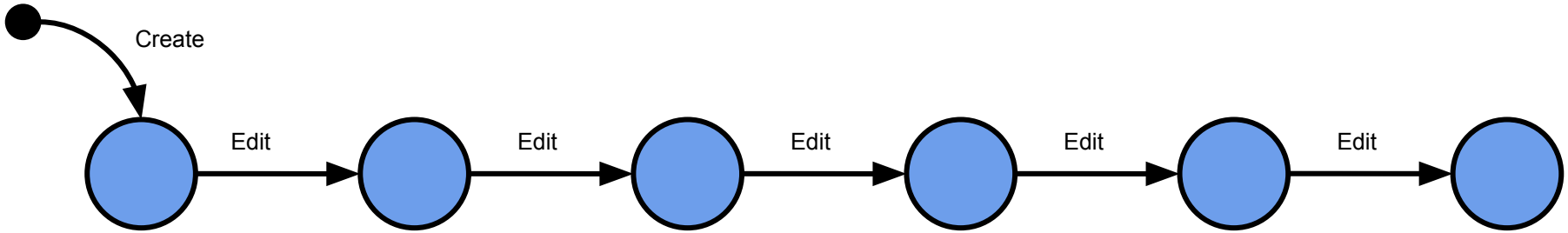
Collaborating on Projects with Version Control

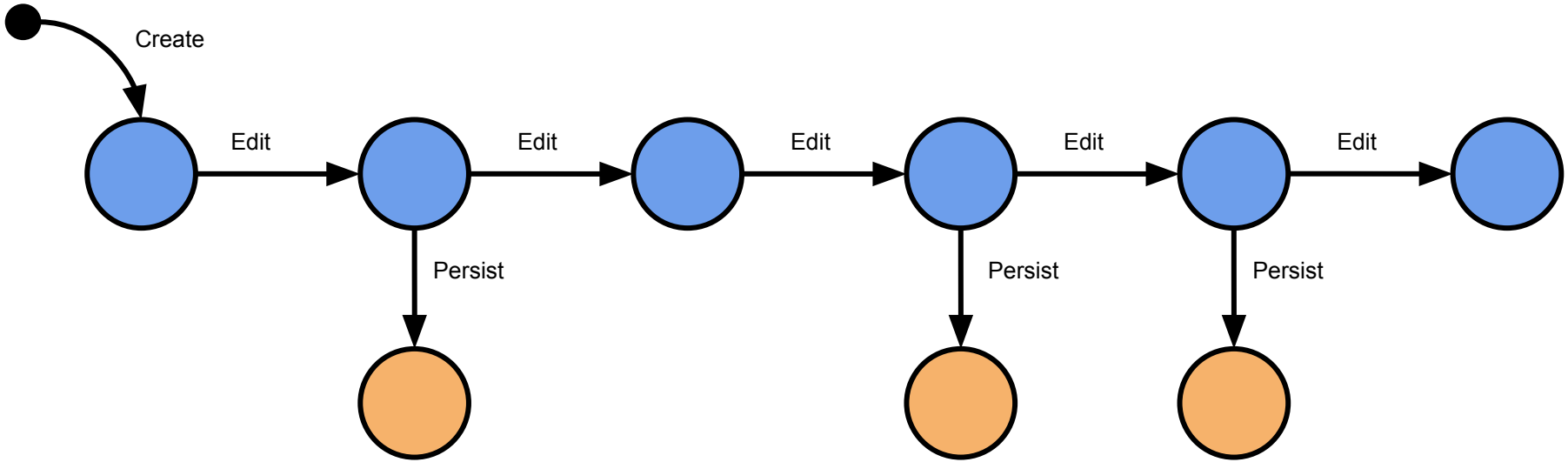
Outline

- Artifacts’ lifecycle
- Managing revisions
- Client/server vs distributed revision systems
- Managing conflicts
- Branching
- Workflows

