CS61B Lecture 32

Monday, April 13, 2020

Git Fundamentals

Git is a distributed version-control system that stores snapshots 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 tools to transmit and reconcile versions between repositories. And it is extremely fast as these things go.

Git was first dveeloped by the creator of Linux, Linus Torvalds, who used a proprietary version control system called Bitkeeper in the development of Linux. The initial implementation effort took roughly 2 to 3 months, and was released in 2005.

Initially, it was a collection of basic primitives called "plumbing" that couild be scripted to provide desired functionality. The higher-level commands ("porcelain") was built on top of the plumbing to provide a convenient user interface.

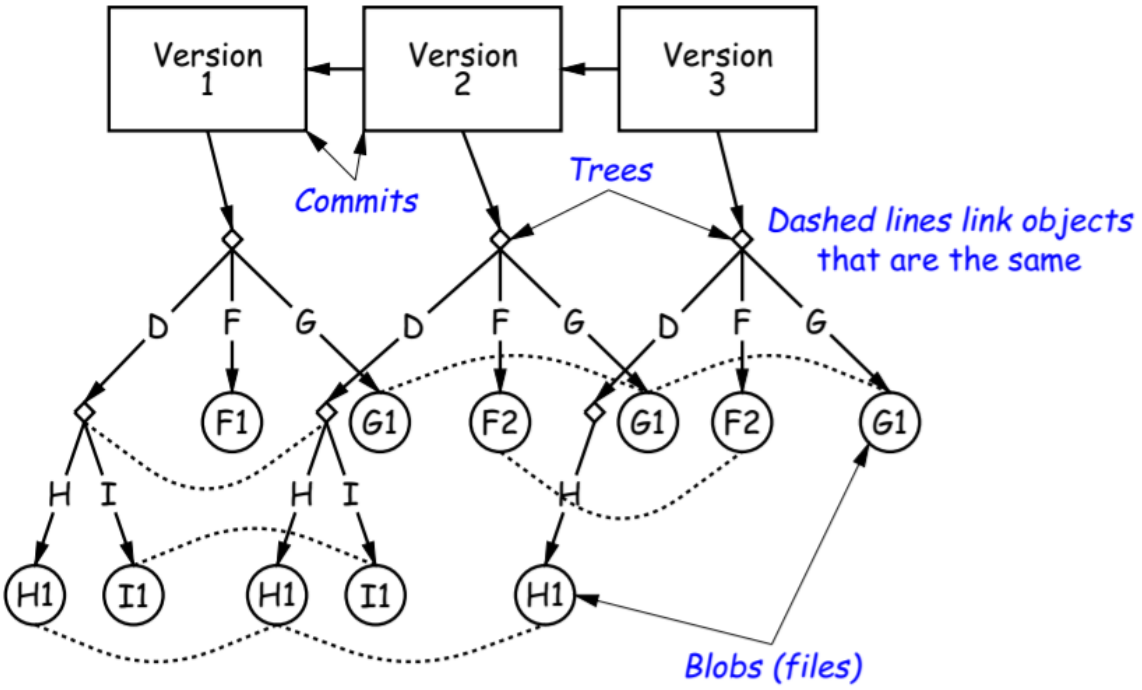
Conceptual Structure

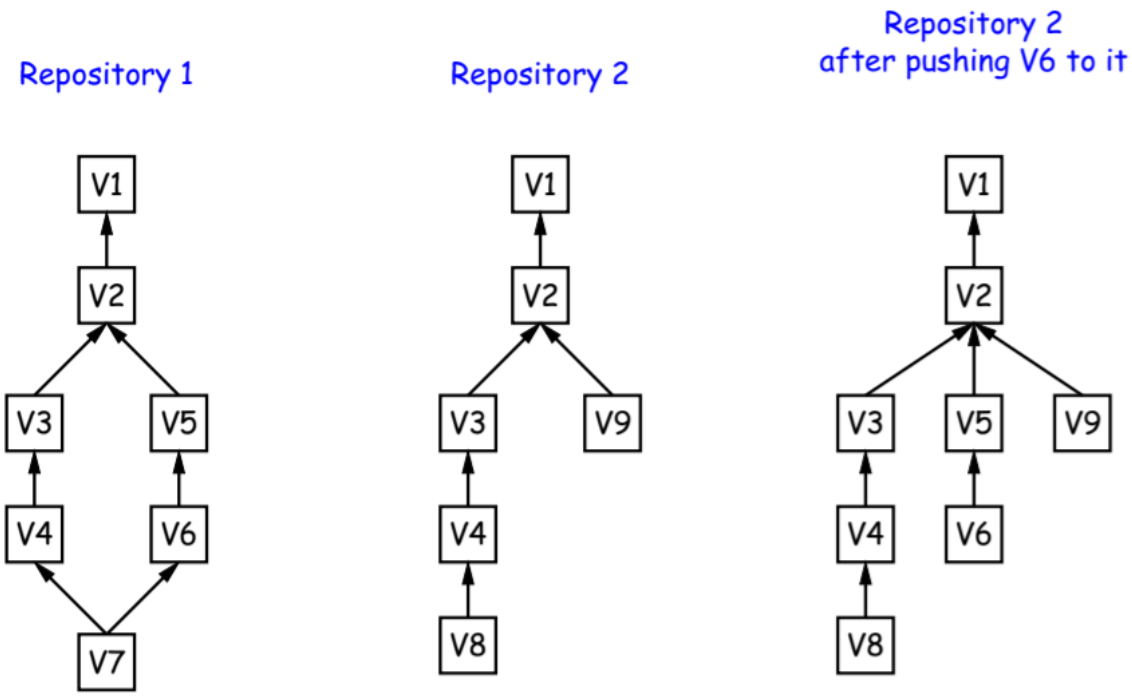
Abstraction is of a graph of versions or snapshots (called commits) of a complete project. The graph structure reflects ancestory: which versions came from which. Each commit contains:

- A directory tree of files
- Information about who committed and when
- A log message
- Pointers to the commit(s) from which the commit was derived.

The main internal components of Git consist of four types of objects:

- Blobs, which hold contents of files
- Trees, directory structures of files
- Commits, contain references to trees and additional information (committer, date, log message)
- Tags, references to commits or other objects, with additional information meant to identify releases and other useful or important information. We will not further discuss tags today.





Each commit has a name that uniquely identifies it to all versions. Repositories can trasmit collections of versions to each other.

Transmitting a commit from repository A to B requires only the transmission of those objects (files or directory trees) that B does **not** yet have (allowing speedy updates).

Repositories maintain named branches, which are simply identifiers of particular commits updated to keep track of the most recent commit in various lines of development. Likewise, tags are essentially named pointers to particular commits. They differ from branches in that they are usually not changed.

Internal Structure

Each Git repository is contained in a directory. The repository may either be bare (just a collection of objects and metadata), or may be included as part of a working directory. The data of the repository is stored in various objects corresponding to files (or other "leaf" content), trees and commits. To save space, data in files is compressed, and Git can garbage collect objects from time to time.

