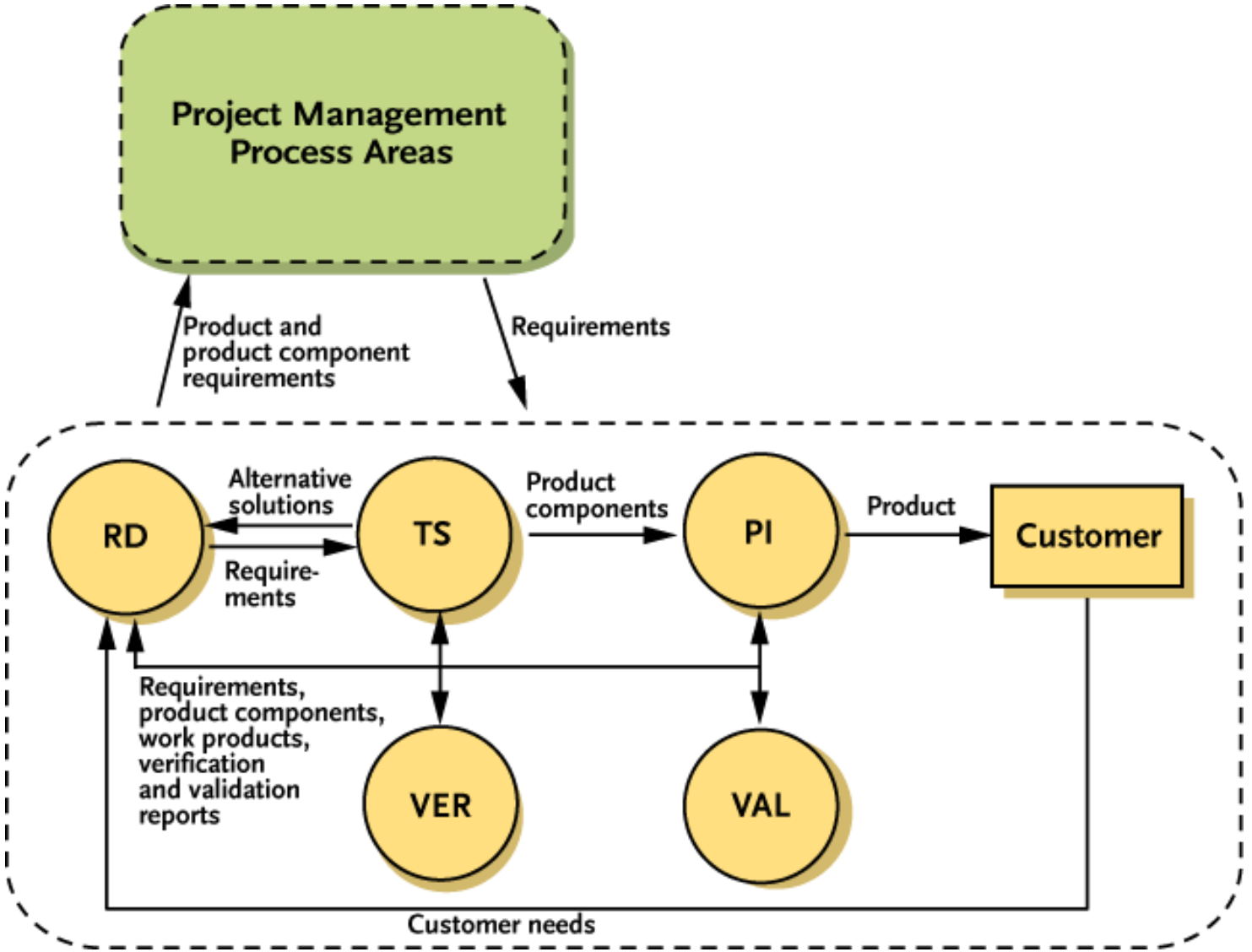
Užduočių programavimas komandoje: versijų kontrolės sistema

Dr. Asta Slotkienė

The Engineering (CMMI-DEV) process areas



The Engineering (CMMI-DEV) process areas: **Technical Solutions**

TS.SG **1 Select Product Component Solutions**

Product or product component solutions are selected from alternative solutions.

TS.SG 2 **Develop the Design**

Product or product component designs are developed.

TS.SG 3 **Implement the Product Design**

Product components, and associated support documentation, are implemented from their designs.

- TS.SP 3.1 **Implement the Design**: implement the designs of the product components.
- TS.SP 3.2 **Develop Product Support Documentation: d**evelop and maintain the end-use documentation.