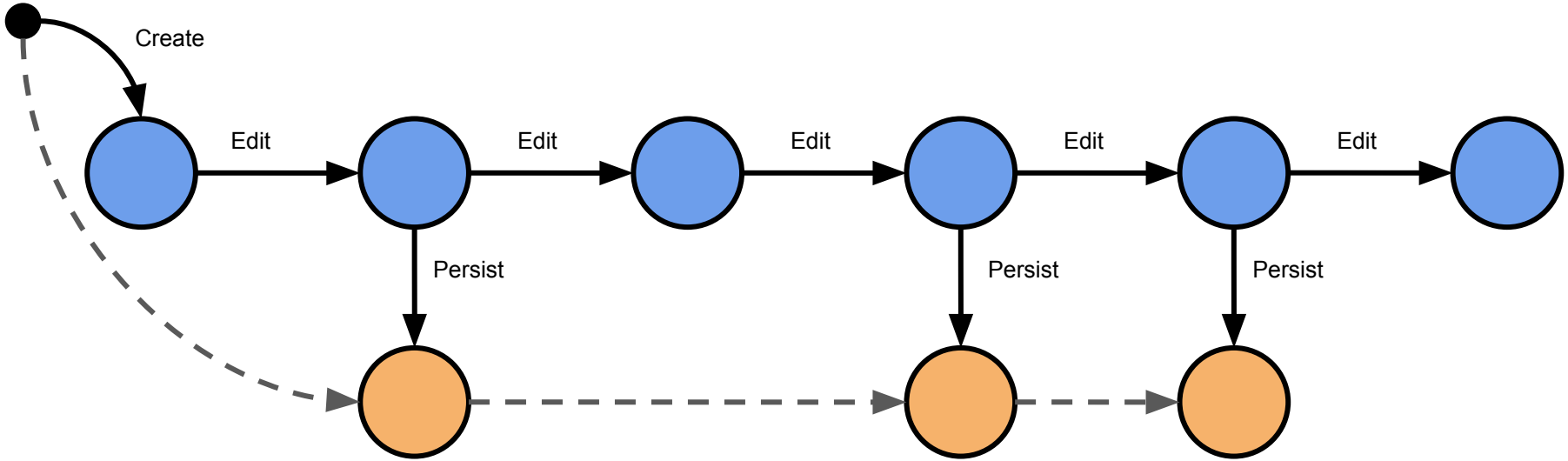
Artifacts’ lifecycle



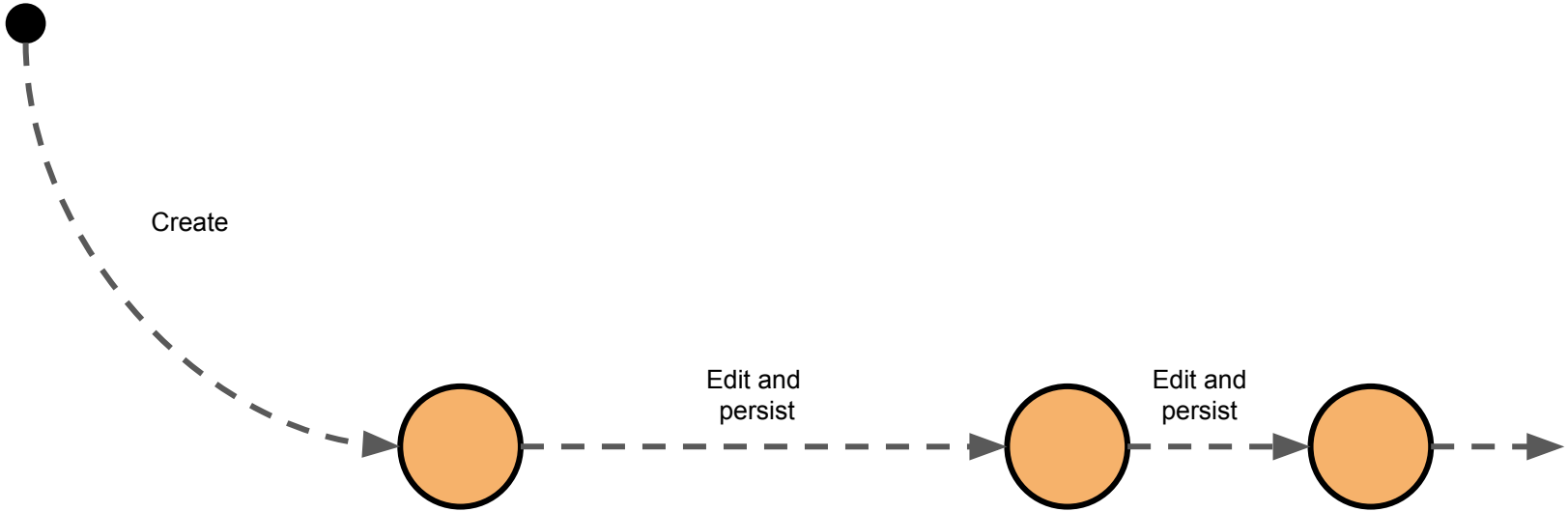
Artifacts’ lifecycle



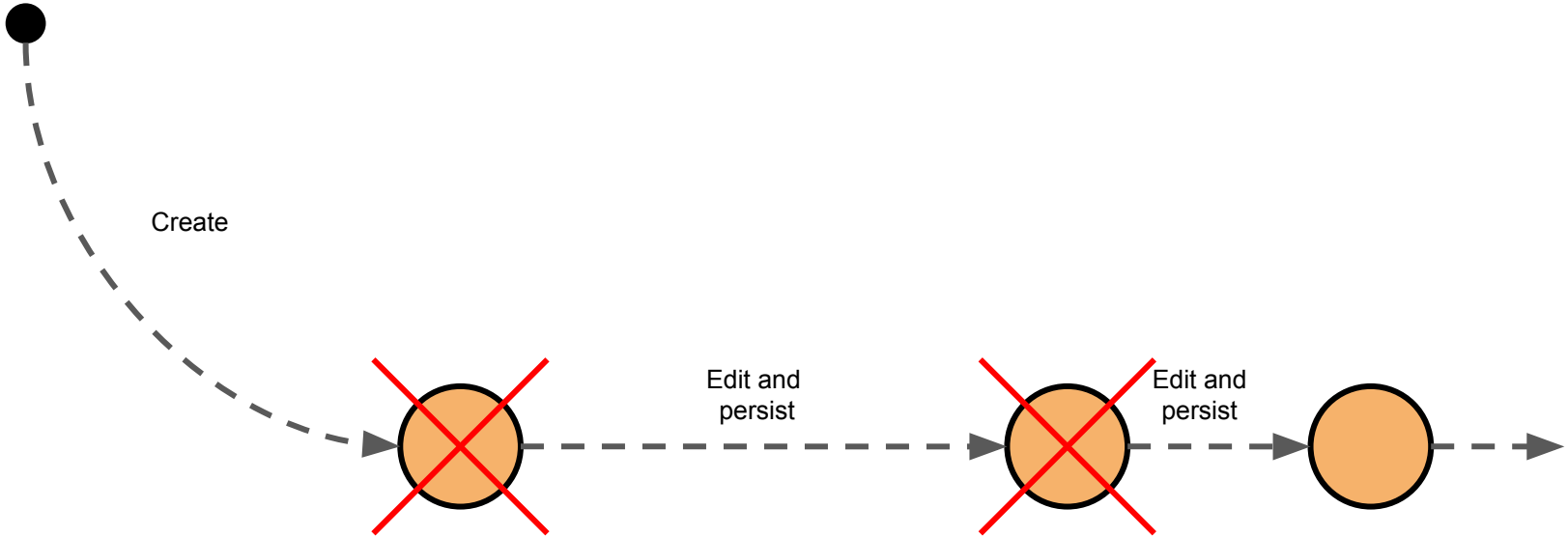
Artifacts’ lifecycle



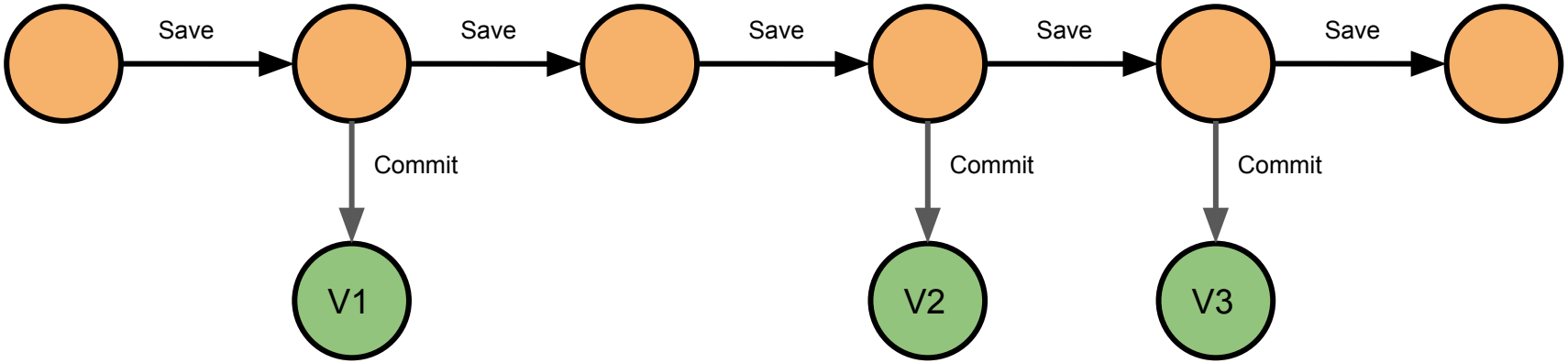
Files’ lifecycle



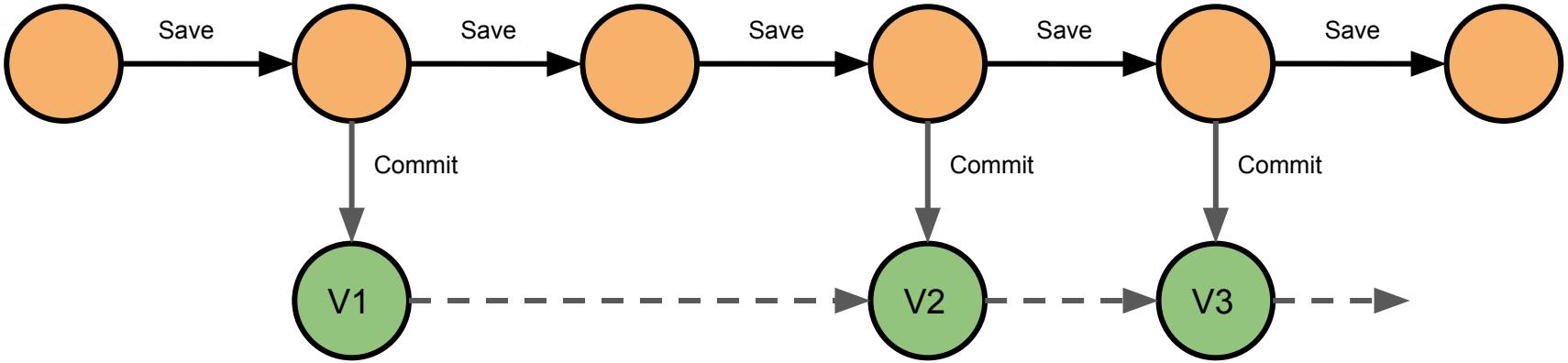
Files’ lifecycle



Files’ lifecycle



Files’ lifecycle



Managing revisions

- Implement a `commit` operation that makes *snapshots* of files
  - Easy to do, e.g.: move snapshots to version-stamped directories or create copies in place appending a version counter at the end of the filename or create a document database or … Let's call this *thing* a **repository**.
- Implement a `restore` operation that retrieves previous snapshots
- We're done, right?

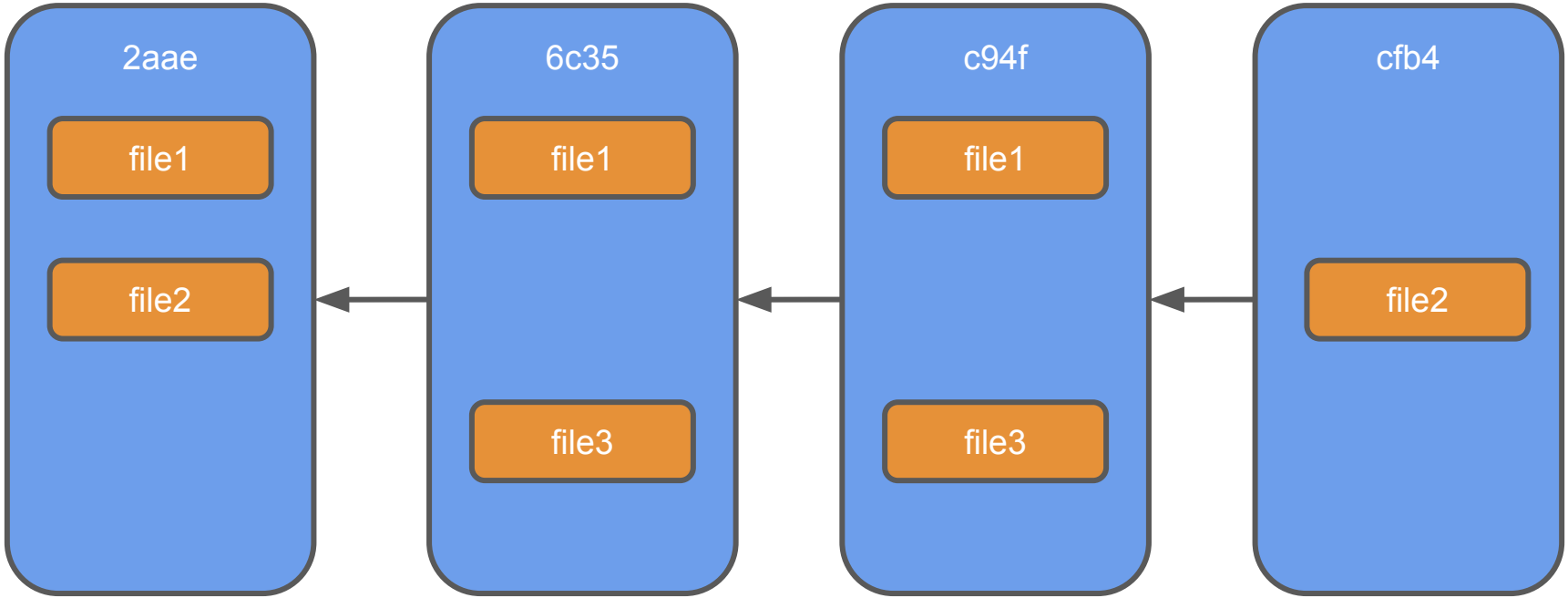
Additional issues

- Metadata
  - When was the snapshot taken?
  - Who edited the file?
  - Why they did it?
- Changesets
  - Are other files involved in the same conceptual change?
- Collaboration
  - How to deal with changes from different developers?

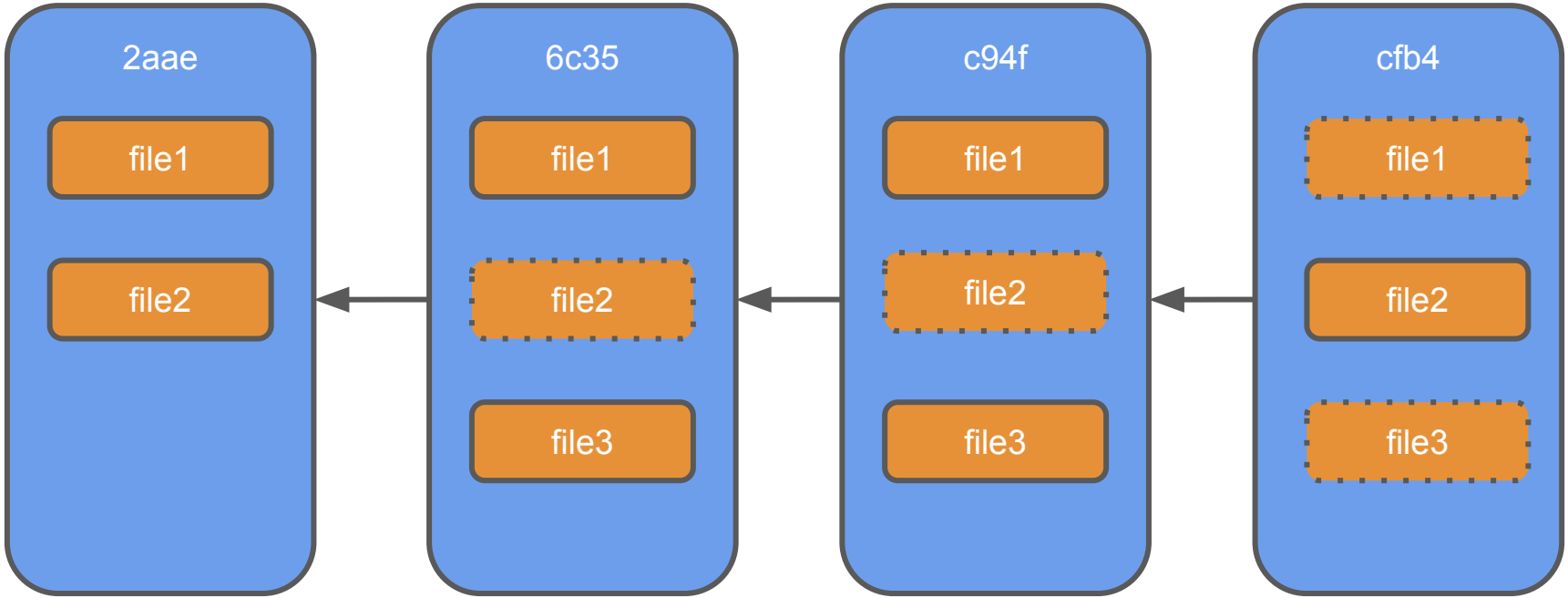
The changeset

When creating revisions we often want to operate not on single files but on a group of correlated files that have all been modified as part of the same conceptual change. This group of files is usually referred to as a *changeset*.

