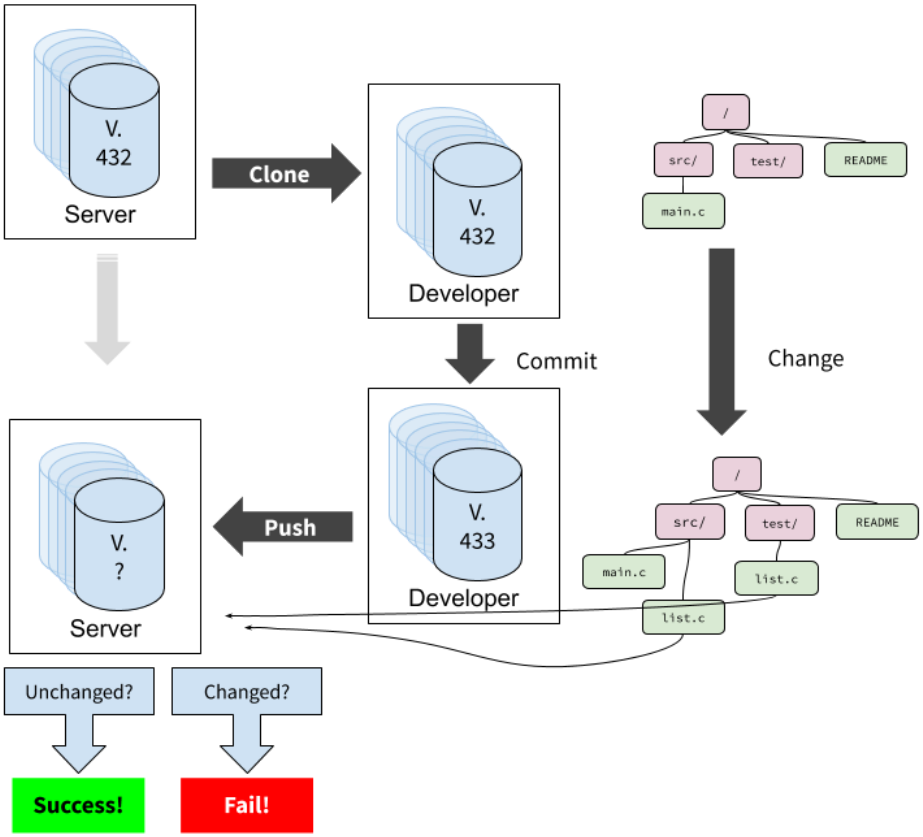


Specifically, distributed version control systems differ from centralized version control systems in two important ways:

First, in contrast to centralized version control systems, distributed version control systems do not have an inherent notion of a canonical set of files, e.g. stored on a centralized server. Instead, each developer has a local snapshot of the repository, including previous versions of the files in the repository.

Second, the unit of a commit in DVCSs is an entire source code tree---a whole directory structure, as opposed to a single file. This is important because it allows developers to update their source code atomically---the source code tree is always in a state that *some* developer intended at *some* point.

Abstract Workflow Example:



1. A developer clones repository and gets a source code tree
2. They create a different tree by modifying or deleting files, creating new ones, etc
3. The developer creates a commit, which names the source code tree as well as the previous commit (and transitively the previous “state” of the source tree)
4. The developer pushes their new tree to the server, no synchronization needed

5. The developer tries to atomically update the commit the server considers most recent, using compare-and-swap
    - If success the update succeeds, we're done
    - If not, it's because somebody else has updated the reference in the meantime. The developer pulls new commits and the related trees, and "merges"[1]

How might we do this using the UNIX file system?

We need some way to succinctly name files, trees, commits, etc in such a way that's easy to compare them.

