Weakly Consistent and Disconnected Operation

Linearizability Recap

Everyone sees same order of read/write operations

- cache coherence, Paxos

Release consistency/fsync:

- at memory barriers/lock/unlock, wait for all reads/writes to complete

Need a different model for always available writes

- Disconnected operation (Bayou, git)
- Massive scale (DNS)
- Low latency even during failures (Dynamo)

Why Disconnected Operation?

Apps that work offline/intermittent connectivity

- Most productivity apps: gmail, google docs, etc.
- Data updated locally, merged later

File synchronization across users / devices

- Dropbox: data updated continuously

Source code control (cvs, git)

- Update data locally, explicit merges

- Writes can conflict, merge later

Two Models for Disconnected Apps

- Applications only communicate with the cloud (Coda, SVN)
  - Log changes, apply on reconnect
- Applications can communicate with cloud and each other (Bayou, git)
  - Log changes, replicas exchange logs and merge
  - Merge again when connect to new replica

Coda

- File system that supports disconnected operation of laptops, PDAs
  - Local file system partial replica of global one
  - System tried to pre-cache everything you might need
- While disconnected, log every modification
  - Like a write ahead log
- Merge on reconnection
  - Reconnection applied atomically

Coda Merge

- On reconnect, merge by applying changes from client log
  - Bring client up to date by applying server log
- If no one else has modified data in the meantime
  - Ex: clients working in different directories
  - Apply changes from log in log order
- What if two clients modify the same data?
  - Apply changes that don’t conflict
  - Flag changes that require manual intervention

Application-specific Merge

- What happens if two disconnected nodes make conflicting updates?
- Detect when merging changes back onto server
- For each change (in Coda)
  - If to different files, ok
  - If create/delete/rename, ok if to different files
  - If changes to same file, app-specific merge
- Merging easier if operational log at app-level
  - Versus logging data structures with changes in them

Bayou

Xerox PARC project to build the first practical PDAs

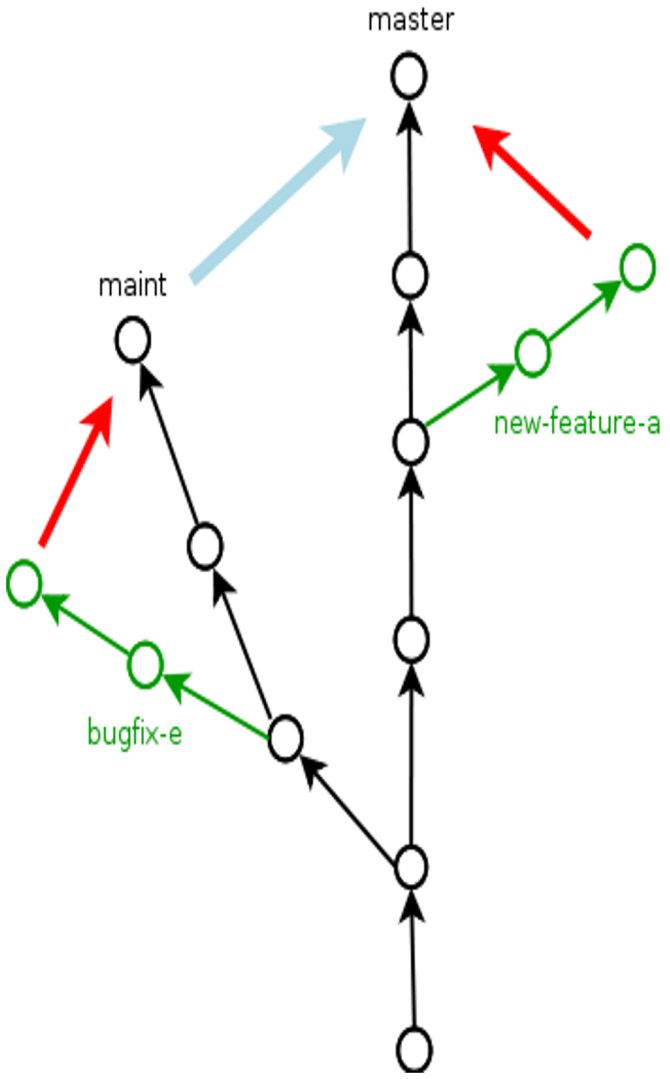
Collaborative apps with partial and limited connectivity