The stream of changesets



The stream of snapshots

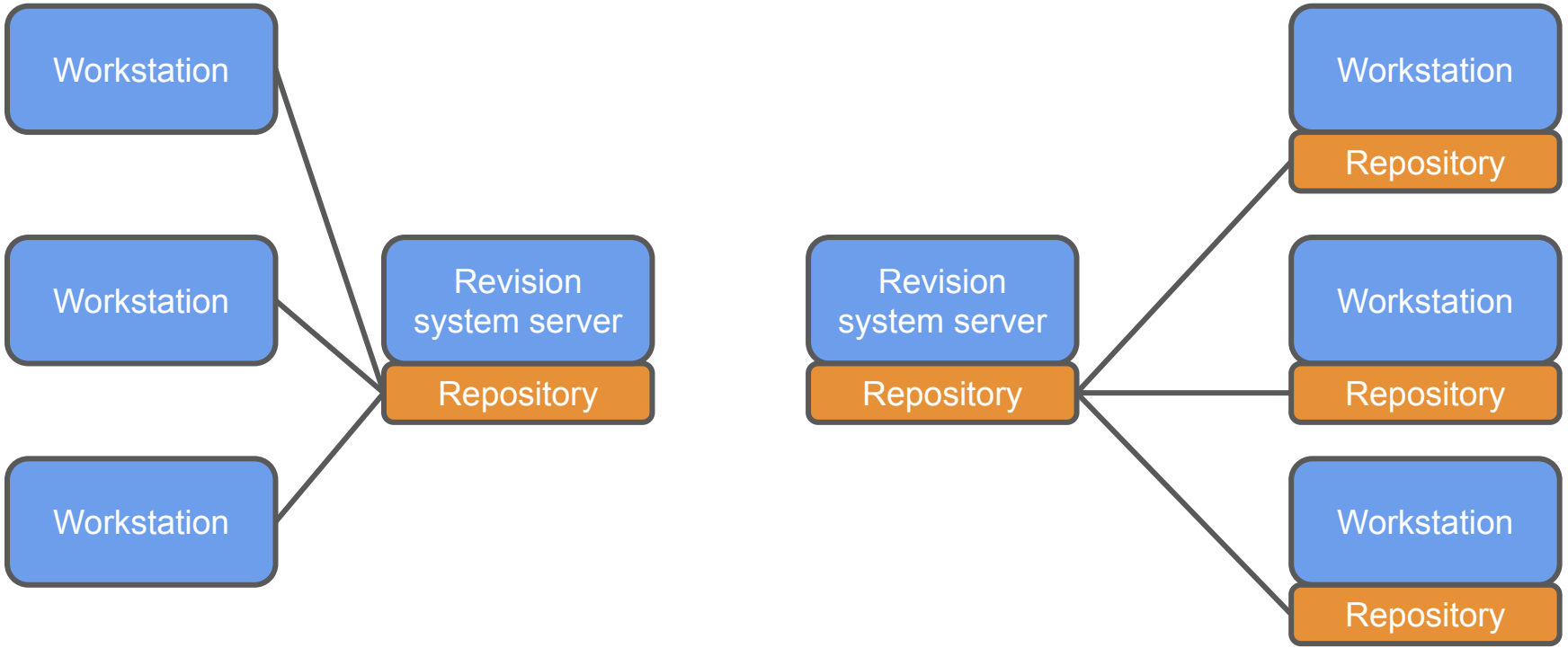


Collaboration

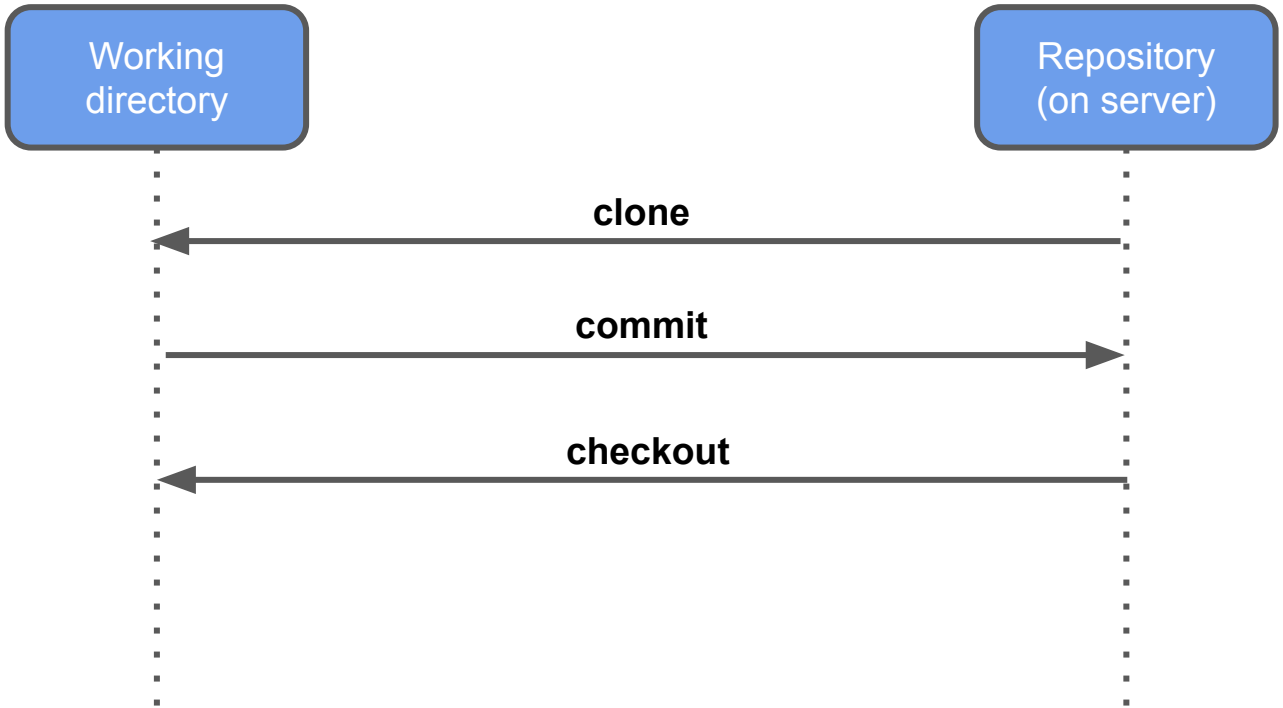
Make the revision system a distributed system.

- Client/server (*centralized*) approach - the revision system runs as a service in a remote host and exposes an API.
- P2P (*distributed*) approach - all workstations have local repositories that can be *synchronized* with remote repositories.
  - Most usually: one remote repository and multiple local repositories that synchronize with the remote one

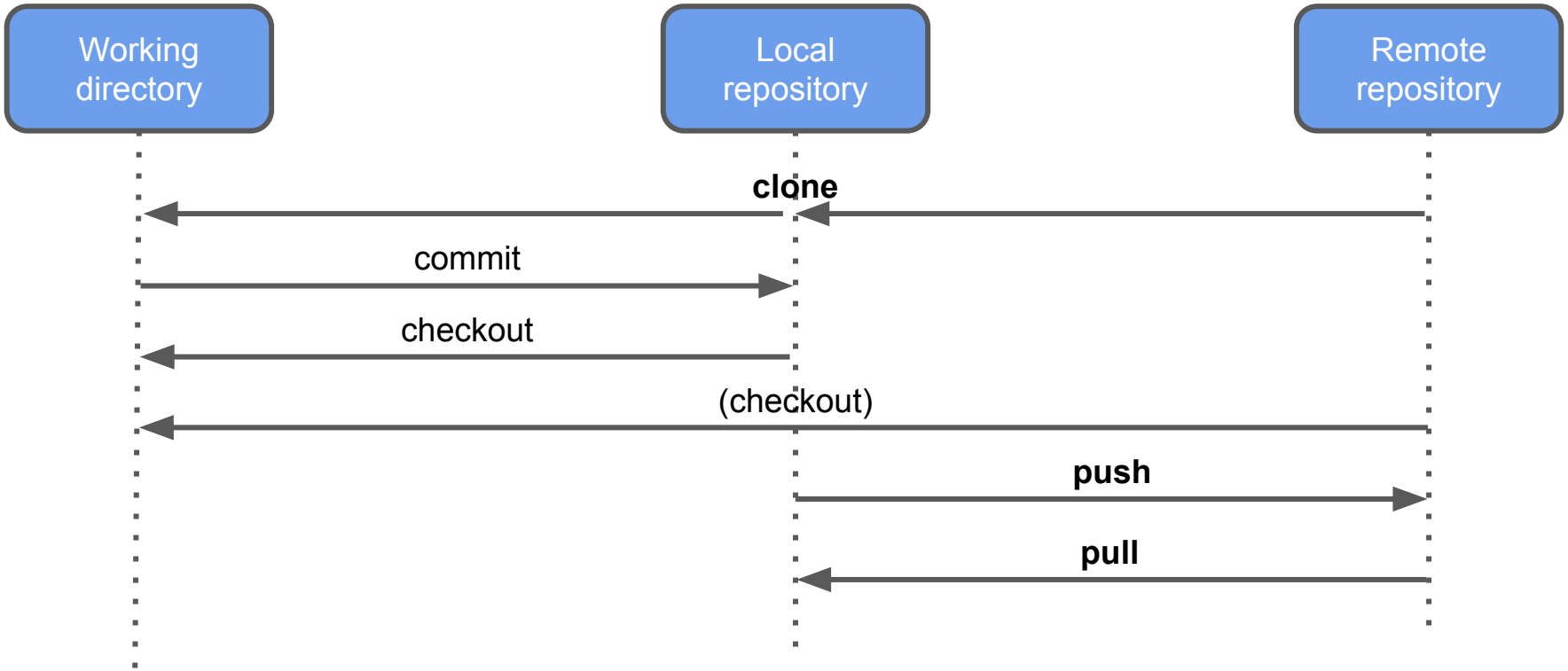
Centralized vs distributed



Centralized revision system



Distributed revision system



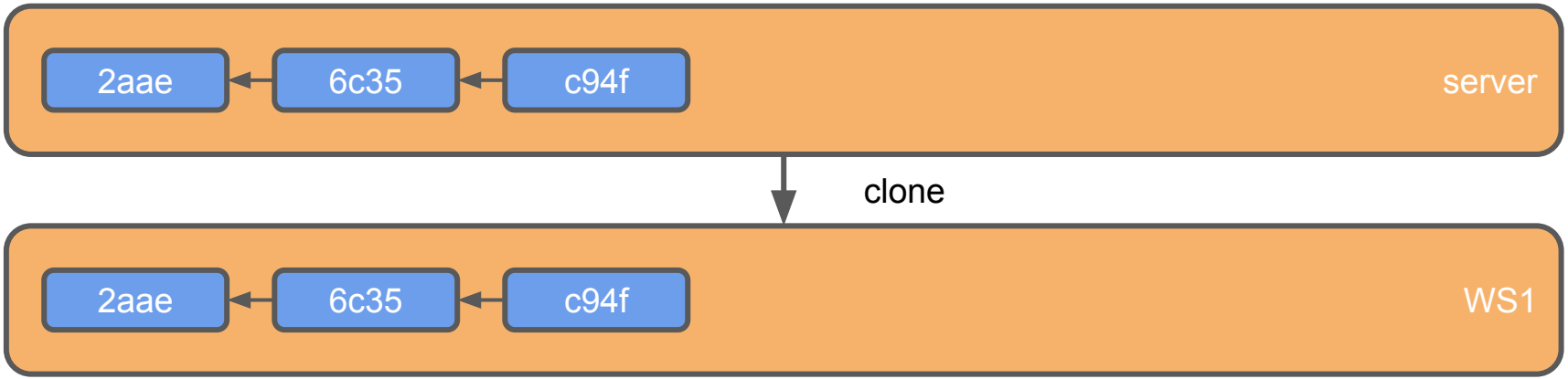
What could ever go wrong?

What could ever go wrong?