Content-based Addresses

Git uses a naming mechanism called "content-based addressing" where the name of an object is derived from it's content. The requirements for such a scheme are that:

1. Names *succinctly* summarize the content---i.e. names should be much shorter than the contents
2. Different contents get unique names
3. The same content always gets the same name

Cryptographic hash functions satisfy these requirements, so they are *often* used for content-based addresses. A cryptographic hash function is a hash function where:

- Hashes have fixed lengths, usually 128-bits, 256-bits, 512-bits, etc (satisfies requirement 1)
- Hashes are deterministic (satisfies requirement 3)
- It is computationally "hard" to find two contents with the same hash value (satisfies requirement 2 with high probability)
- It is computationally "hard" to come up with contents that yields a specific hash value (not necessary, but doesn't hurt)
- Similar contents have dissimilar hashes (not necessary, but doesn't hurt)

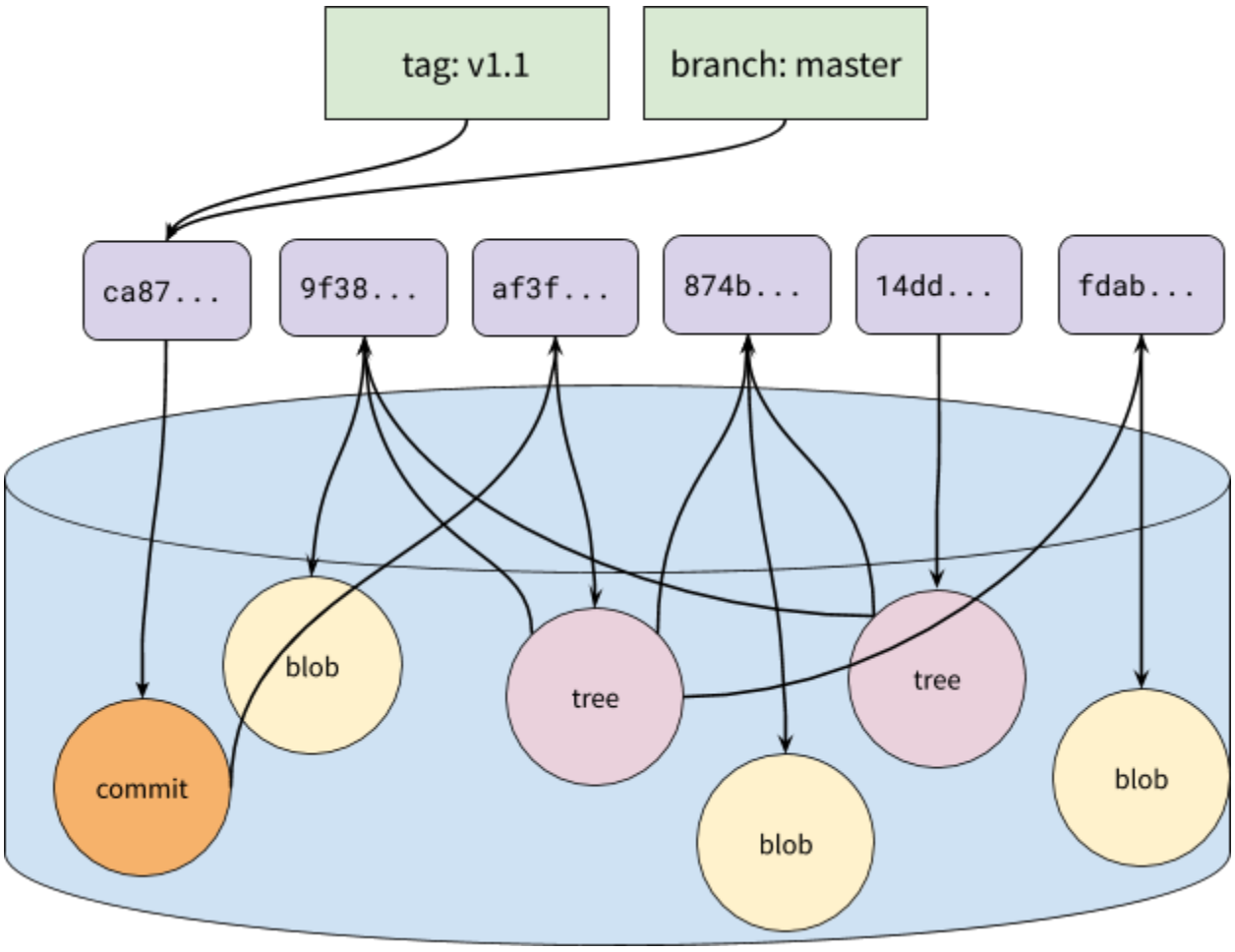
Git Internals

Git uses four layers that *employs* the UNIX file system as the underlying storage mechanism, but use a different kind of naming scheme called "content-addressable"

---

[1] Merging is actually just a convention, we can also "force push" and overwrite without first checking for previous changes, in which case we may erase those commits from the history.

