Git: A Case Study in System and Data-Structure Design

- Git is a distributed version-control system, apparently the most popular of these currently.
- Conceptually, it stores snapshots (versions) of the files and directory structure of a project, keeping track of their relationships, authors, dates, and log messages.
- It is distributed, in that there can be many copies of a given repository, each supporting independent development, with machinery to transmit and reconcile versions between repositories.
- Its operation is extremely fast (as these things go).
A Little History

• Developed by Linus Torvalds and others in the Linux community when the developer of their previous, proprietary VCS (Bitkeeper) withdrew the free version.

• The initial implementation effort seems to have taken about 2–3 months, in time for the 2.6.12 Linux kernel release in June, 2005.

• As for the name, according to Wikipedia,

  Torvalds has quipped about the name Git, which is British English slang meaning “unpleasant person”. Torvalds said: “I’m an egotistical bastard, and I name all my projects after myself. First ‘Linux’, now ‘git’.” The man page describes Git as “the stupid content tracker.”

• Initially, it was a collection of basic primitives (now called “plumbing”) that could be scripted together to provide the desired functionality.

• Then, higher-level commands ("porcelain") was built on top of these to provide a convenient user interface.
User-Level Conceptual Structure

• The main abstraction is that of a graph of versions or snapshots (called commits) of a complete project.

• The graph structure reflects ancestry: which versions came from which.

• Each commit contains
  - A directory tree of files (like a Unix directory).
  - Information about who committed and when.
  - Log message.
  - Pointers to commit (or commits, if there was a merge) from which the commit was derived.
Internal Structures

- The main internal components consist of four types of object:
  - **Blobs**: basically files of text or bytes.
  - **Trees**: directory structures of files.
  - **Commits**: objects containing references to trees, ancestor commits, and additional information (committer, date, log message).
  - **Tags**: references to commits or other objects, with additional information, intended to identify releases, other important versions, or various useful information. (We won't go into this today).
Commits, Trees, Files

- **Commits**
- **Trees**

Dashed lines link objects that are shared among commits

Blobs (files)

Last modified: Sun Apr 10 17:11:56 2022
Version Histories in Two Repositories

Repository 1

V1
  ` ` V2
  ` ` V3
  ` ` V4
  ` ` V7

Repository 2

V1
  ` ` V2
  ` ` V3
  ` ` V4
  ` ` V8

Repository 2 after pushing V6 to it

V1
  ` ` V2
  ` ` V3
  ` ` V5
  ` ` V9

V1
  ` ` V2
  ` ` V3
  ` ` V5
  ` ` V9
Major User-Level Features (II)

- Each commit has a name that uniquely identifies it to all repositories.
- Repositories can transmit collections of versions to each other.
- Transmitting a commit from repository $A$ to repository $B$ requires only the transmission of those objects (files or directory trees) that $B$ does not yet have (allowing speedy updating of repositories).
- Repositories maintain named branches, which are simply identifiers of particular commits that are updated to keep track of the most recent commits in various lines of development.
- Likewise, tags are essentially named pointers to particular commits. Differ from branches in that they are not usually changed.
Internals

- Each Git repository is contained in a directory.

- A repository may either be *bare* (just a collection of objects and metadata), or it may be included as part of a working directory (where it is named `.git`).

- The data of the repository is stored in various *objects* represented as files corresponding to data files (or other “leaf” content), trees, and commits.

- To save space, data in these objects is *compressed*.

- Git can *garbage-collect* the objects from time to time to save additional space and can *pack* bunches of objects into compressed files.
The Pointer Problem

- Objects in Git are represented as files. How should we represent pointers between them (e.g., from trees to files, or from commits to trees and other commits)?

- We want to be able to transmit objects from one repository to another with different contents. How do you transmit the pointers?
  - … Since the address of an object on one machine is very unlikely to have anything to do its address on another.

- We only want to transfer those objects that are missing in the target repository. How do we know which those are?

- We could use a counter in each repository to give each object there a unique file name. But how can that work consistently for two independent repositories?
Content-Addressable File System

- Could use some way of naming objects that is universal.
- We use the names, then, as pointers.
- Solves the “Which objects don’t you have?” problem in an obvious way.
- Conceptually, what is invariant about a version of an object, regardless of repository, is its contents.
- But we can’t use the contents as the name for obvious reasons (like trying to use a country as its own map.)
- **Idea:** Use a hash of the contents as the address.
- **Problem:** That doesn’t work!
Content-Addressable File System

• Could use some way of naming objects that is universal.

• We use the names, then, as pointers.

• Solves the “Which objects don’t you have?” problem in an obvious way.

• Conceptually, what is invariant about a version of an object, regardless of repository, is its contents.

• But we can’t use the contents as the name for obvious reasons (like trying to use a country as its own map.)

• Idea: Use a hash of the contents as the address.

• Problem: That doesn’t work!

• Brilliant Idea: Use it anyway!!
How A Broken Idea Can Work

- The idea is to use a hash function that is so unlikely to have a collision that we can ignore that possibility.

- **Cryptographic Hash Functions** have relevant properties.

- Such a function, \( f \), is designed to withstand cryptoanalytic attacks. In particular, it should have
  - **Pre-image resistance**: given \( h = f(m) \), it should be computationally infeasible to find such a message \( m \), given \( h \).
  - **Second pre-image resistance**: given message \( m_1 \), it should be infeasible to find \( m_2 \neq m_1 \) such that \( f(m_1) = f(m_2) \).
  - **Collision resistance**: it should be difficult to find any two messages \( m_1 \neq m_2 \) such that \( f(m_1) = f(m_2) \).

- With these properties, the scheme of using hashes of contents as names is extremely unlikely to fail, even when the system is used maliciously.
SHA1

- Git uses **SHA1** (Secure Hash Function 1).
- Can play around with this using the `hashlib` module in Python3.
- All object names in Git are therefore 160-bit hash codes of contents, in hex.
- E.g. a recent commit in the Spring 2022 shared CS61B repository could be fetched (if needed) with

  ```
git checkout 72c136ba61fb468bf4c8ac906cd57a2c7fa25025
  ```
Low-Level Blob Management

- You can find out the hashcode that Git uses for the blob containing file *something.java* with the command

  ```bash
git hash-object something.java
  ```

- And if this tells you that the file would have hash code

  192a0ca0d159f1550b0b5e102f7e06867cc44782

  and you actually `git add` this file, its compressed contents will be stored (as a blob) in the file

  `.git/objects/19/2a0ca0d159f1550b0b5e102f7e06867cc44782`

  [Why do you suppose the implementors of Git chose this name?]

- You can look at the (uncompressed) contents of the blob with

  ```bash
git cat-file -p 192a0ca0d159f1550b0b5e102f7e06867cc44782
  ```