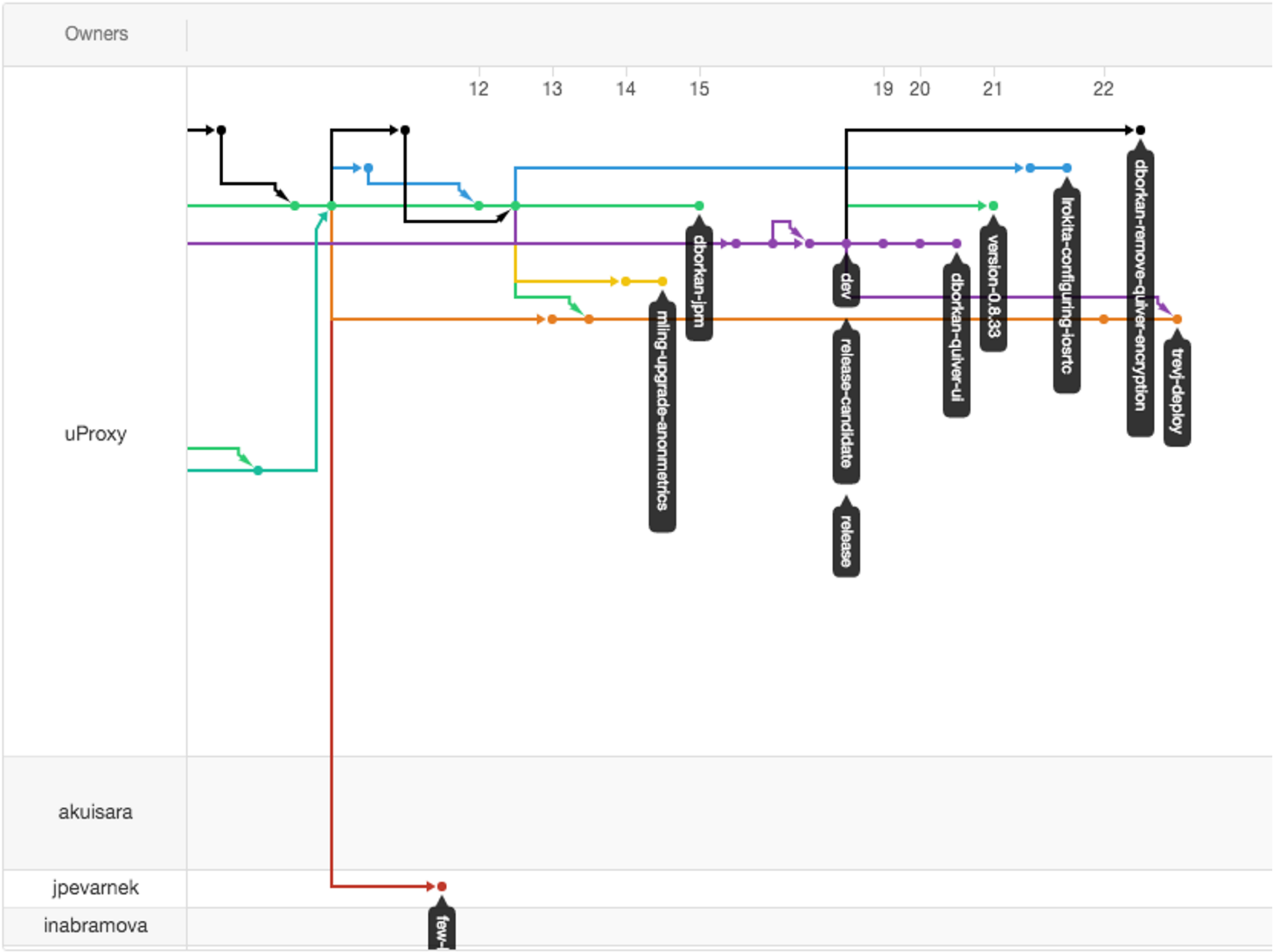
- Sometimes no connectivity
- Sometimes only peer-to-peer connectivity
- Sometimes peer-to-server connectivity

Forced to address the general problem

Source Code Control

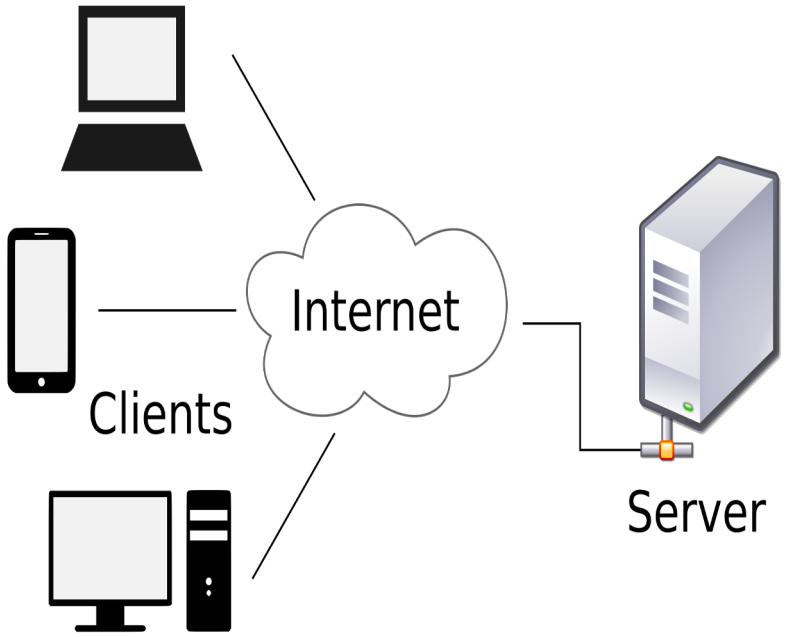
- Eventual Consistency
  - Read/write local copy
  - Fix conflicts later
- Track history (with metadata)
- Concurrent editing / Many contributors
- Working copy: files don’t change beneath you
  - Push / Pull to server/peers
  - Contributors may be offline / disconnected