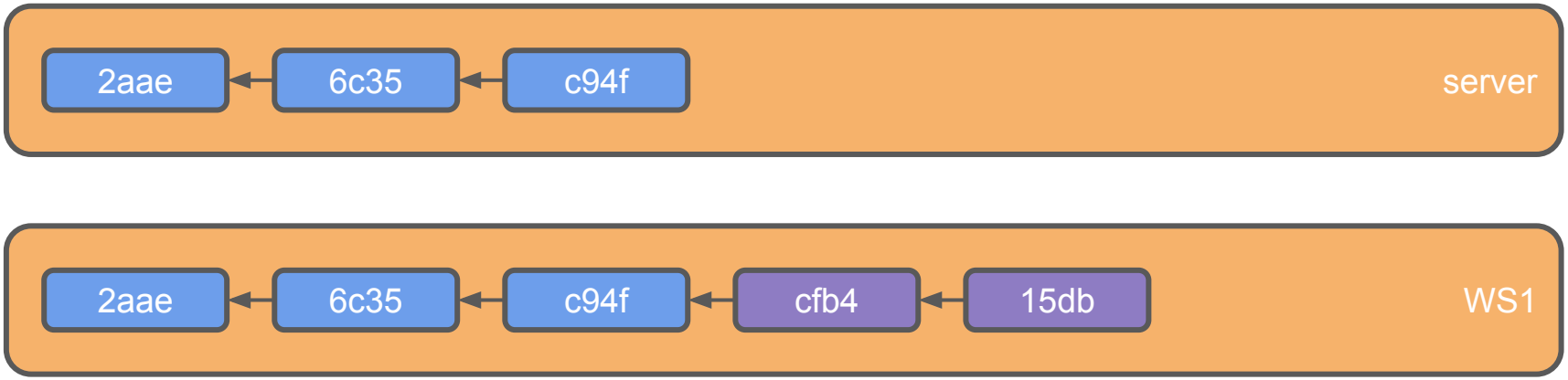
Conflicts: multiple users modify the same file(s) concurrently.

Conflicting histories in distributed VCSs

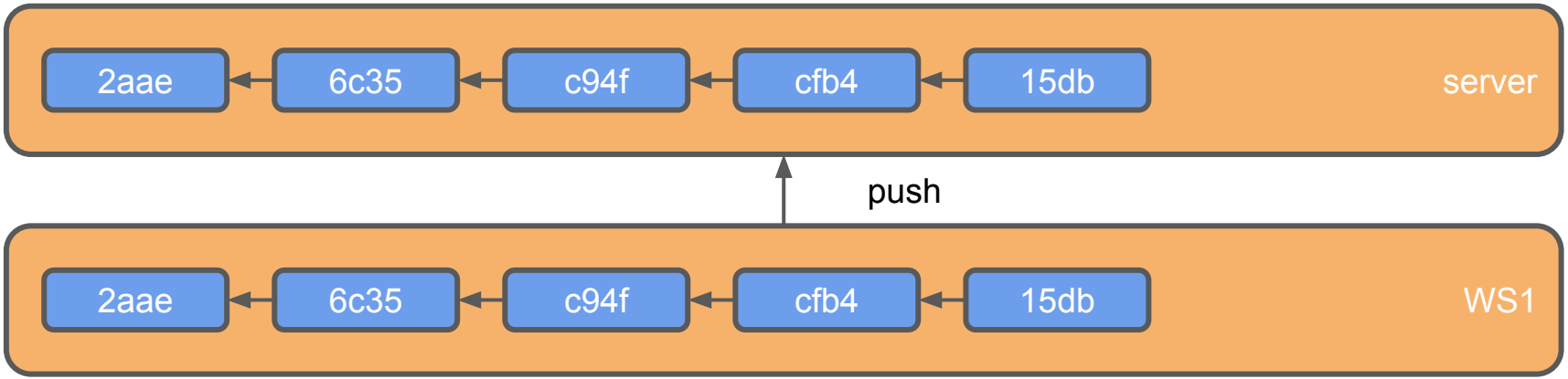
Conflicting histories in distributed VCSs



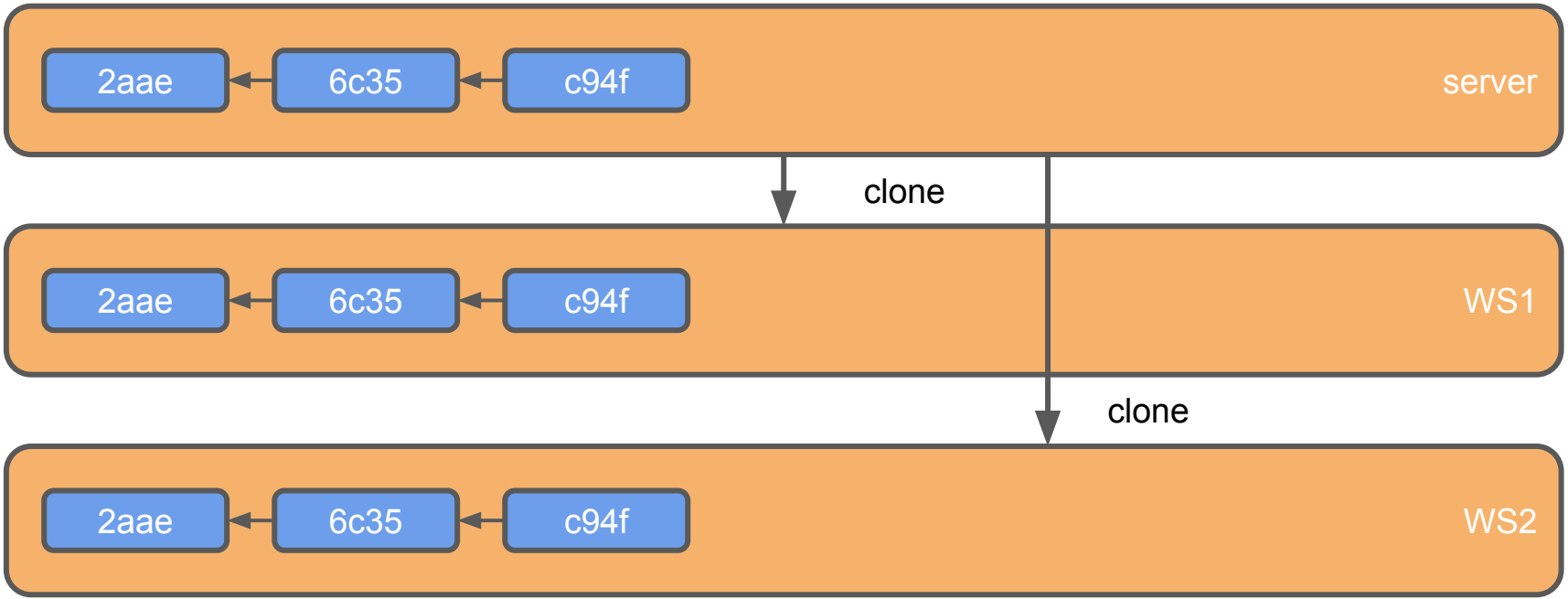
Conflicting histories in distributed VCSs



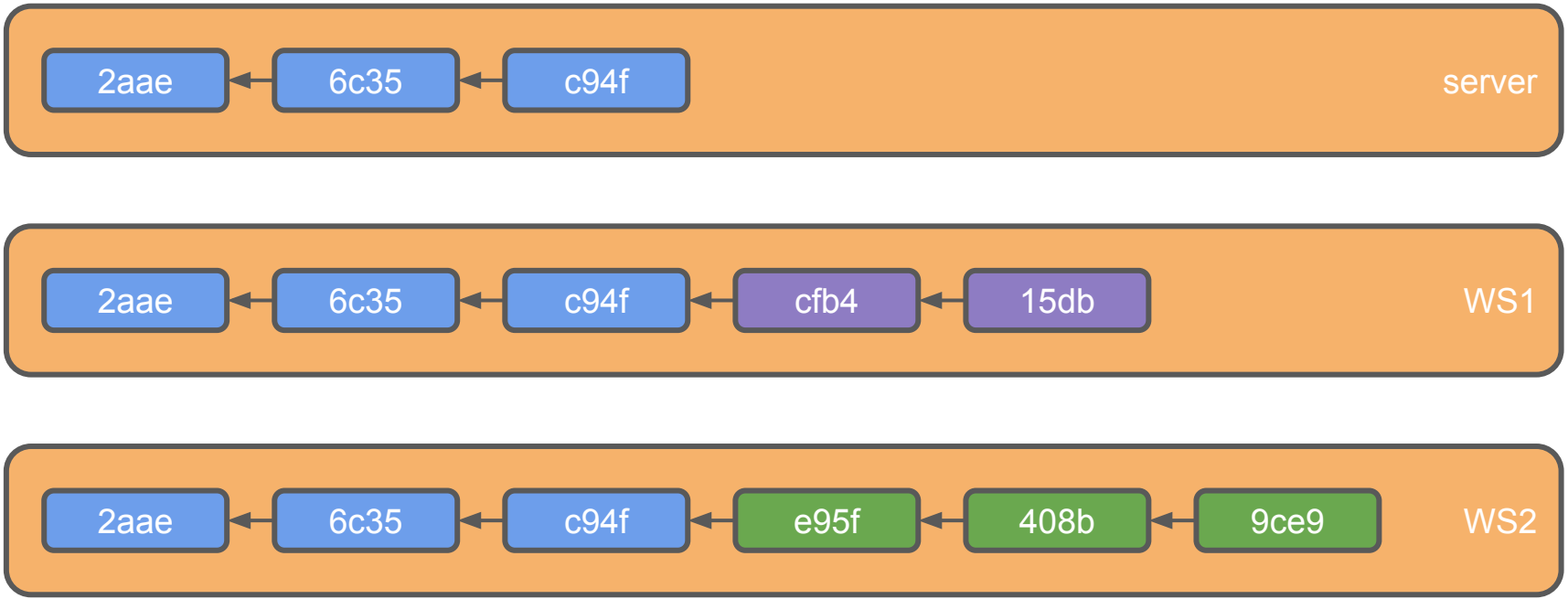
Conflicting histories in distributed VCSs



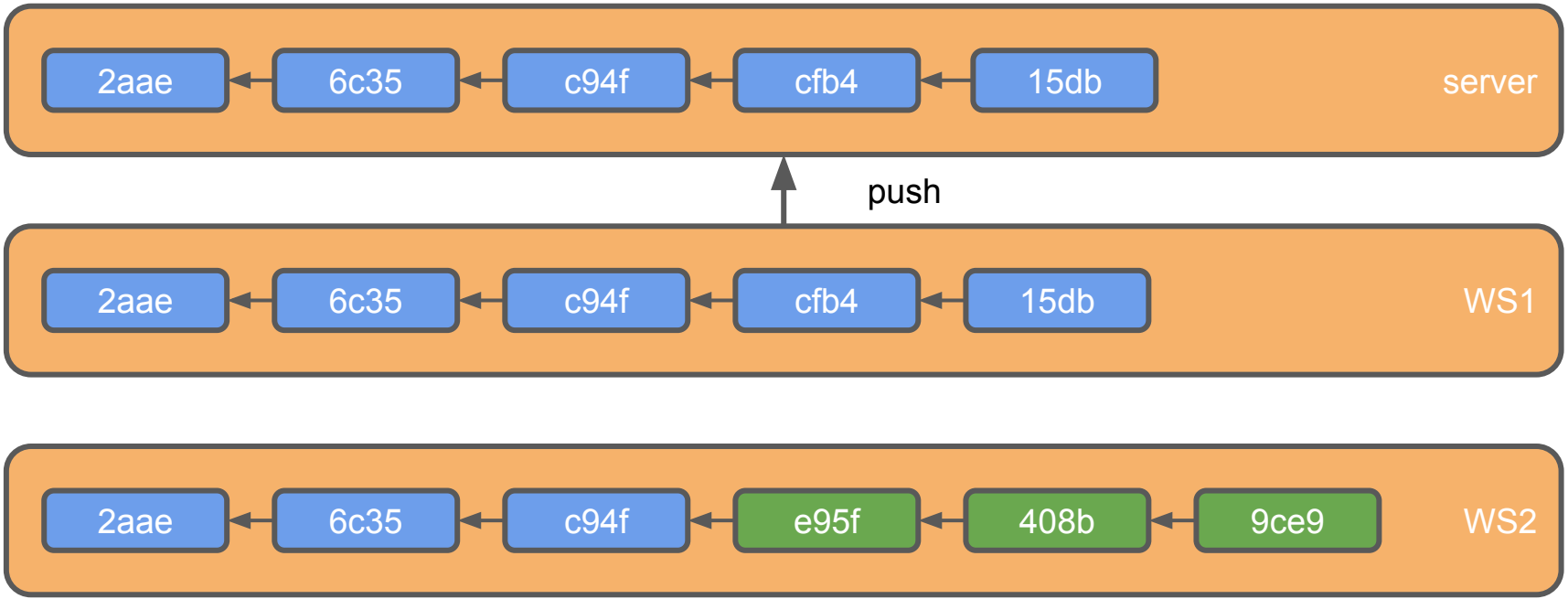
Conflicting histories in distributed VCSs



