# Git's Content Addressable Storage

COS 316: Principles of Computer System Design

***Amit Levy*** & Ravi Netravali

# Last time: UNIX File System Layers

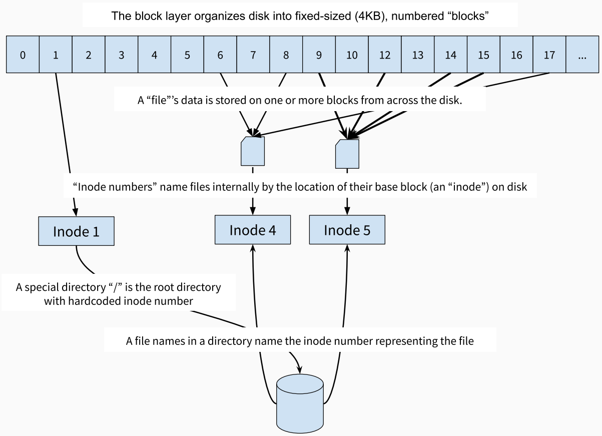



**Figure 1:** UNIX File System Layers

# UNIX File System Layers

An example of “location-based” naming schemes:

# UNIX File System Layers

An example of “location-based” naming schemes:

- The block layer names blocks based on the order in which they appear on disk

# UNIX File System Layers

An example of “location-based” naming schemes:

- 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

# UNIX File System Layers

An example of “location-based” naming schemes:

- 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 inode number layer gives files names that correspond to their block number or *location* within an inode table

# UNIX File System Layers

An example of “location-based” naming schemes:

- 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 inode number layer gives files names that correspond to their block number or *location* within an inode table
- The absolute path name layer provides the *location* of the root directory

# UNIX File System Layers

An example of “location-based” naming schemes:

- 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 inode number layer gives files names that correspond to their block number or *location* within an inode table
- The absolute path name layer provides the *location* of the root directory

# UNIX File System Layers

An example of "location-based" naming schemes:

- 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 inode number layer gives files names that correspond to their block number or *location* within an inode table
- The absolute path name layer provides the *location* of the root directory

***Location-based naming scheme***

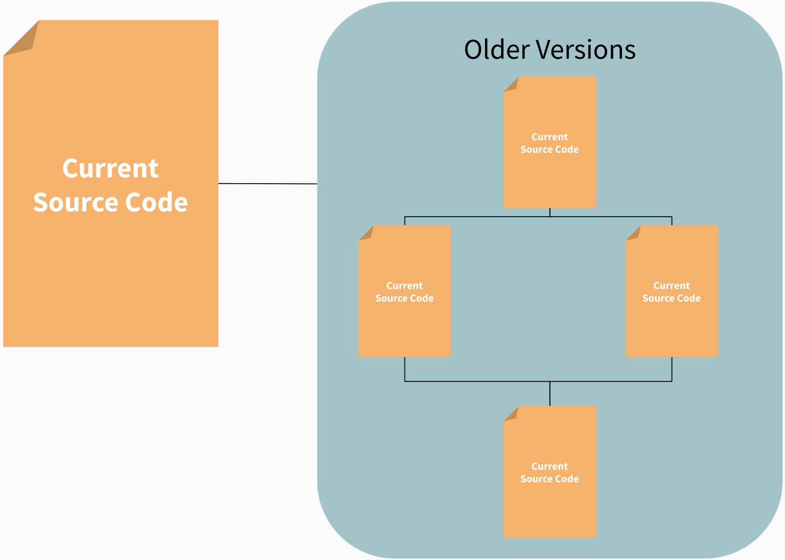
# Today: When do locations fall short

- UNIX File System takes a location-centric view of the data it stores
  - Point is: where on disk can I find this data I care about?
- When might this view be insufficient?