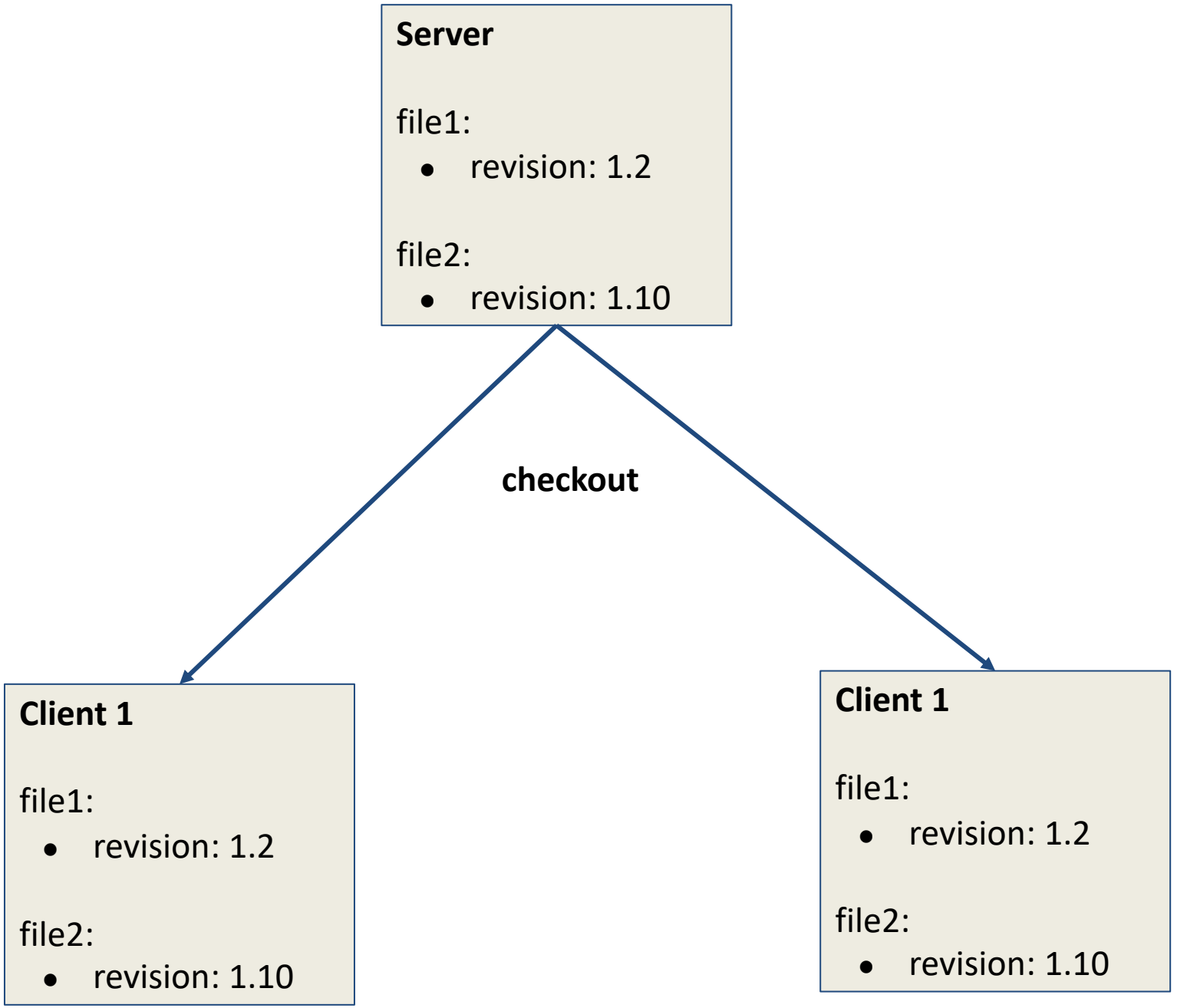


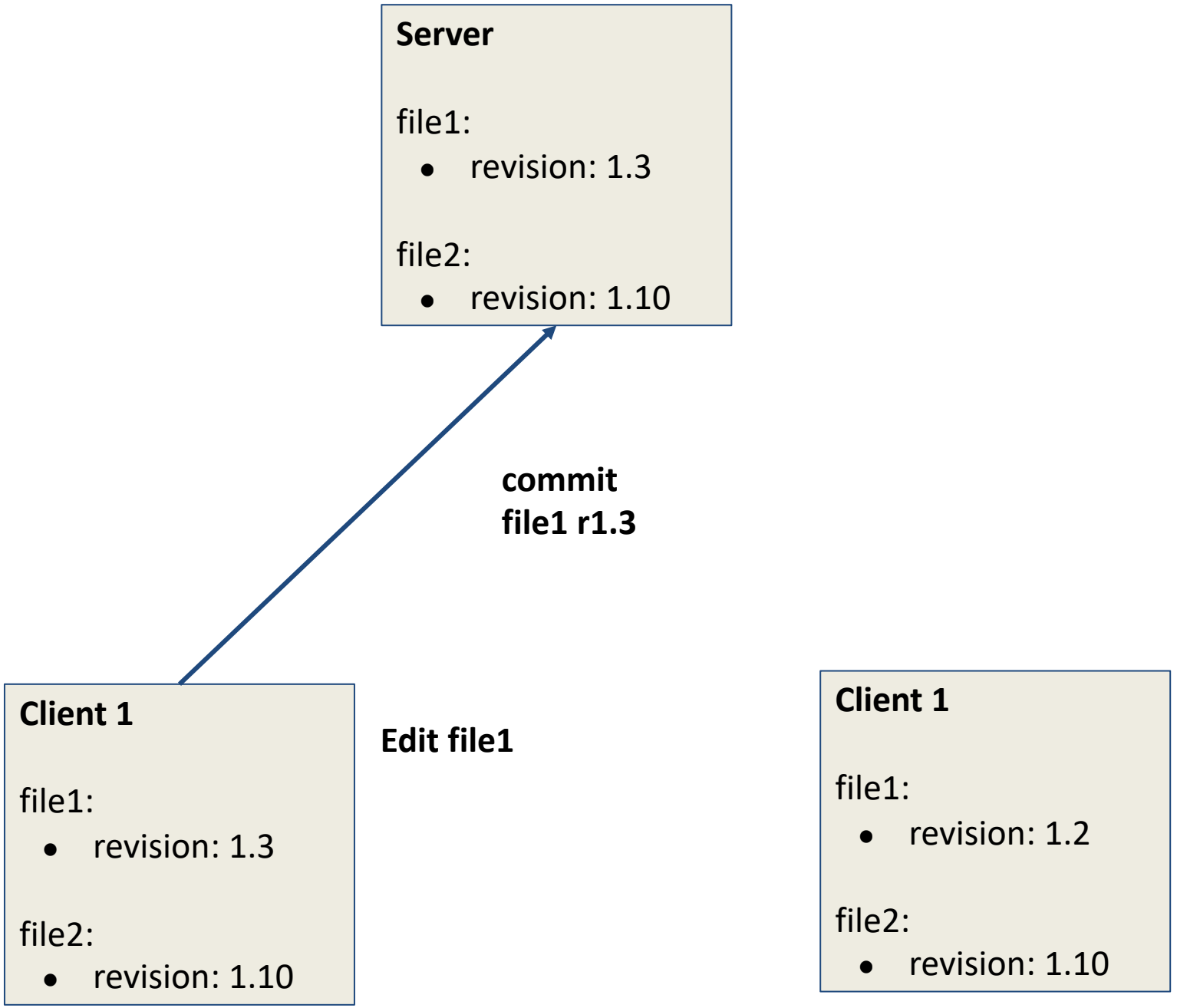

Owners
12
13
14
15
19
20
21
22
uProxy
akuisara
jpevarnek
inabramova
dborkan-jpm
mling-upgrade-anonmetrics
