Lecture #35

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 indepenent 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, propietary VCS (Bitkeeper) withdrew the free version.
- 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, was a collection of basic primitives (now called "plumbing") that could be scripted to provide desired functionality.
- Then, higher-level commands ("porcelain") built on top of these to provide a convenient user interface.

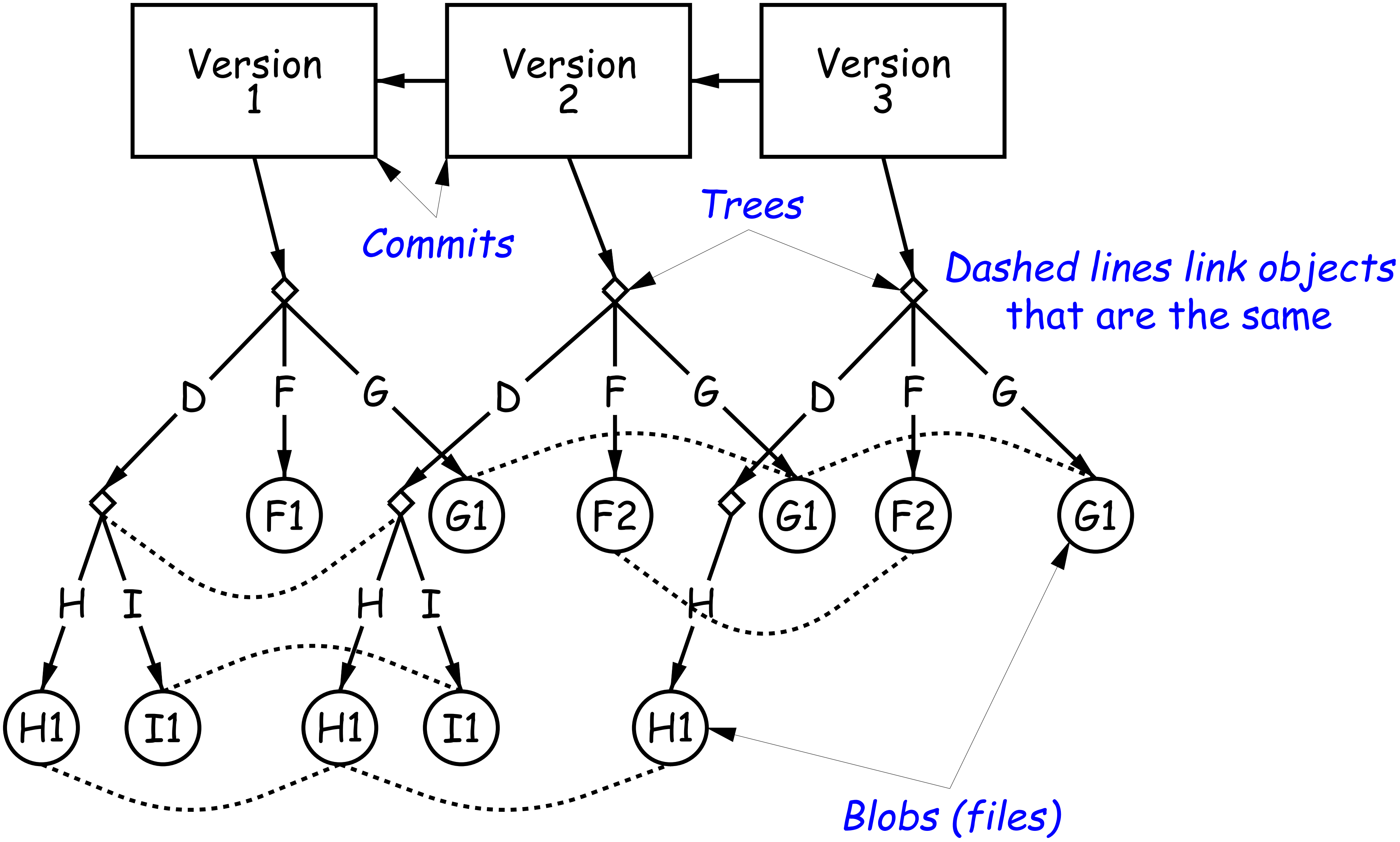
Major User-Level Features (I)

- 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 (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.

Conceptual Structure

- Main internal components consist of four types of *object:*
  - *Blobs:* basically 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, intended to identify releases, other important versions, or various useful information. (Won't mention further today).