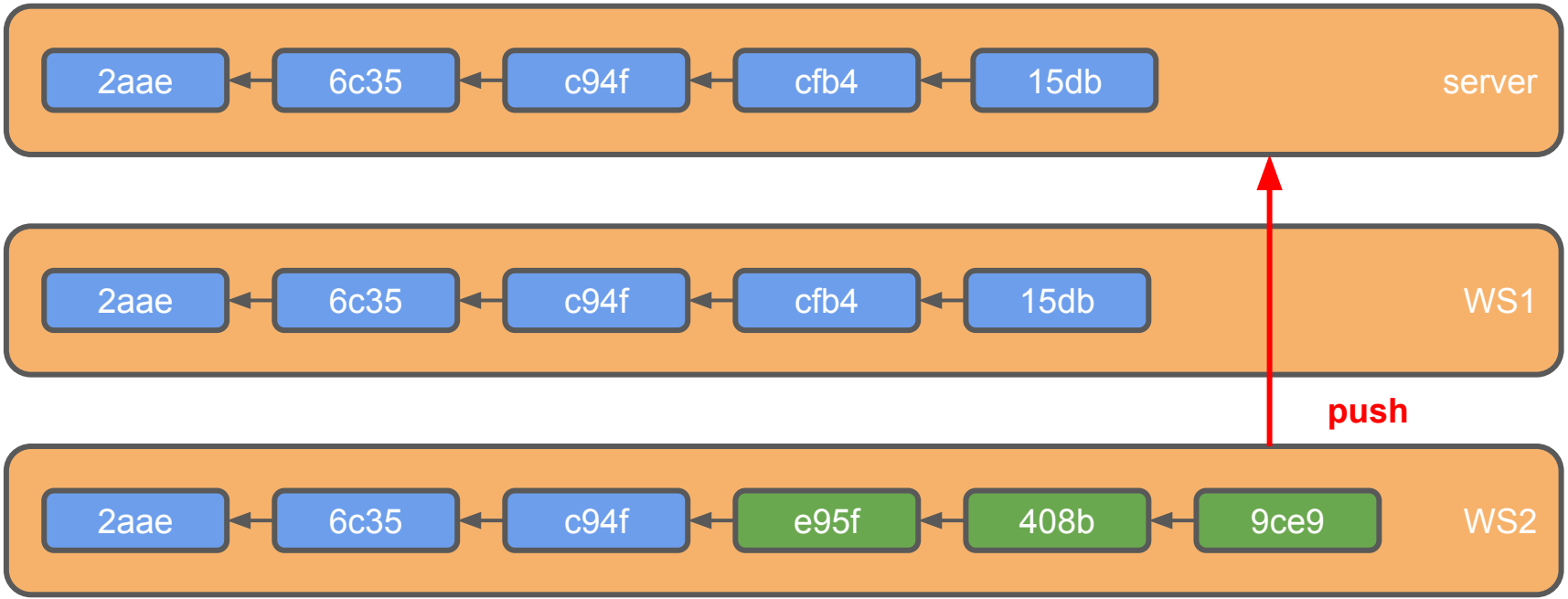
Conflicting histories in distributed VCSs



Conflicting histories in distributed VCSs



Conflicting histories in distributed VCSs



Facing conflicts

- Centralized approach: avoid them
  - Use locking: only the user owning the lock for a file can create revisions for that file.
- Distributed approach: manage them
  - Define *workflows* to minimize the impact of conflicts; when concurrent changes happen, select a change over another or *merge* changes.

Issues with the client/server approach

- Everybody sees everybody's revisions (also if that's partial work).
- If somebody forgets to release a lock, it's a problem.
- Bad habits with lock management impact all developers.
- If the server is *down* everybody is stuck.

git

Git (*noun*): an unpleasant or contemptible person.

[Oxford Languages]

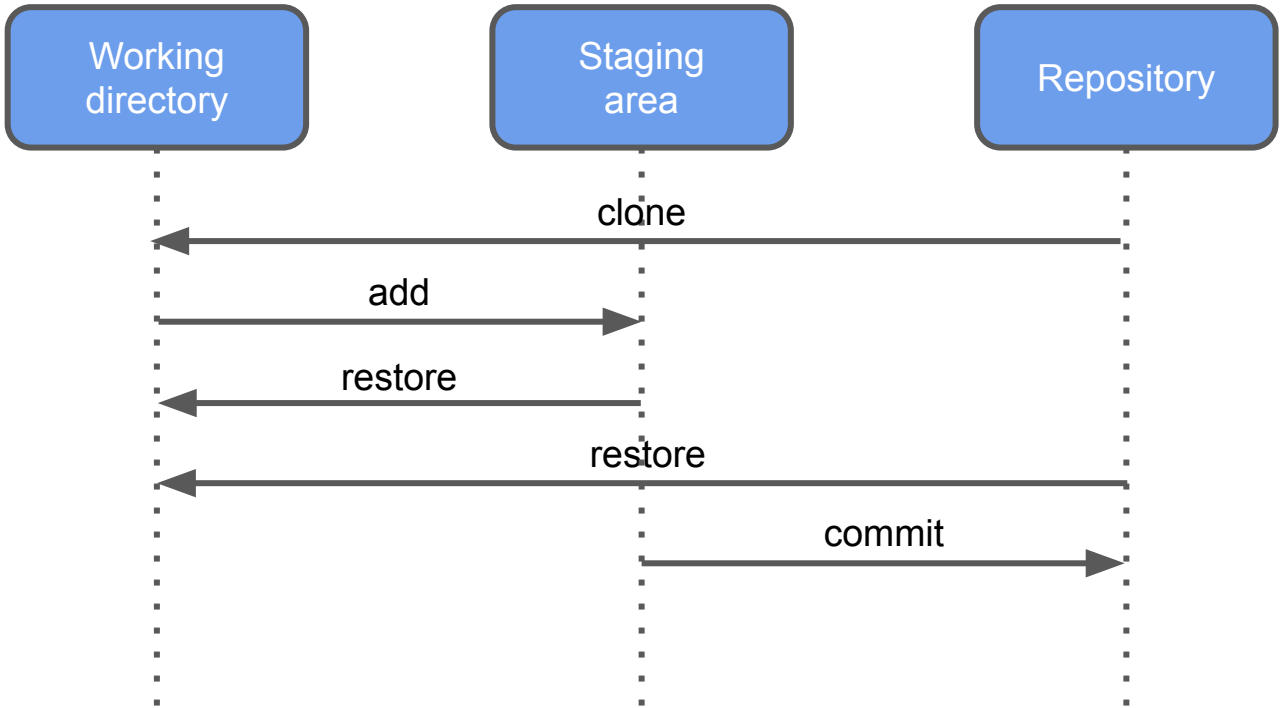
Originally developed by Linus Torvalds in 2005 to support the collaborative development of the Linux kernel. Main design goals: efficiency (esp. w.r.t. branching), robustness.

git

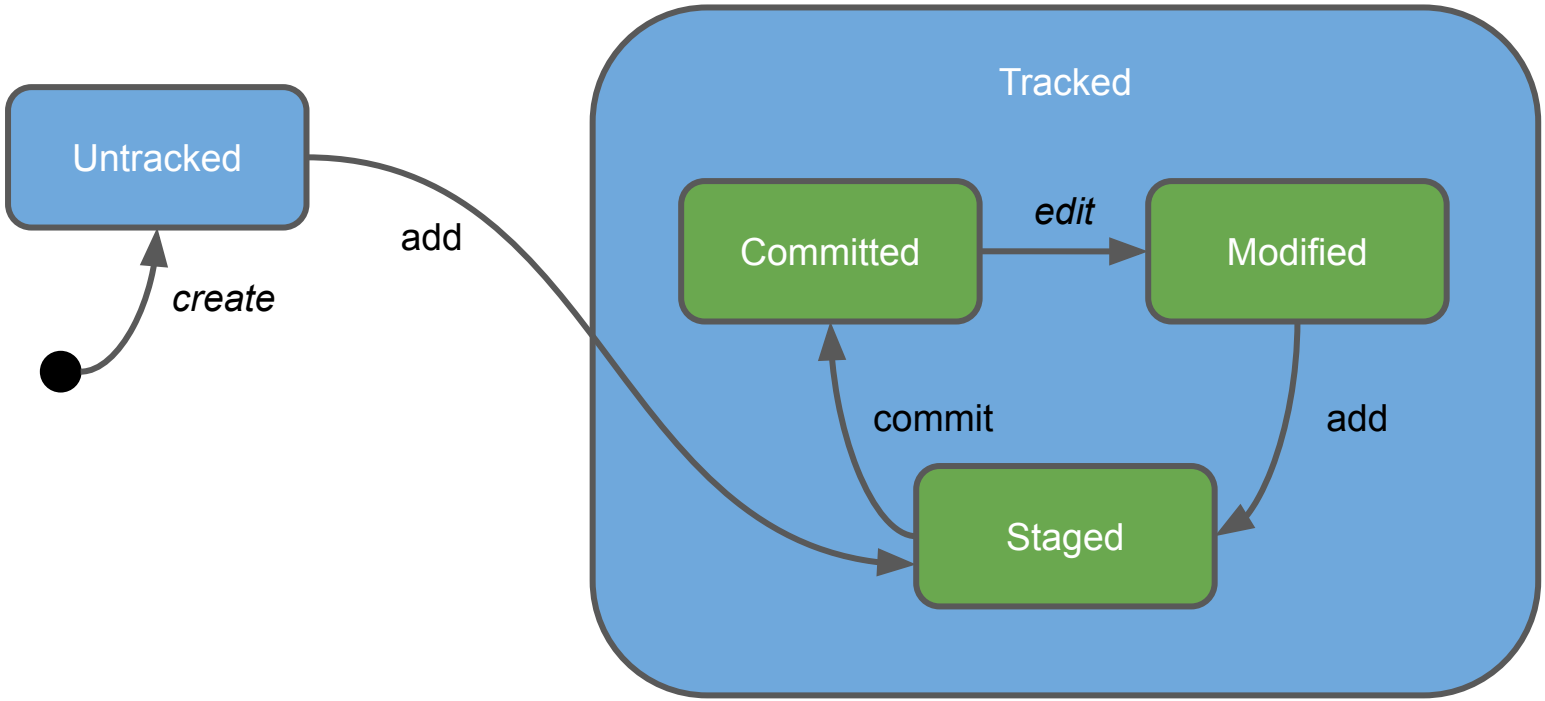
git is a distributed versioning systems, initially devised as a *kernel* on top of which user-friendly versioning systems could be built.

The user friendly systems, however, never appeared.