dev
release-candidate
release
dborkan-quiver-ui
version-0.8.33
lrokita-configuring-iosrtc
dborkan-remove-quiver-encryption
trevj-deploy

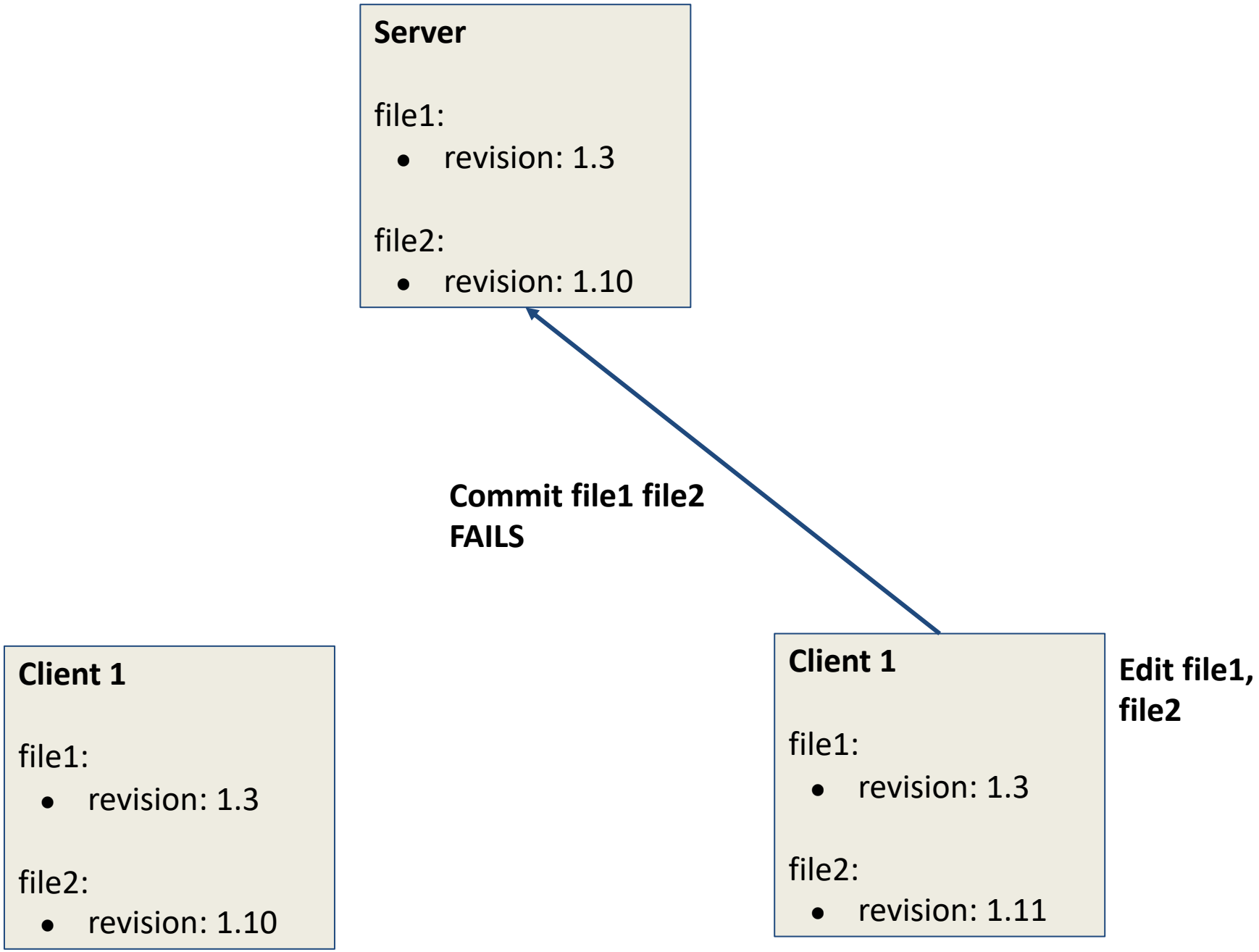
CVS (1990)