Layer	Purpose
Object layer	Names blobs, trees, commits
Tree layer	Hierarchical human-readable names
Commit layer	History of related trees
Reference layer	Maps logical names to content-addresses



Object Layer

Objects are the basic storage unit in Git, similar to blocks in the UNIX file system. All data---including blobs, trees and commits---are stored in objects, providing a single storage abstraction for nearly all of Git's layers.

1. Names

Object names are their content-based addresses. Specifically, Git uses the 160-bit SHA-1 hash[2] of the object contents, normally formatted as a 40-byte hex string.

An important property of using content-addressable names at the bottom layer is that it allows Git to *automatically* deduplicate identical content.

For example, suppose we're storing a new version of the source code, with only one minor change: we've only modified a single file. The rest of the files will remain unchanged, so when we store the new tree, rather than storing two copies each of the identical files, Git will know not to store a new copy of any object with an identical name.

2. Values

- Blobs: opaque binary blobs, the equivalent of files
- Trees: basically directories
- Commits: points to a tree and a previous commit

3. Allocation

Since names are generated from the content, we allocate a name by taking the SHA-1 hash of the content we're trying to store. This name is guaranteed (with very high probability) to be unique.

4. Lookup

At the heart of any content-based storage system, looking up values from content-based names needs to actually find the *locations* in which an object is stored. Git uses the file system to store objects, so it essentially reuses the layers we discussed last time to allocate space for values.

An object is stored in a file named with the object name in hex format inside the `.git/objects` subdirectory of a Git repository[3].