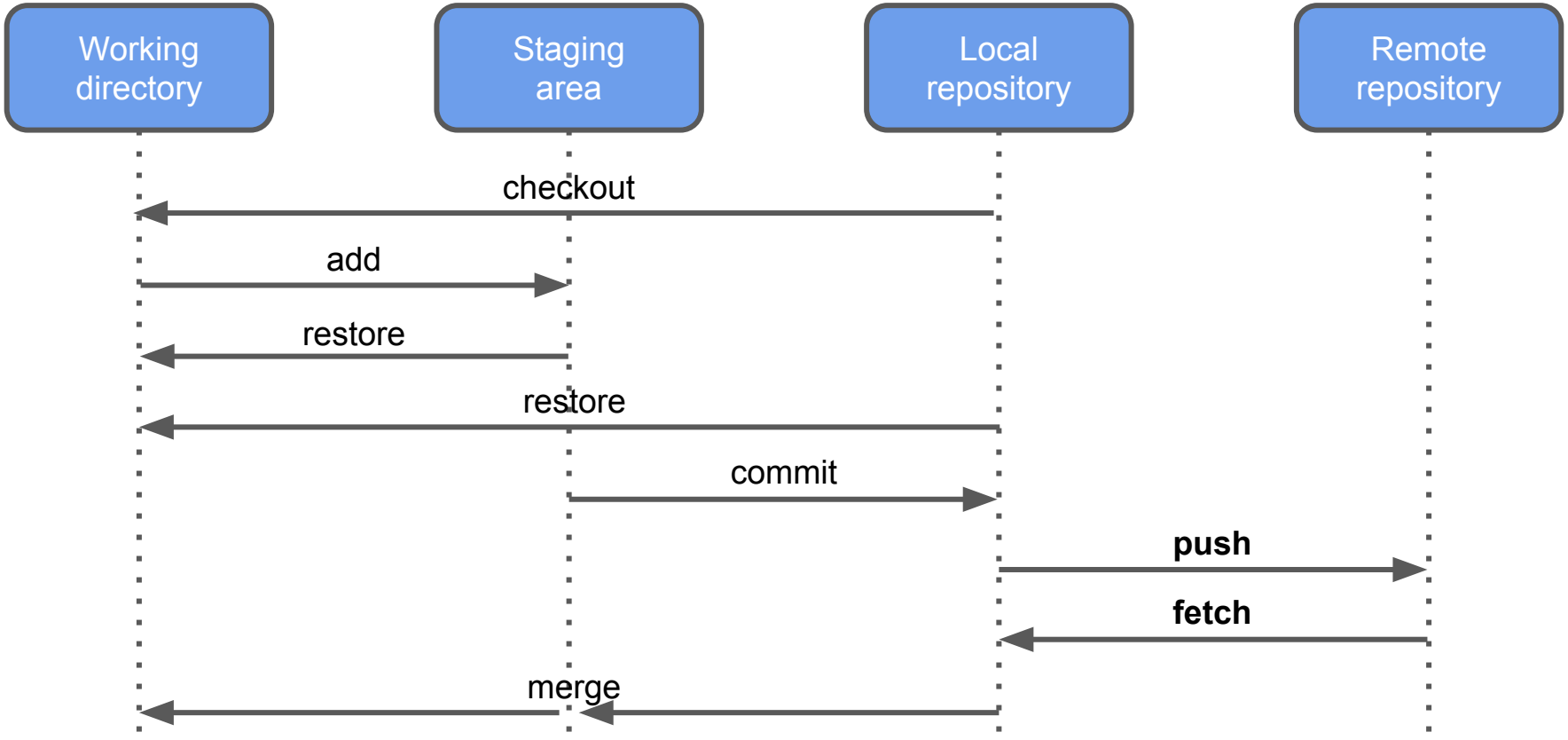
git: the areas



git: the states

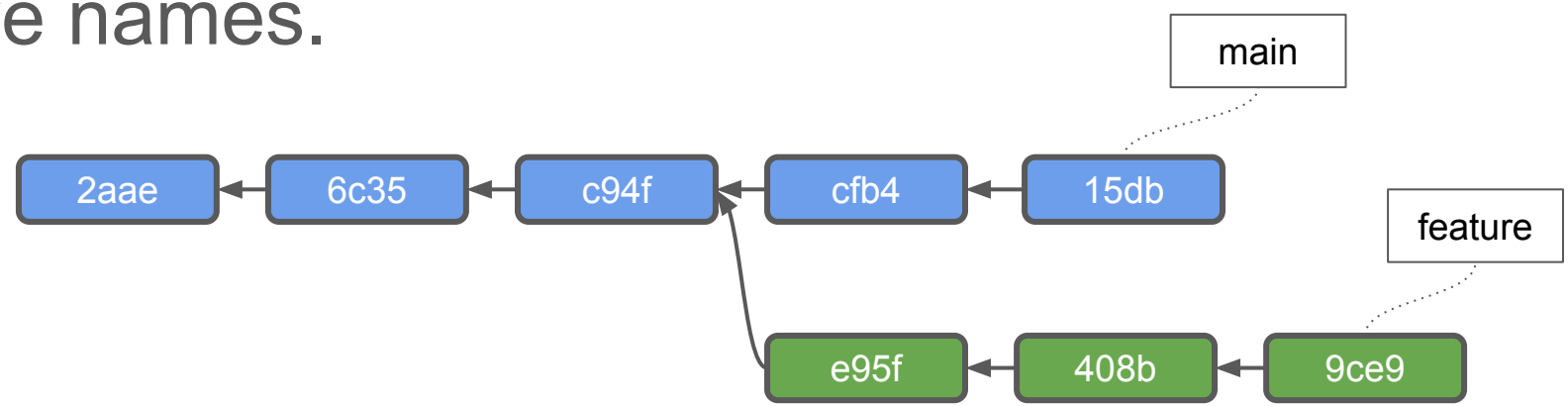


Distributed git



Branching

A branch is a history line diverging from the main one. Branches have to be created explicitly and have names.



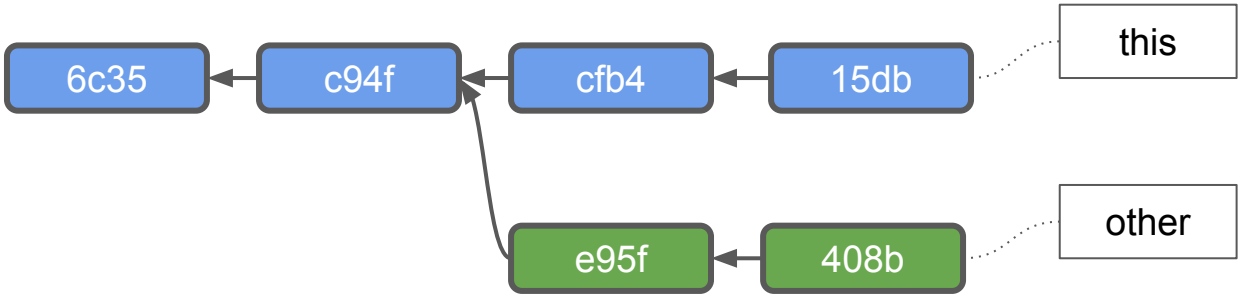
Branches

- Branches form hierarchies:
  - Child branch.
  - Parent (or upstream).
  - Trunk (parent less branch).
- Divergent branches can later be merged (i.e. integrated with an ancestor).
- A branched not intended to later be merged is usually called a *fork.*

Reconcile diverging histories

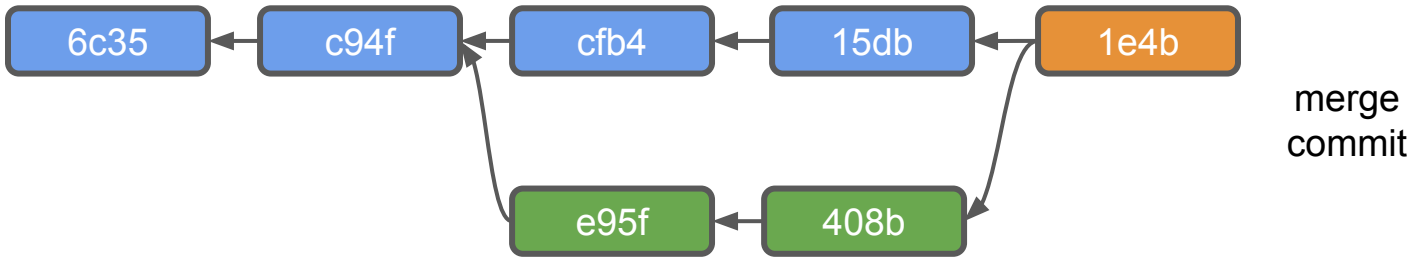
Integrate changes from diverging history lines into a single timeline.

Two main approaches: merge and rebase.



Merge

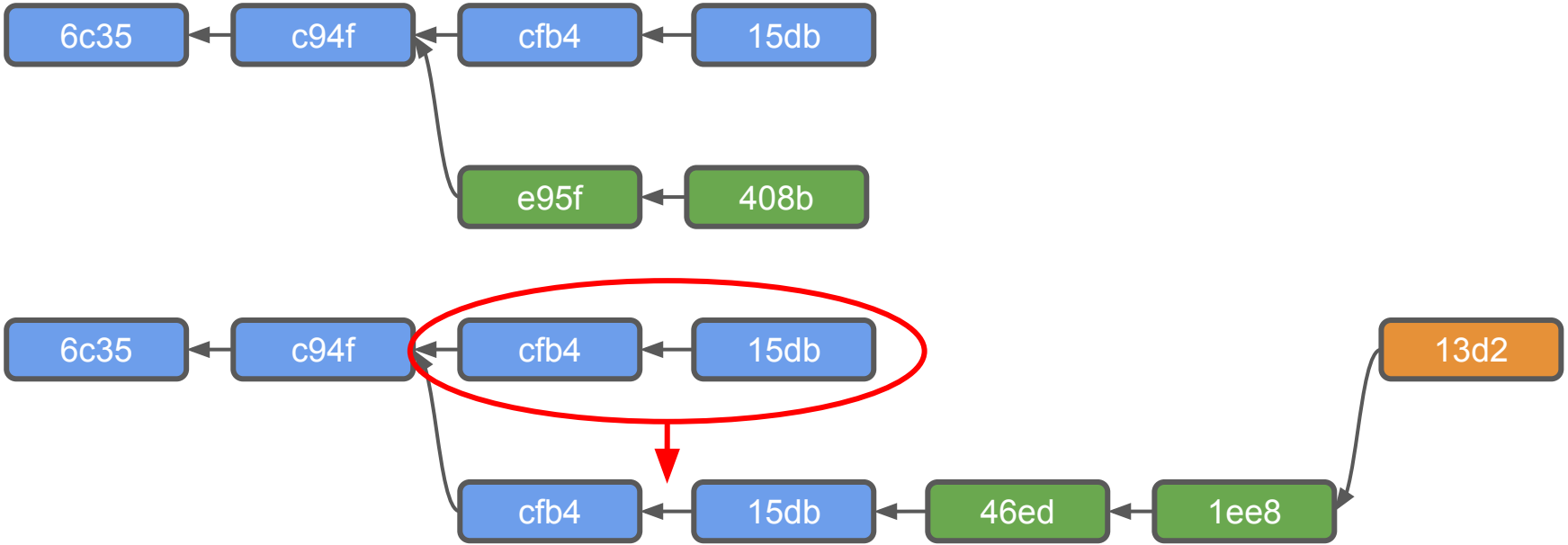
Merging creates a new commit with two parents that integrates the changes from both branches.