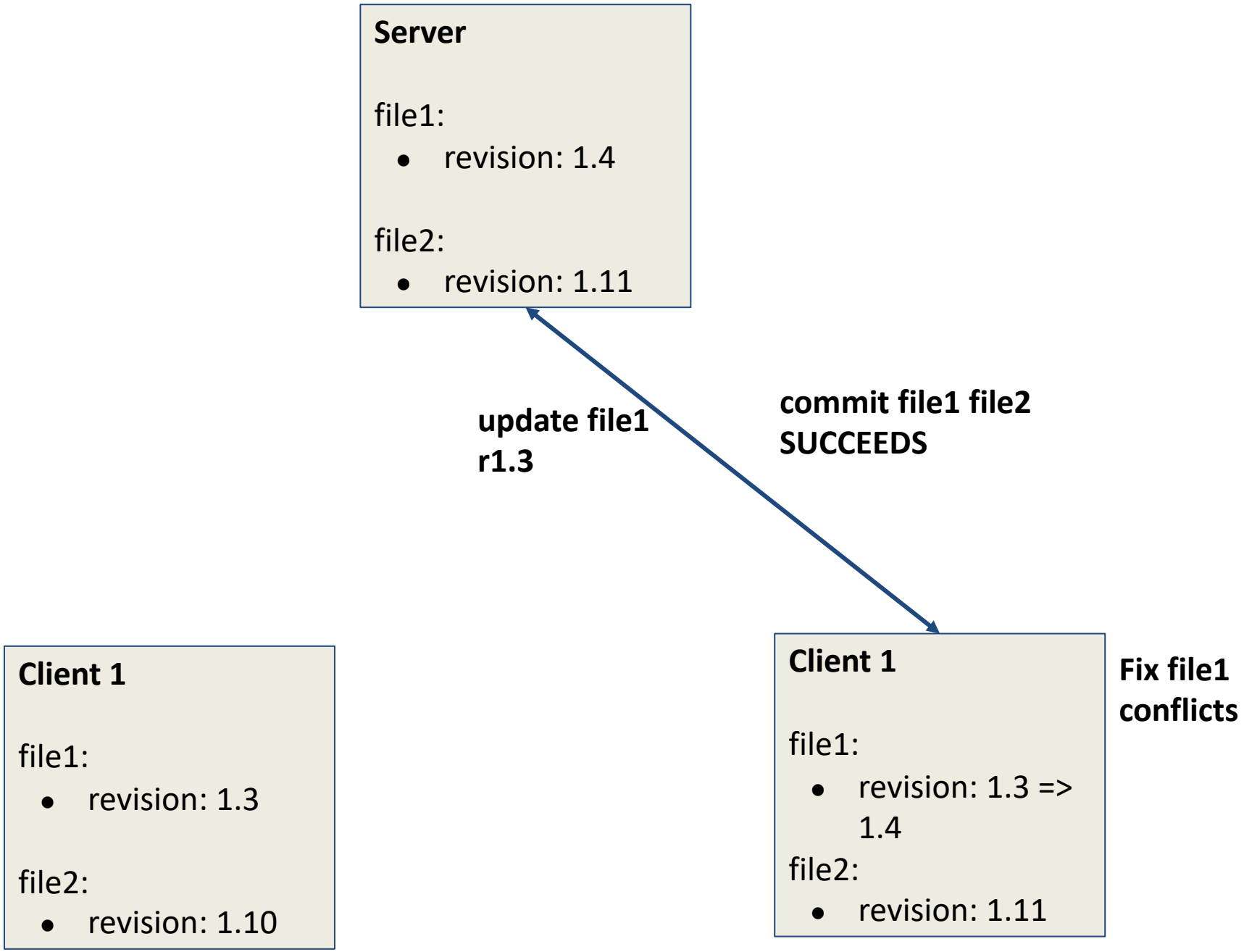
- Client-server model
  - Check out working copy
  - Check in your changes
- Server arbitrates order
  - Only accept changes to the most recent version
  - Developers must always keep their files up to date




Server
file1:
revision: 1.2
file2:
revision: 1.10
checkout
Client 1
file1:
revision: 1.2
file2:
revision: 1.10
Client 1
file1:
revision: 1.2
file2:
revision: 1.10


Server
file1:
• revision: 1.3
file2:
• revision: 1.10
commit
file1 r1.3
Client 1
file1:
• revision: 1.3
file2:
• revision: 1.10
Edit file1
Client 1
file1:
• revision: 1.2
file2:
• revision: 1.10


Server
file1:
revision: 1.3
file2:
revision: 1.10
Commit file1 file2
FAILS
Client 1
file1:
revision: 1.3
file2:
revision: 1.10
Client 1
file1:
revision: 1.3
file2:
revision: 1.11
Edit file1,
file2