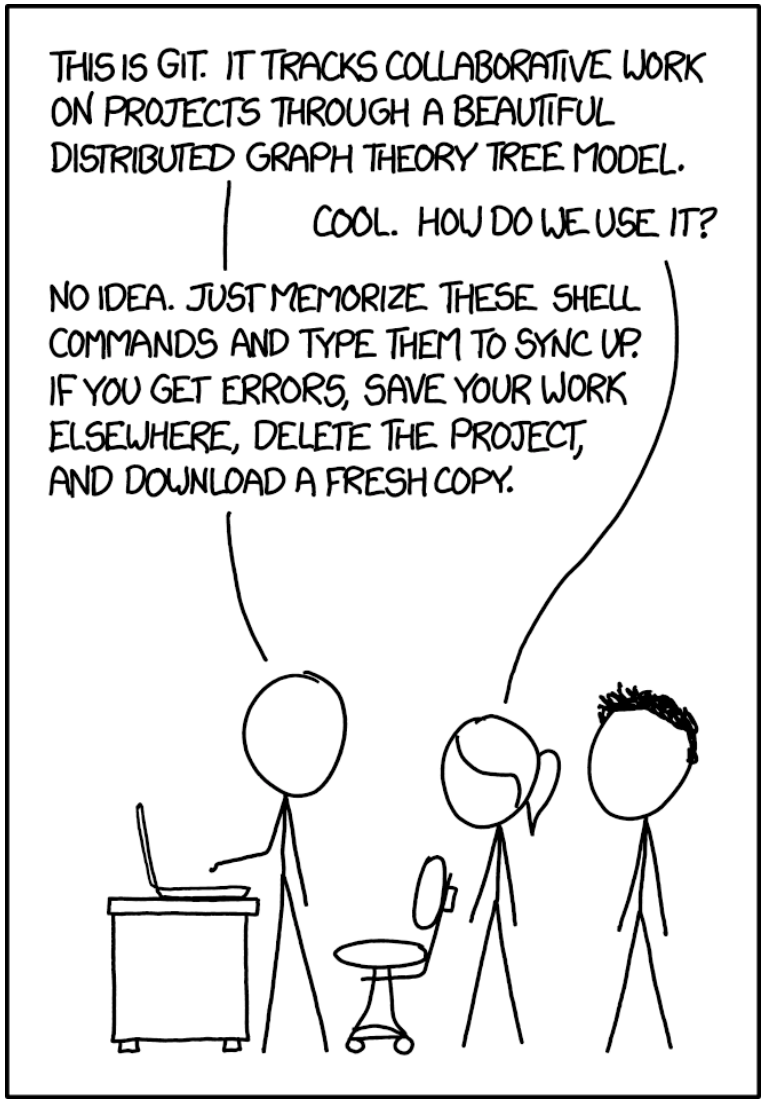
https://www.wibas.com/cmmi/tssp-31-implement-the-design

How to effectively manage a team (tasks/issues programing)

How to effectively manage a team (tasks programing)

1. Ideally, different people work on different features
2. Need to separate your work from that of others
3. Need to be able to only add your work when it's done

How to effectively manage a team



Reason to Use Version Control

- You find a set of files like this in a directory:

```
Report_draft.txt
Report.doc
Report2.doc
Report2a.doc
Report_2019_03_02.doc
...
NewReport_2019_05.doc
NewReport_submitted.doc
NewReport_submittedv2.doc
```

- Can I just use the newest?

Folder system/File system

- Progressively harder to manage
- What if you have collaborators?
- What if you want to go back to a certain state?
- How do you track changes?

Problems Working Alone

- Ever done one of the following:
  1. dead code that worked
  2. made changes and saved it, which broke the code,
  3. and now you just want the working version back…

  - *Accidentally deleted a critical file, and hundreds of lines of code are gone…*
  - *Somehow messed up the structure/contents of your code base, and want to just "undo" the crazy action you just did*
  - *Hard drive crash!!!! Everything's gone the day before the deadline.*

- Possible options:
  - Save as (MyClass-v1.java)
  - UPS. And now a single line change results in duplicating the entire file…

Problems Working in the Teams

- Whose computer stores the "official" copy of the project?
  - Can we store the project files in a neutral "official" location?
- Will we be able to read/write each other's changes?
  - Do we have the right file permissions?
  - Lets just email changed files back and forth!
- What happens if we both try to edit the same file?
  - X member just overwrote a file I worked on for 6 hours!
- What happens if we make a mistake and corrupt an important file?
  - Is there a way to keep backups of our project files?

How do I know what code each teammate is working on?

Version Control System

- A software tool that helps you **keep track of changes** in your data (code) over time

- Version Control System (VCS) is a tool or set of tools that provides management of changes to files over time:
  - Uniquely identified changes (**what**)
  - Time stamps of the changes (**when**)
  - Author of the changes (**who**)

What is version control?

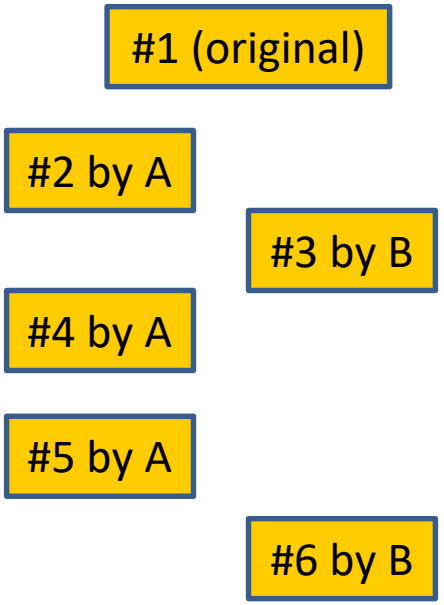
- **Version control** (or *revision control*) is the term for the management of source files, and all of the intermediate stages as development proceeds.
- A **version control system** is a repository of files.
  - Every change made to the source is tracked, along with who made the change, etc.
- Other items can be kept in a **version control system** in addition to source files -- Project Charter, Product Backlog, Design Document, Sprint Planning Document, Sprint Retrospective….