Types of merging

- Merging with non overlapping changesets.
  - Create a new history by applying (from a common ancestor) first the changes in *other*, then the changes in *this*.
- Merging with overlapping changesets
  - The same files have been modified in both histories.
    - For text files: if the edits are not overlapping, apply them all.
    - In all other cases: no automatic fix, let the user decide how to resolve conflicts (potentially creating a new revision).

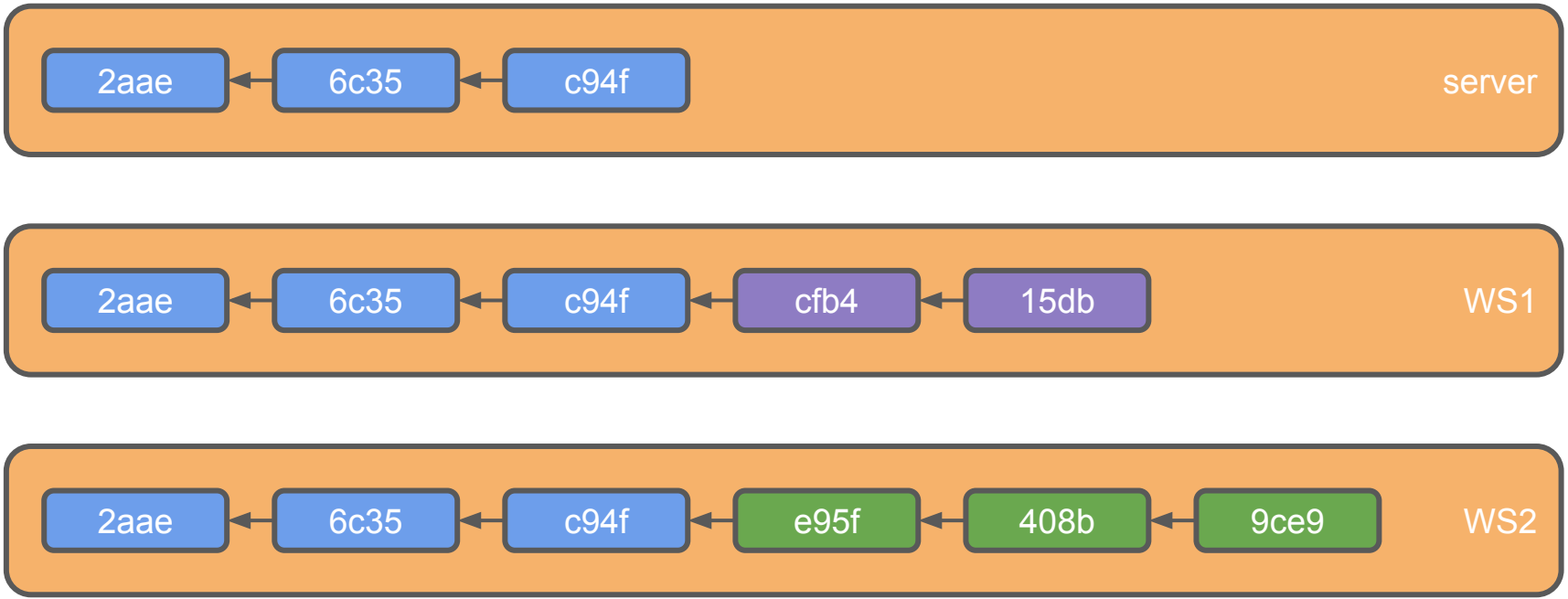
Rebase


6c35
c94f
cfb4
15db
e95f
408b
6c35
c94f
cfb4
15db
13d2
cfb4
15db
46ed
1ee8

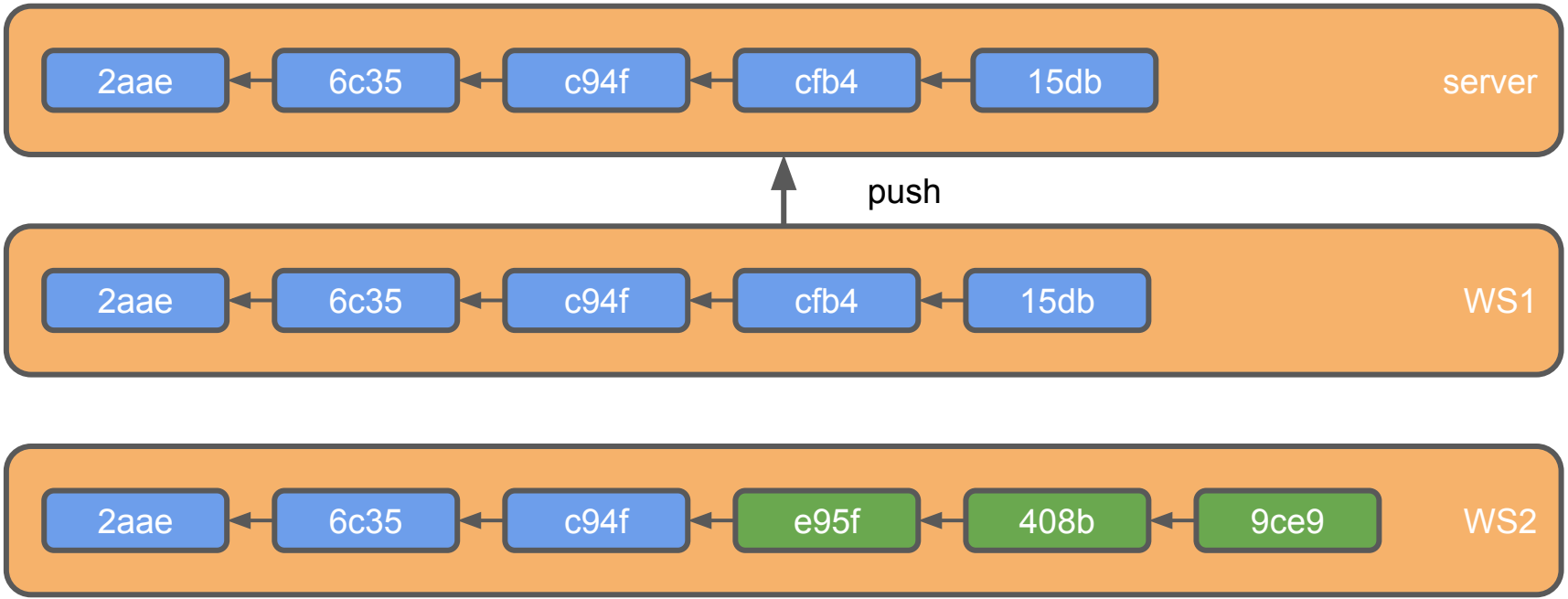
git centralized workflow

- No branches.
- When conflict takes place (during `push` operations) align local history with remote (e.g. rebasing with `pull --rebase`) and push again.

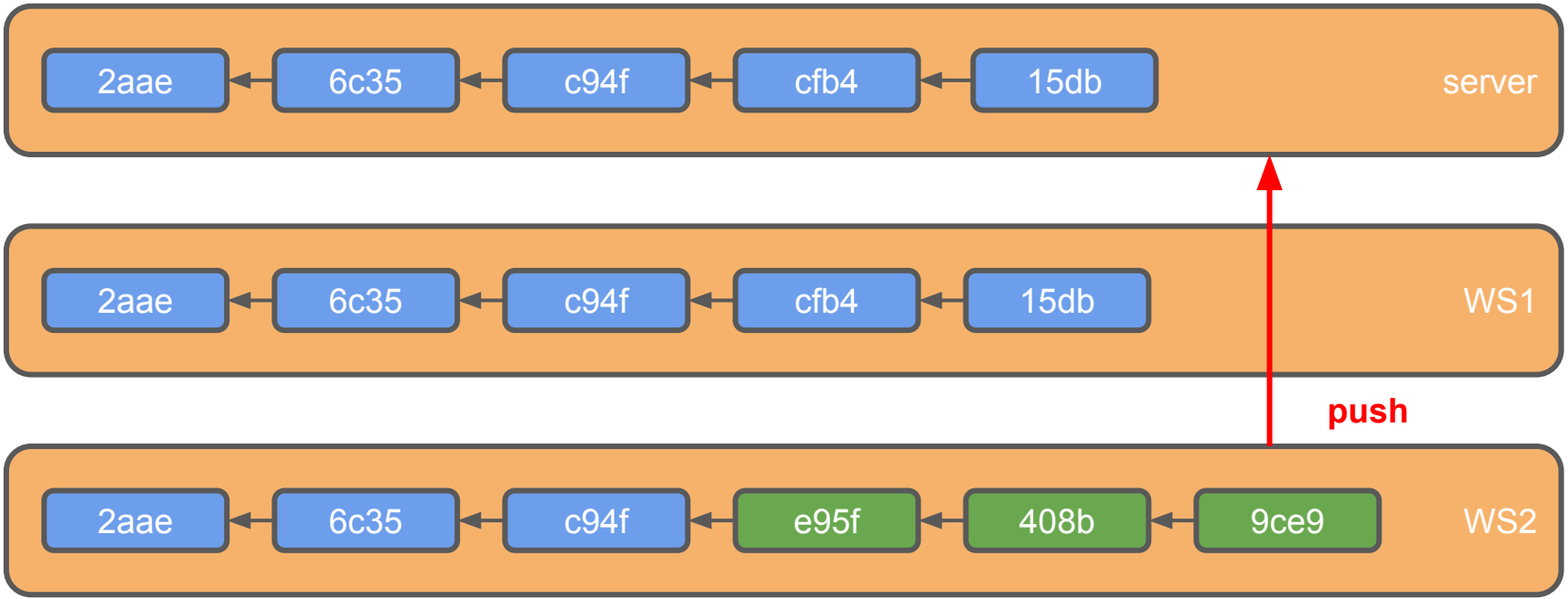
git centralized workflow



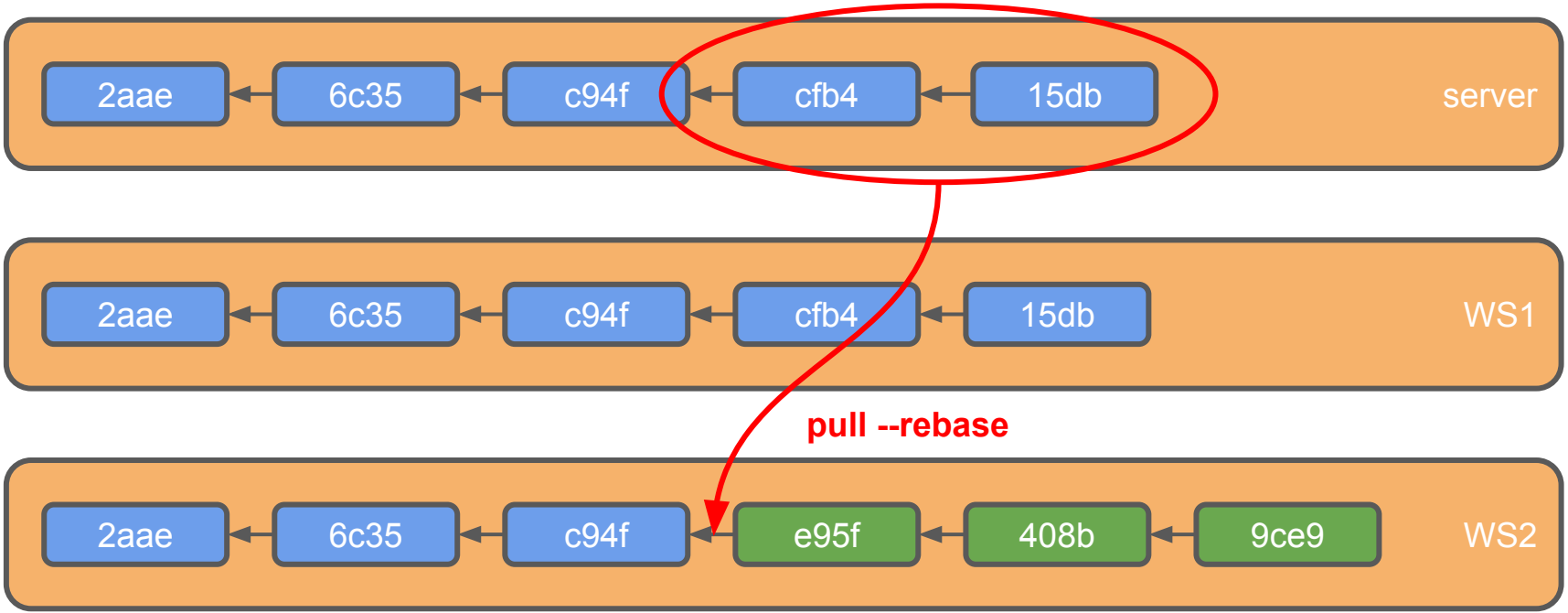
git centralized workflow



git centralized workflow



git centralized workflow



Centralized WF: conflict resolution

```
> git pull --rebase
```

```
> git log
```

```
<list of conflicting files>
```

“Fix” the conflicting files

```
> git add <fixed files>
```

```
> git rebase --continue
```

```
> git push
```

git branching workflows

Several *branching workflows* have been proposed, mostly as variations of *feature branching*.