Server
file1:
revision: 1.4
file2:
revision: 1.11
update file1
r1.3
commit file1 file2
SUCCEEDS
Client 1
file1:
revision: 1.3
file2:
revision: 1.10
Client 1
file1:
revision: 1.3 =>
1.4
file2:
revision: 1.11
Fix file1
conflicts

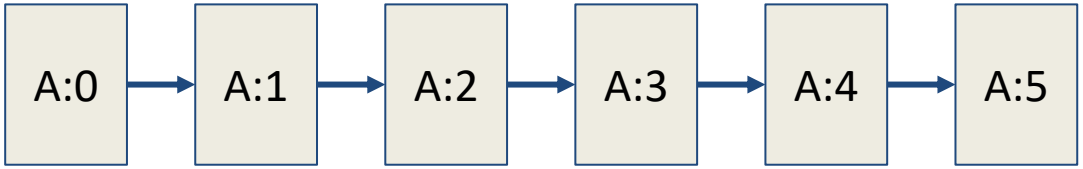
CVS Limitations

- Everyone edits the same repository
  - How does a subgroup implement a complex feature?
- No local version control
  - cvs commit ~ git commit && git push
- No log/ time travel
- No versioning of moving / renaming files
- Depends on live server to operate
  - Scaled / backed up / reachable
- Branches were expensive
- Updates not atomic (!)

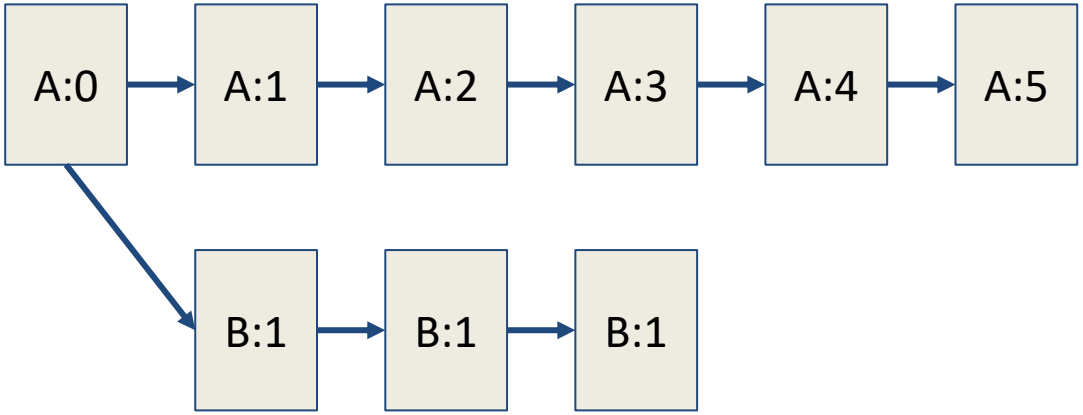
Apache SVN (2000)

- Improvements
  - Atomic commits
  - Renamed / moved / copied files retain version history
  - Versioning of directories and metadata
  - Cheap branches / tagging
- Centralized - server/client architecture
- Still active
  - All of Facebook’s source code was in a single SVN repository until 2014

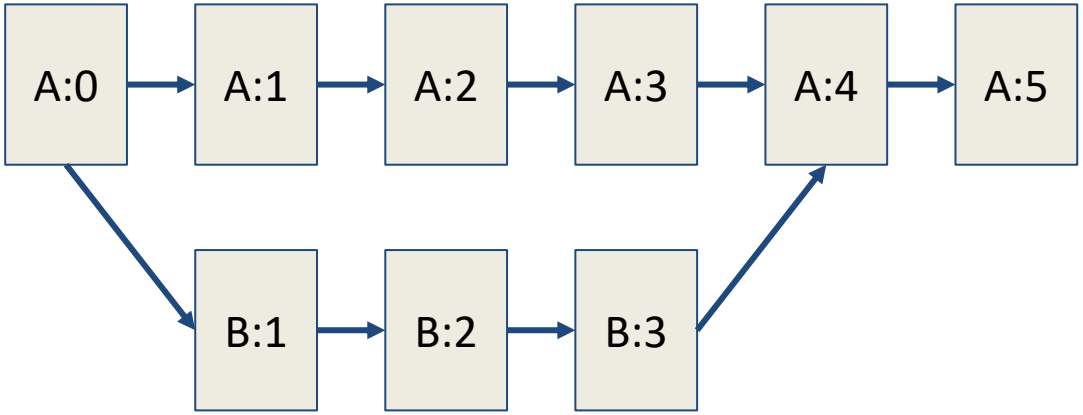
Commit Log



Branching



Merging



A:5 Ancestry Set
{A: 0-4, B:1-3}

Conflicting updates detected with vector clocks
What then?