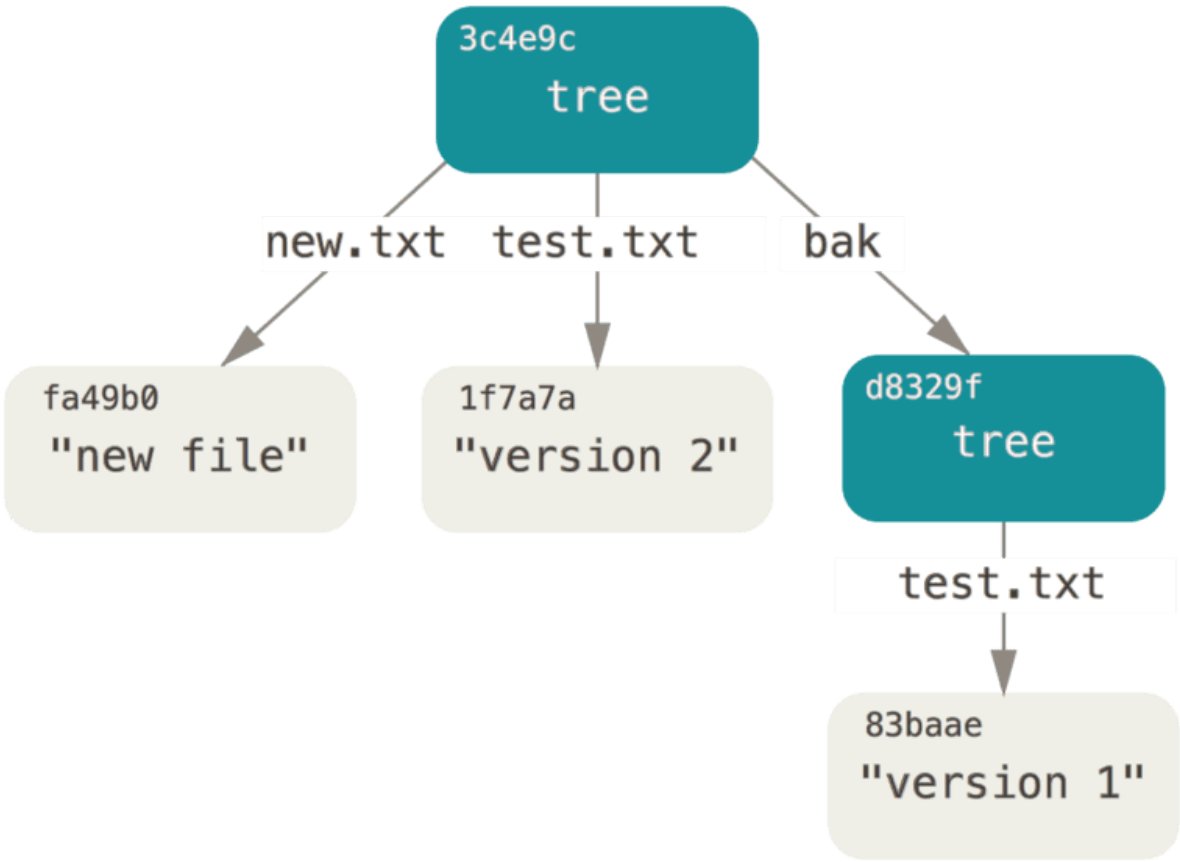
---

[2] As of early 2020 Git is transitioning to SHA-256 as SHA-1 is cryptographically weak and now deprecated

[3] Git also uses pack-files to compact many small objects into fewer larger files. See the Git internals chapter of the git book for details.

Tree Layer

Trees are similar to (and model) directories in the UNIX file system. They provide a hierarchy of trees and blobs that can be traversed using human-readable and meaningful names, just like a directory structure in UNIX.



Trees are stored in Git's object layer, so each tree is also an object with a content-based address.

5. Names

   The tree layer names Git objects using human readable names, similar to how directories in UNIX name files and other directories using human-readable names.

6. Values

   - Object addresses

- Type a permissions (a subset of UNIX file permissions)

7. Allocation

Names are allocated just like in UNIX---they are provided by the user. In Git, this generally happens because Git mirrors the “working directory”, which is a normal file system directory structure, into the Git data store.

8. Lookup

Trees have a storage format similar to directories, so lookup is similar. A tree is simply a list of entries, where each entry includes the human readable name of the object, metadata (permissions), and the address of the object. Instead of inode numbers, we use the content-based address of the object.

Commit Layer

The commit layer gives Git a way to express a version history of the source code tree. Commit objects contains

- A reference to the tree
- Metadata about the tree (the author of this version, when it was “committed”, a message describing the changes from the previous version, etc...)
- A reference to the previous commit

Commits are stored in Git’s object layer, so each commit is also an object with a content-based address.

9. Names

Names in the commit layer are fixed. Each commit has a set number of names that can be used to reference particular data in the commit. These include[4]:

- “Tree”
- “Parent”
- “Author”
- “Commiter”
- “Commit Message”

10. Values

- “Tree” - the tree object this commit references

[4] There are a few more possible names to do with commits that merge multiple version histories. See the Git book for a more detailed reference.

- “Parent” - zero or more parent commits in the version history
- “Author” - the author of the tree being committed
- “Commiter” - similar to the author, but can be different if a different developer authored the tree than created the commit
- “Commit Message” - a note about, e.g., what changes this commit include relative to previous versions

11. Allocation

Because commits have a fixed set of names, and values are supplied by the client, allocation trivial.

12. Lookup

Commits objects use a fixed format to store the commit struct, so lookup is just a matter of reading the struct format from the commit object.

Reference Layer

Commits, trees, and blobs have content-based names: 40-byte long hex strings that basically appear random. That’s useful for organizing and deduplicating objects internally, but it’s not very useful for humans to reference. For example, communicating which commit is “the most recent” or the “development branch” is hard to do with just a hash of its content because it’s always changing.

Git adds a final layer that provides human-readable names for an object at the top-level (rather than inside another Git object as in the tree layer).

When we refer to the “main” branch[5], we’re using a *reference* to name the most recent commit on that branch.

13. Names

These are human-readable names, e.g. “main,” “alevy/wip,”, “HEAD,” etc...

14. Values

These can be *any* Git object, though most typically the value is a commit object

---

5

15. Allocation

Reference names are assigned and managed by users. By convention there are a set of standard reference names:

- main: refers to the most recent “canonical” version of the source code
- HEAD: refers to the most recently committed tree on the local repository
- origin/*: refers to a reference on the “origin” repository, where this repository was cloned from

16. Lookup

References are stored as UNIX files in a special subdirectory of the .git directory and are simply files containing the address of the object they refer to