In feature branching developers working on new features create a branch, commit to this branch until their work is done, then the branch is merged (and removed).

Pull requests

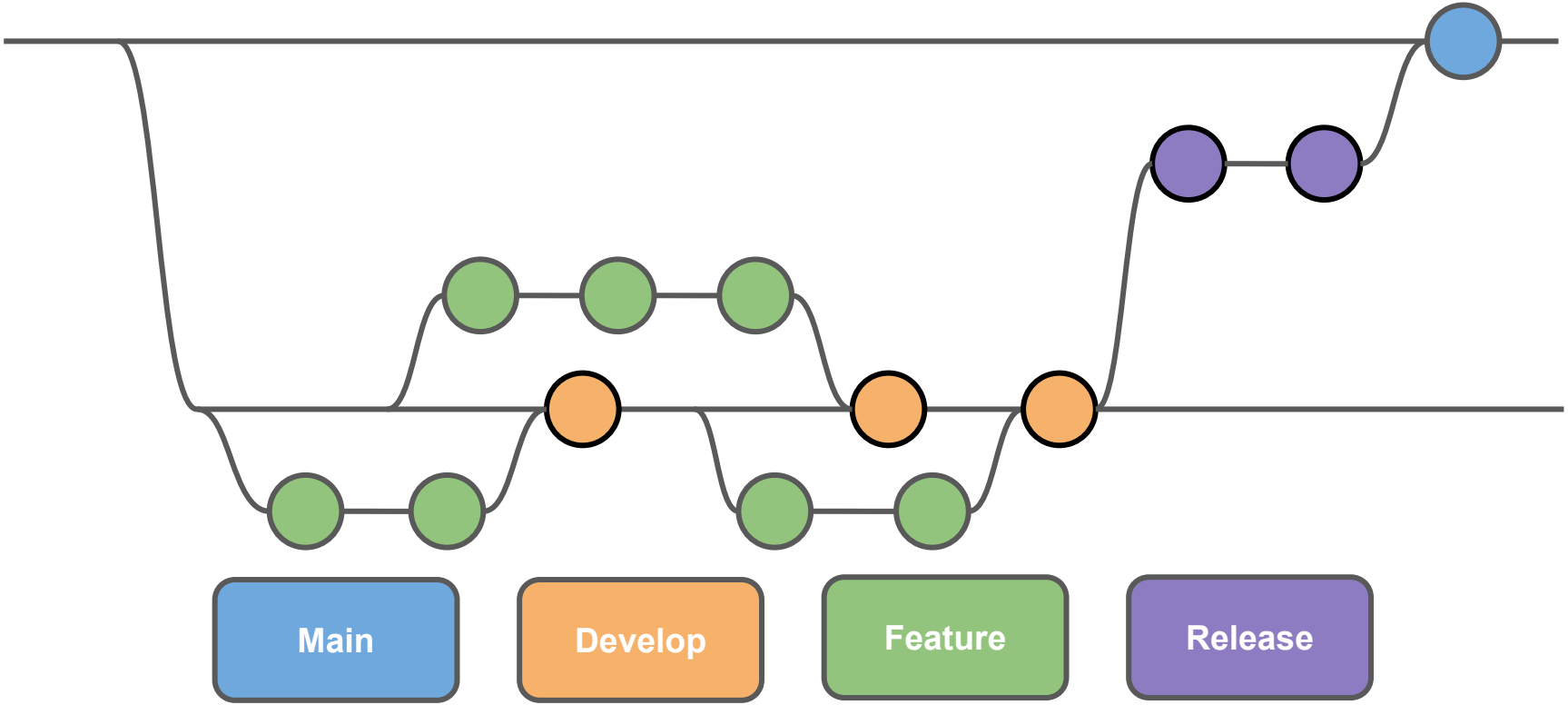
A pull request (or *merge request*) is a *practice*, it is not a technical feature of the VSC.

When adopting pull requests, the merging of a feature branch is not performed by the developer working on the branch but by a *repository maintainer* (possibly after a *review process*).

Gitflow

- Two main (historical) branches: **Main** and **Develop**.
- **Feature** branches derive from Develop (and are merged with it).
- When getting close to a release a **Release** branch is derived from Develop; no new features are committed to a Release branch; when ready to ship Release is merged with Main.

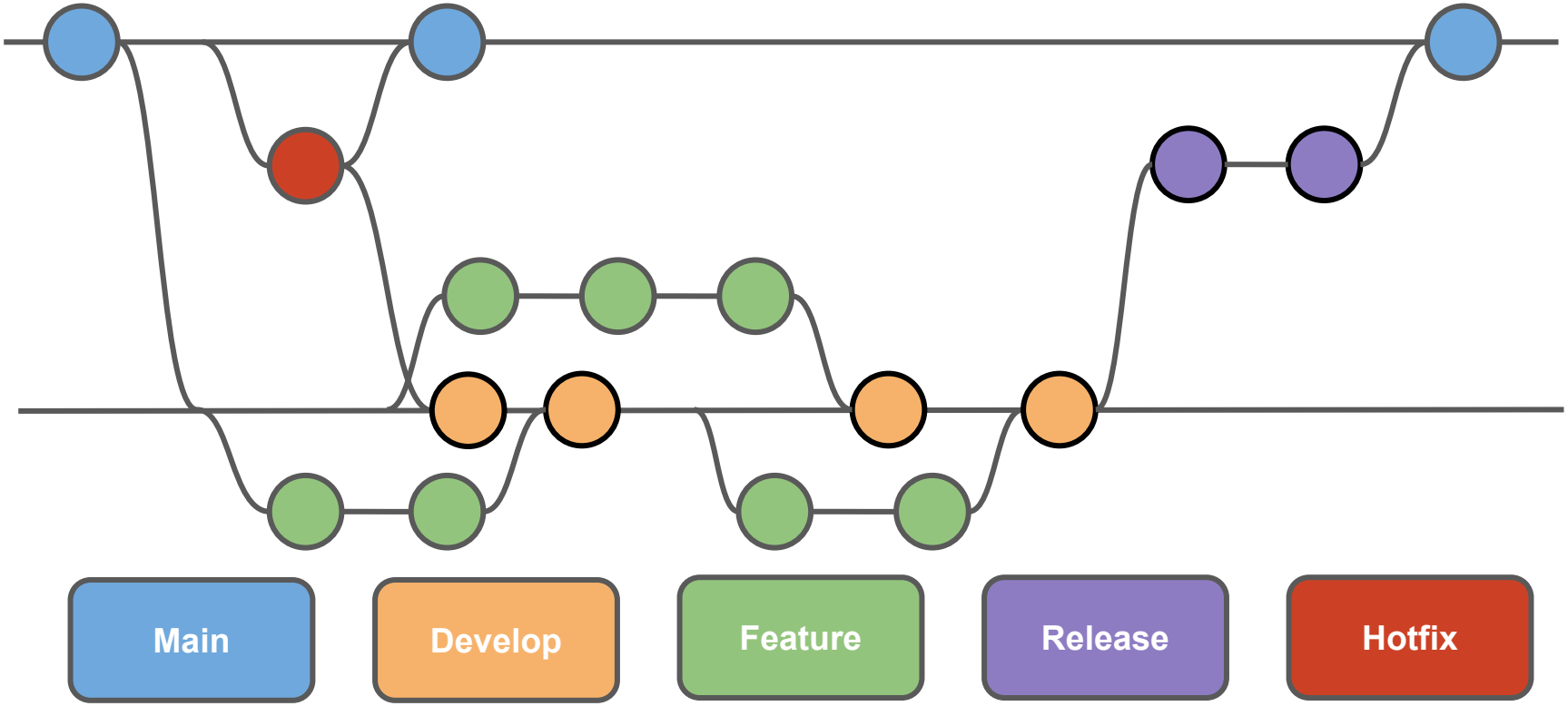
Gitflow



Gitflow: hotfixes

Hotfix branches are the only branches derived from Main; they are used for bug-fixing only, and soon merged back.

Gitflow: hotfixes



Hash, refs and tags

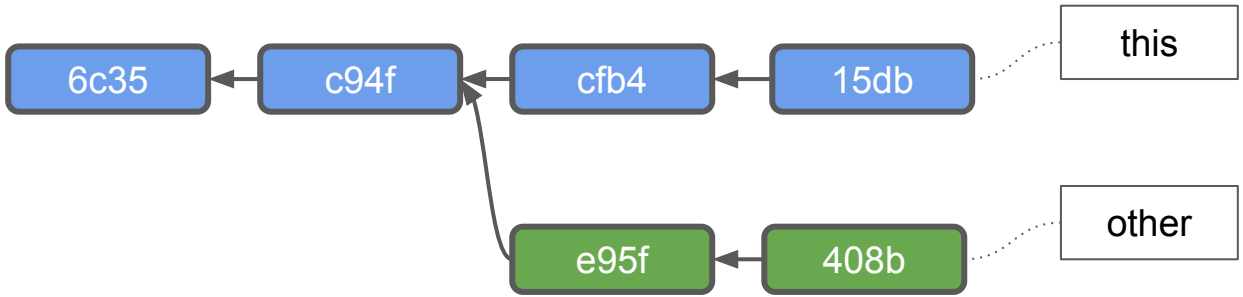
git commits (as other *objects*) are identified by a unique id (a *SHA-1 hash*).

Refs are names that can substitute for hashes (or for other refs in the case of *symbolic refs*). git uses several refs to keep track of the repo's status. It is common for refs to change their content, for example to track the last commit of the current history.

Tags are **friendly, stable, non-symbolic** refs.

Branches and refs

Branches are just refs pointing to a *tip* commit.



HEAD

The current status of the working directory is always computed with respect to a specific commit. **HEAD** is the ref pointing to that commit.

Changing *current branch*, just makes HEAD refer that branch (which, in turn, refers the tip commit of a history line).

HEAD is usually a symbolic ref associated to a branch, if you make HEAD point directly to a non-tip commit (with, e.g. `git checkout`), your HEAD becomes *detached*.