Merge Conflicts

Easy: create/delete/rename different files in directory => union of changes

Medium: changes to different lines of text file => diff+patch
Change to file that has been renamed => apply

Hard: changes to the same line of C source => ask user to fix

Another option: operational transforms

Merging and Causal Ordering

Operations that potentially are causally related are seen by every node of the system in the same order

Example:

C1: f=1 -> C2

C2: f=2 -> C3

C3: f=3 -> C1

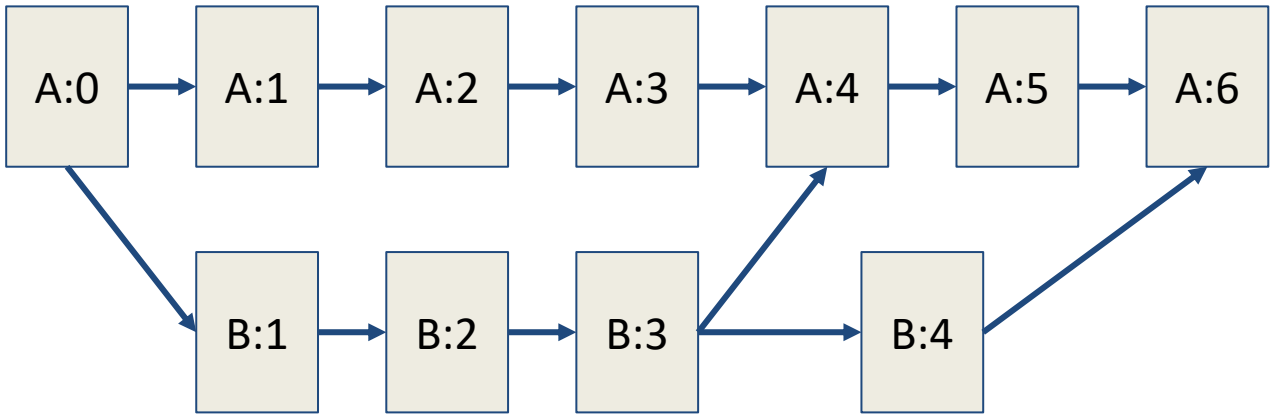
Example:

C1: a=1 -> C2

C2: b=2 -> C3

C3: c=3 -> C1

Merging



A:6 Ancestry Set
{A: 0-5, B:1-4}

A:5 Ancestry Set
{A: 0-4, B: 1-3}

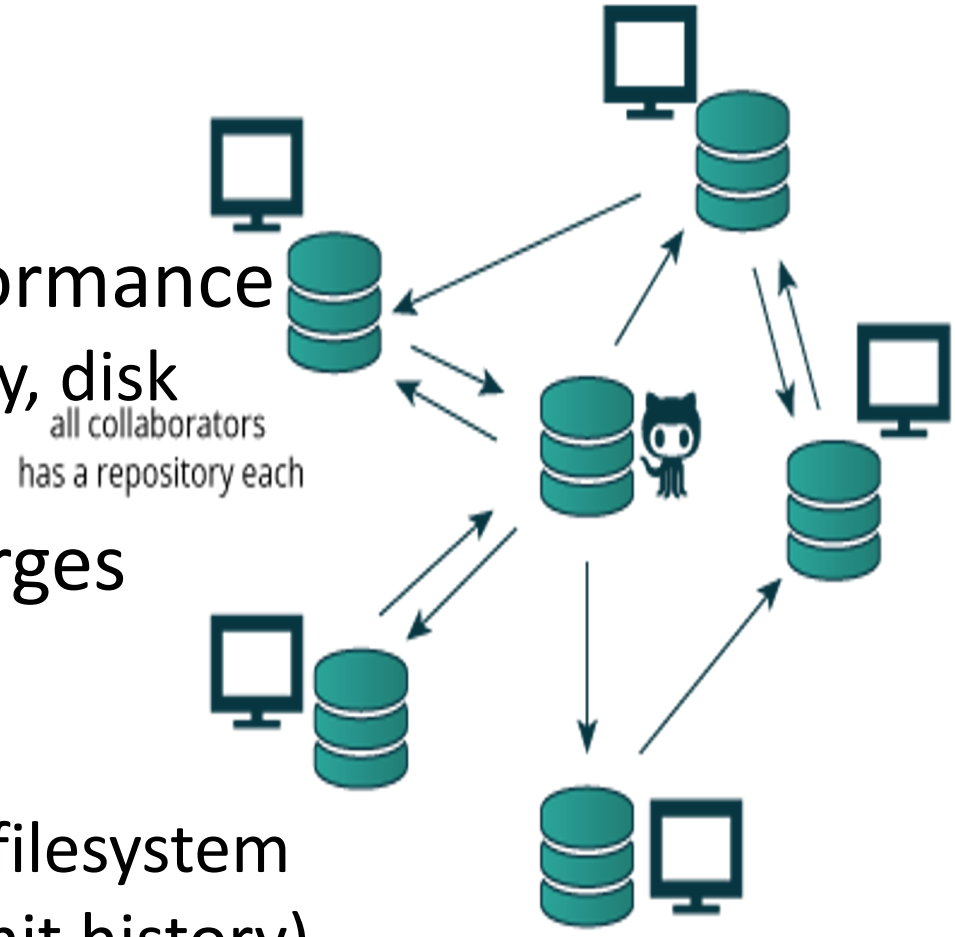
B:4 Ancestry Set
{B: 1-3}

Garbage Collection

- When is it safe to garbage collect the log of changes?