# Today: When do locations fall short

- UNIX File System takes a location-centric view of the data it stores
    - Point is: where on disk can I find this data I care about?
- When might this view be insufficient?
- Today: Git as a lens for:
    - How location-based names fall short
    - How content-based names can help

# Version Control Overview



# A Brief History of Version Control

**Local version control**

- 1972: Source Code Control System (SCCS) developed by early UNIX developers
- 1982: Revision Control System (RCS) developed by GNU project

**Client/Server Centralized Version Control**

- 1986: Concurrent Versions System (CVS) developed as front-end to RCS to collaborate on Amsterdam Compiler Kit at Vrije University
- 2000: Subversion (SVN) a redesign of CVS widely used by open source projects

**Distributed Version Control**

- 2000: BitKeeper developed to address Linux's distributed and large community development model
- 2005: Git & Mercurial developed concurrently to replace BitKeeper after BitMover starts charging open source projects.

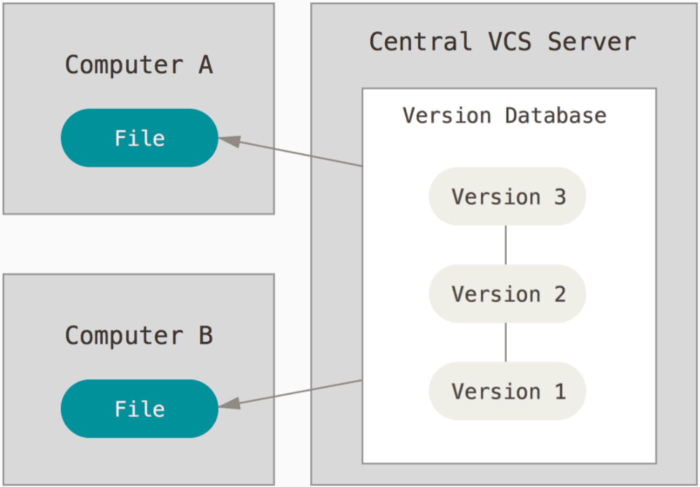



**Figure 2:** Centralized Version Control

# Centralized Version Control

- Central server holds "canonical" version of each file
- Files committed and versioned independently
- Typically only one or a few checkouts of a file
- Conflicts between developers expected to be rare
- All versioning and conflict resolution mediated by the server

# Centralized Version Control

- Central server holds "canonical" version of each file
- Files committed and versioned independently
- Typically only one or a few checkouts of a file
- Conflicts between developers expected to be rare
- All versioning and conflict resolution mediated by the server

Main role: efficiently store versions of the same file and coordinate updates to individual files.

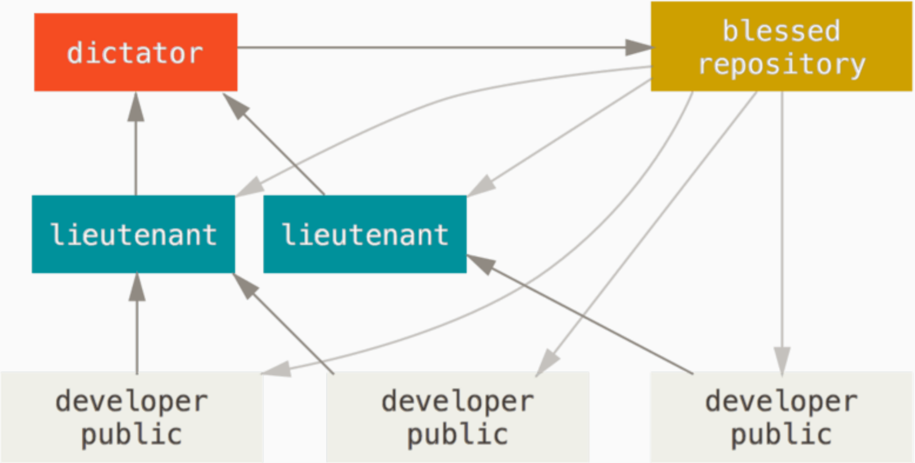
# Centralized Version Control

- Central server holds “canonical” version of each file
- Files committed and versioned independently
- Typically only one or a few checkouts of a file
- Conflicts between developers expected to be rare
- All versioning and conflict resolution mediated by the server

Main role: efficiently store versions of the same file and coordinate updates to individual files.