The Pointer Problem

Objects in Git are files. How should we represent pointers between them? We want to be able to transmit objects from one repository to another with different contents, and we only want to transfer objects that are missing in the target repository. How can we transmit the pointers, and how do we know what they are?

We could use a counter in each repository to give each object a unique name, but how can we make this work consistently for two independent repositories.

Content Addressable File System

We could use some way of naming objects that is universal, using the names as pointers. This solves the problem of "which objects don't you have?" in an obvious way. Conceptually, what is invariant about an object, regardless of its repoistory, is its contents. We obviously cannot use the contents as the name, but what we could do is use a hash of the contents as the address. This doesn't really work, because we know from hashing it's close to impossible for a hash function to be completely unique. The brilliant idea is we do it anyway, and pretend the hash code is unique.

How a Broken Idea Can Work

The idea is to use a hash function so unlikely to have a collision we can ignore this possibility. A cryptographic hash function has this relevant property. Such a function `f` is designed to withstand cryptoanalytic attacks, which has these properties:

- **Pre-image resistance**: given

  $$h = f(m)$$

  it should be computationally infeasible to find such a message `m`.
- **Second pre-image resistance**: given message `m1`, it should be infeasible to find

  $$m_2 \neq m_1$$

  such that

  $$f(m_1) = f(m_2)$$
- **Collision resistance**: should be difficult to find

  $$m_2 \neq m_1$$

  such that

  $$f(m_1) = f(m_2)$$

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

SHA-1

Git uses SHA-1 (Secure Hash Function 1). We can play around with this using the `hashlib` module in Python 3. All object names in Git are therfore 160-bit hash codes of contents in hex.

For example, a recent commit in the shared CS61B repository could be fetched with:

```
git checkout e59849201956766218a3ad6ee1c3aab37dfec3fe
```

SHA-1 is an older cryptographic function; it has recently been phased out of security implementations because it has been demonstrated to be able to collide (violating property 3). However, for the purposes of Git naming, it still works fine, and we are still very very unlikely to get the same commit name for two different files.