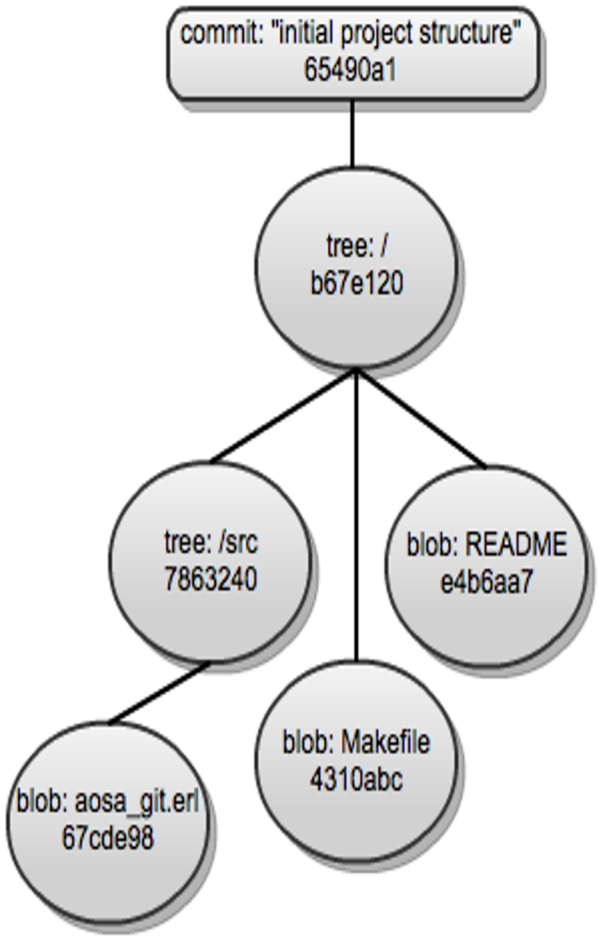
git (2005)

- Distributed!
  - Everyone is a replica
- Consistency and performance
  - Protects from memory, disk corruption
- Cheap branches / merges
- .git/
  - Config
  - Content-addressable filesystem
  - Log of changes (commit history)



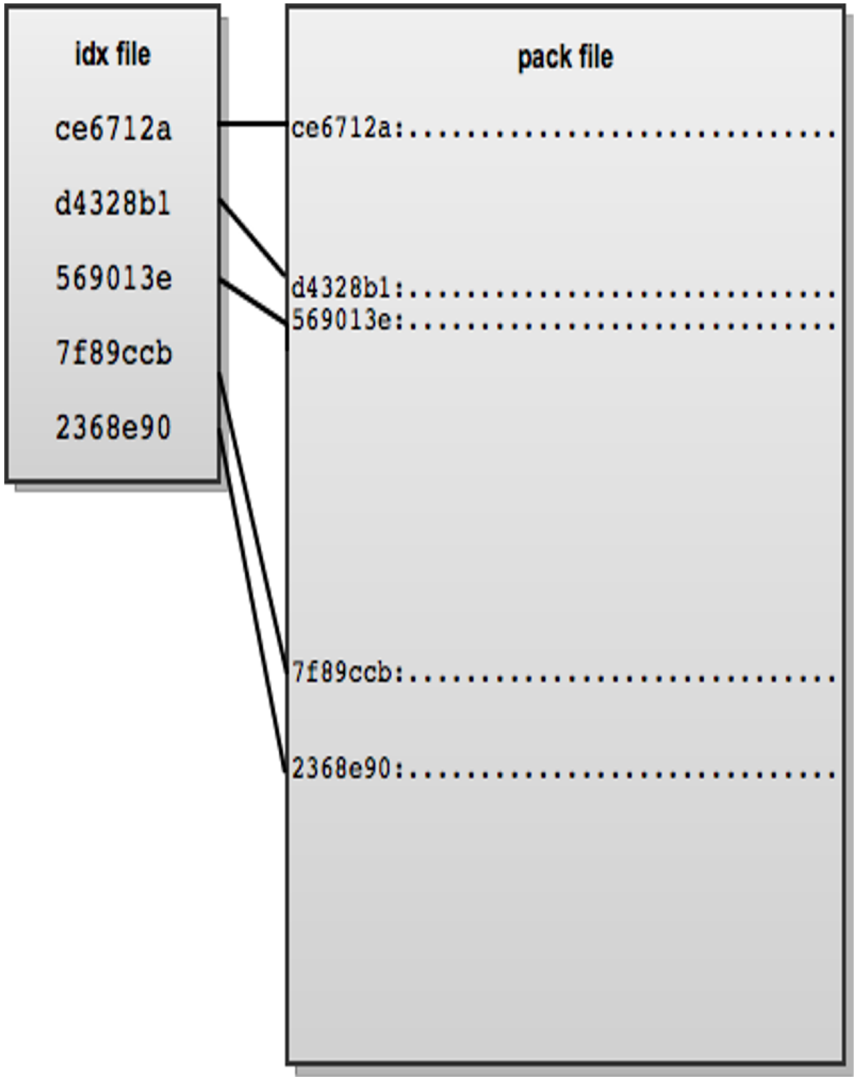
Logs (Commit Histories)

- Complete log of changes (needed for time travel with source code control)
  - Directed acyclic graphs (DAG)
- commit
  - parents
  - deltas (changes to content)
  - hash - for consistency
  - metadata



Content Addressable Filesystem

.git/objects



Git Example

```
$ git init
$ echo “version 1” > test.txt
$ git add test.txt
$ git commit -m “first commit”

$ echo “version 2” > test.txt
$ echo “new file” > new.txt
$ git add ./
$ git commit -m “second commit”

$ mkdir bak
$ echo “version 1” > bak/test.txt
$ git add bak/
$ git commit -m “third commit”
```