UNIX file system is a pretty good match!

# Linux development model



# Centralized Version Control Shortcomings...

- Are the set of files in the canonical version collectively valid?
- Not egalitarian: What if we don't want just one "central" server?
  - P2P collaboration, hierarchical, etc...
- What happens if the data on the central server is corrupted?

# Distributed Version Control

Two important differences from centralized:

1. No inherent "canonical" version
2. Unit of a commit is a complete source code tree
   - Each "version" represents a state that *some* developer intended at *some* time
   - Versioning *files* is incidental

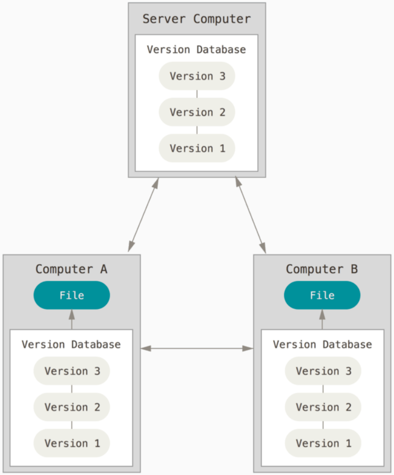
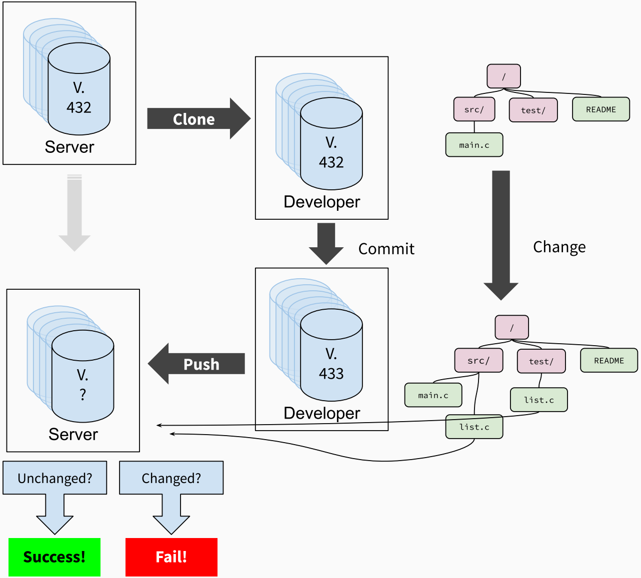



**Figure 3:** Distributed Version Control

# Distributed Version Control Workflow Example



**How would we do this with the UNIX file system?**

**How would we do this with the UNIX file system?**

We need a simple way to succinctly *name* files, trees, commits, etc such that we can easily compare them.

We need to efficiently store and transmit many versions of source code tree. Most files in each version will be unchanged.

# The Content-based Address

- A succinct summary of the content

# The Content-based Address

- A succinct summary of the content
- that's unique for different content

# The Content-based Address

- A succinct summary of the content
- that's unique for different content
- and the same for the same content

# The Content-based Address

- A succinct summary of the content
- that's unique for different content
- and the same for the same content

# The Content-based Address

- A succinct summary of the content
- that's unique for different content
- and the same for the same content

Cryptographic hash functions maps arbitrary size data to a fixed-sized bit-string that is:

- Deterministic
- Computationally "hard" to generate a message that yields a *specific* hash value
- Computationally "hard" to find two messages with the same hash value
- *Similar messages have dissimilar hashes*

# Git Internals

# Git Layers

| Layer | Purpose |
|---|---|
| *Object layer* | Stores objects in a content-addressable store |
| *Tree layer* | Organizes “blobs” into a directory-like hierarchy |
| *Commit layer* | Versions the tree layer |
| *Reference layer* | Provides human-readable names for trees, blobs, commits |

Similar to UNIX file system layers, but uses content-based names instead of location-based names.