Commits, Trees, Files



Commits

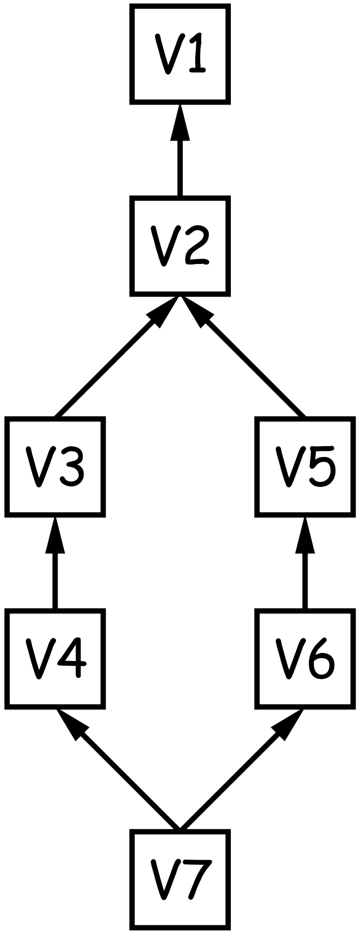
Trees

Dashed lines link objects that are the same

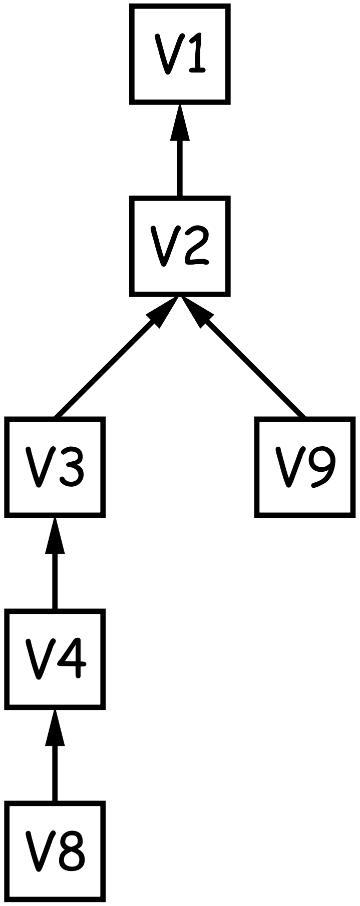
Blobs (files)

Version Histories in Two Repositories

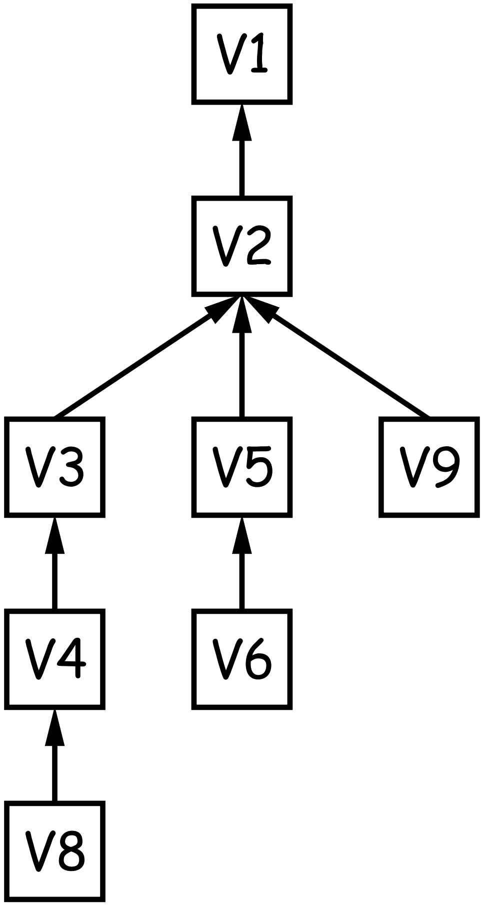
Repository 1



Repository 2



Repository 2
after pushing V6 to it



Major User-Level Features (II)

- Each commit has a name that uniquely identifies it to all versions.
- 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.
- 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*.
- Git can *garbage-collect* the objects from time to time to save additional space.

The Pointer Problem

- Objects in Git are files. How should we represent pointers between them?
- Want to be able to *transmit* objects from one repository to another with different contents. How do you transmit the pointers?
- Only want to transfer those objects that are missing in the target repository. How do we know which those are?
- Could use a counter in each repository to give each object there a unique 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 an object, regardless of repository, is its *contents*.
- But can't use the contents as the name for obvious reasons.
- 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 property.
- Such a function, f, is designed to withstand cryptoanalytic attacks. In particular, should have
  - *Pre-image resistance:* given $h = f(m)$, should be computationally infeasible to find such a message m.
  - *Second pre-image resistance:* given message m_1, 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 *any* two messages $m_1 \neq m_2$ such that $f(m_1) = f(m_2)$.
- With these properties, scheme of using hash of contents as name is extremely unlikely to fail, even when 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 shared CS61B repository could be fetched (if needed) with

```
git checkout e5984920195676218a3ad6ee1c3aab37dfec3fe
```