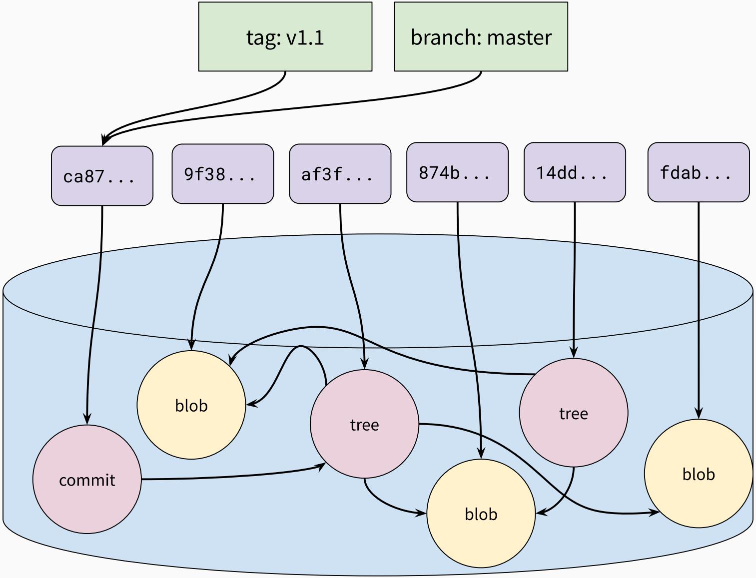



**Figure 5:** Git's Layers

# Object Layer

"Objects" are the basic storage unit in Git, similar to blocks in the UNIX file system.
*All data is stored as objects.*

**Names**

- The SHA-1 hash of the object's content: 40-byte string in hex (160-bits)
- `aa8074278ed2c4803a2a545f277d1e0afe5039c3`

**Values**

- *Blobs*: similar to files
- *Trees*: similar to directories
- *Commits*: points to tree and previous commit

# Object Layer

**Allocation**

- Names “allocated” by taking the hash of the object content

**Lookup**

- Git uses the UNIX file system to store objects on disk
  - We need to translate to locations at *some* point
- Objects stored in a directory `.git/objects`.
- Filename is the 40-byte hex string of the object’s name

# Tree Layer

Similar to, and modeled after, directories in the UNIX file system:

Provide hierarchy of trees and blobs that can be traversed using human-meaningful names.

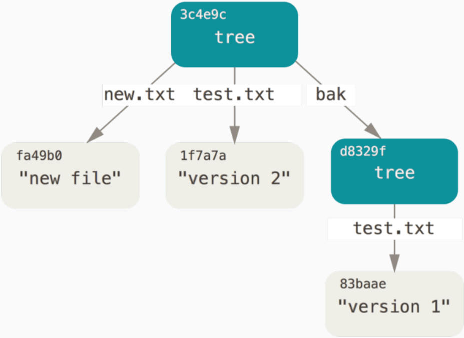



**Figure 6:** Git tree objects

# Tree Layer

**Names**

- Human-readable strings, just like in UNIX directories

**Values**

- Object name
- Object type
- Permissions (a subset of UNIX permissions)

**Allocation**

- Names are supplied by the user, just like in UNIX
- Generally, git mirrors an actual directory structure

# Tree Layer

### Lookup

- Trees stored as a list of entries, similar to directories

```
$ git cat-file -p 3914fbcc30ea8092034ca5ea4e6ebd0c887495df
100644 blob 96e87117fc618fc54a770bfc938405a29cca1fbb    .gitignore
100644 blob 077b93358fba58cacc6acaf098baa317408aa16e    Makefile
100644 blob 7addb405782f208c54f6d31182e173304ee117b9    README.md
040000 tree 303c20a830ce296d625fbf0fe4e4cd99fc33f3b1    http_router
040000 tree 85c17ff71ae5cfafcb1affebc4fbc1e8e67bd23c    microblog-client
040000 tree a7dc7cfb0850fbfd4fcdf49310fd2e757cb42c08    microblog-server
```

# Commit Layer

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

- 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

# Commit Layer

**Names**

- "Tree"
- "Parent"
- "Author"
- "Committer"
- "Commit message"

**Values**

- Object name of the tree
- Object name of the parent commit(s)
- Author/committer name and e-mail, and date committed
- Message as a string

# Commit Layer

**Allocation**

- Names don't need to be allocated because they are pre-determined

**Lookup**

- Commit objects have a defined format such that each name has a particular location in the object

## Reference Layer

Commits, trees, and blobs names not convenient for humans.

- Can't remember hashes
- Not useful for discovery
- *Need some point of synchronization*
  - e.g., how do we know which is the most recent commit?

References provide global, human readable names for objects

# Reference Layer

**Names**

- Human readable names: e.g. “master”, “alevy/wip”, “HEAD”, etc

**Values**

- A commit name

# Reference Layer

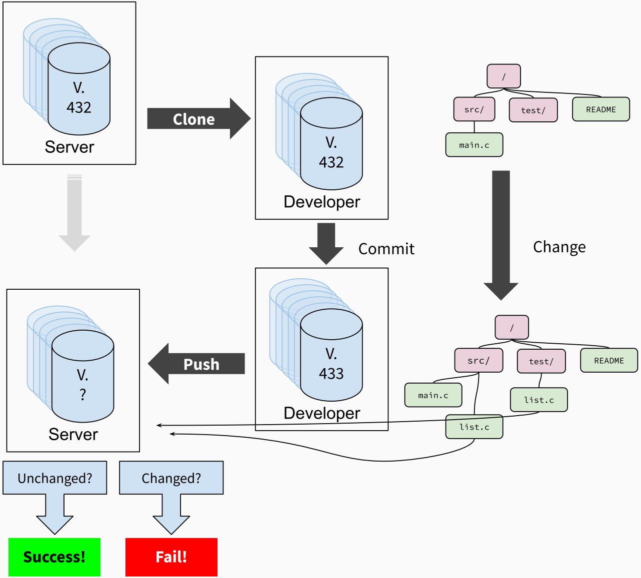
**Allocation**

- Reference names are assigned and managed by users
- Some standard reference names by convention:
  - master: 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

**Lookup**

- Stored as UNIX files in a special subdirectory of the .git directory
- Each reference is a file containing the name of the object they refer to

# Distributed Version Control Workflow



# Contrasting Location-based names & Content-based names

# Layers of names

Both systems we looked at use layers of simple naming schemes.

- Makes reasoning easier
- UNIX File System
  - Blocks, files, inode numbers, directories, absolute path
- Git
  - Objects, blobs, trees, references
- Allow extensibility at multiple levels
  - Can re-use block layer for other storage systems, e.g. databases
- Allows portability at multiple levels
  - Can port files & directories to non-block storage

# Economy of mechanism

Both systems we looked at reuse mechanisms where possible

- UNIX file system
  - Stores everything in blocks: inodes, file data, file system metadata
  - Reuses inodes for files and directories
- Git
  - Stores everything in objects: blobs, trees, commits
  - Single naming allocation scheme: secure hash function

# Naming design trade-offs

| | **Location-based names** | **Content-based names** |
|---|---|---|
| *Necessary* | Yes! | Nope |
| *Discovery* | Easy | Hard |
| *Decentralized* | No | Yes |
| *Integrity* | Hard | Easy |
| *Transactions* | Hard | Easy |

# Up Next

- Naming in Networking
- Assignment 1 due next Wednesday