History

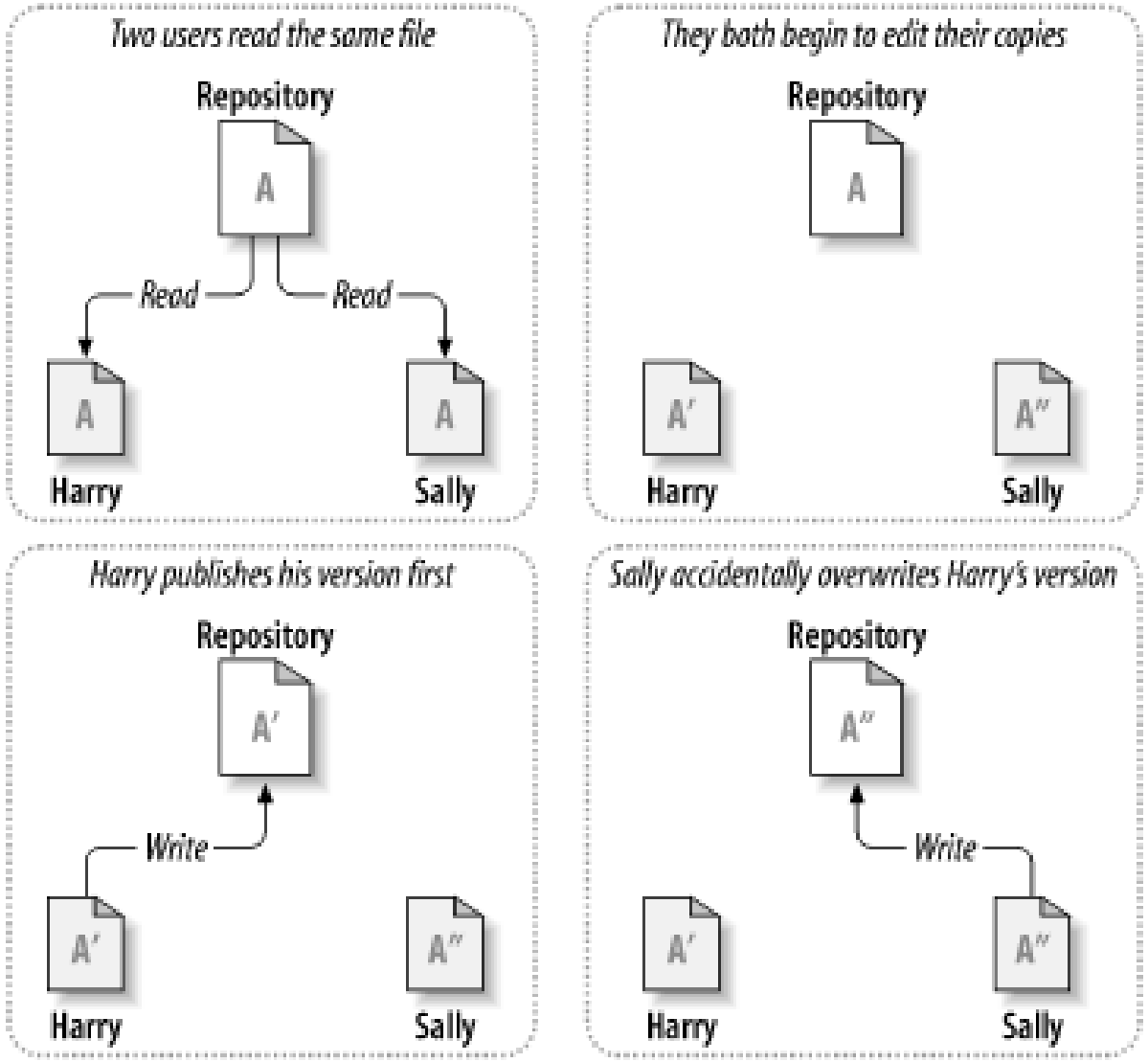
- 1972 – Source Code Control System (SCCS)
  - Store changes using deltas
  - Keeps multiple versions of a complete directory
  - Keeps original documents and changes from one version to the next
- 1982 – Revision Control System (RCS)
  - Keeps the current version and applies changes to go back to older versions
  - Single file at a time
- 1986 – Concurrent Versions Systems (CVS)
  - Start as scripts on top of the RCS
  - Handle multiple files at a time
  - Client-Server architecture

Three Generations of Version Control

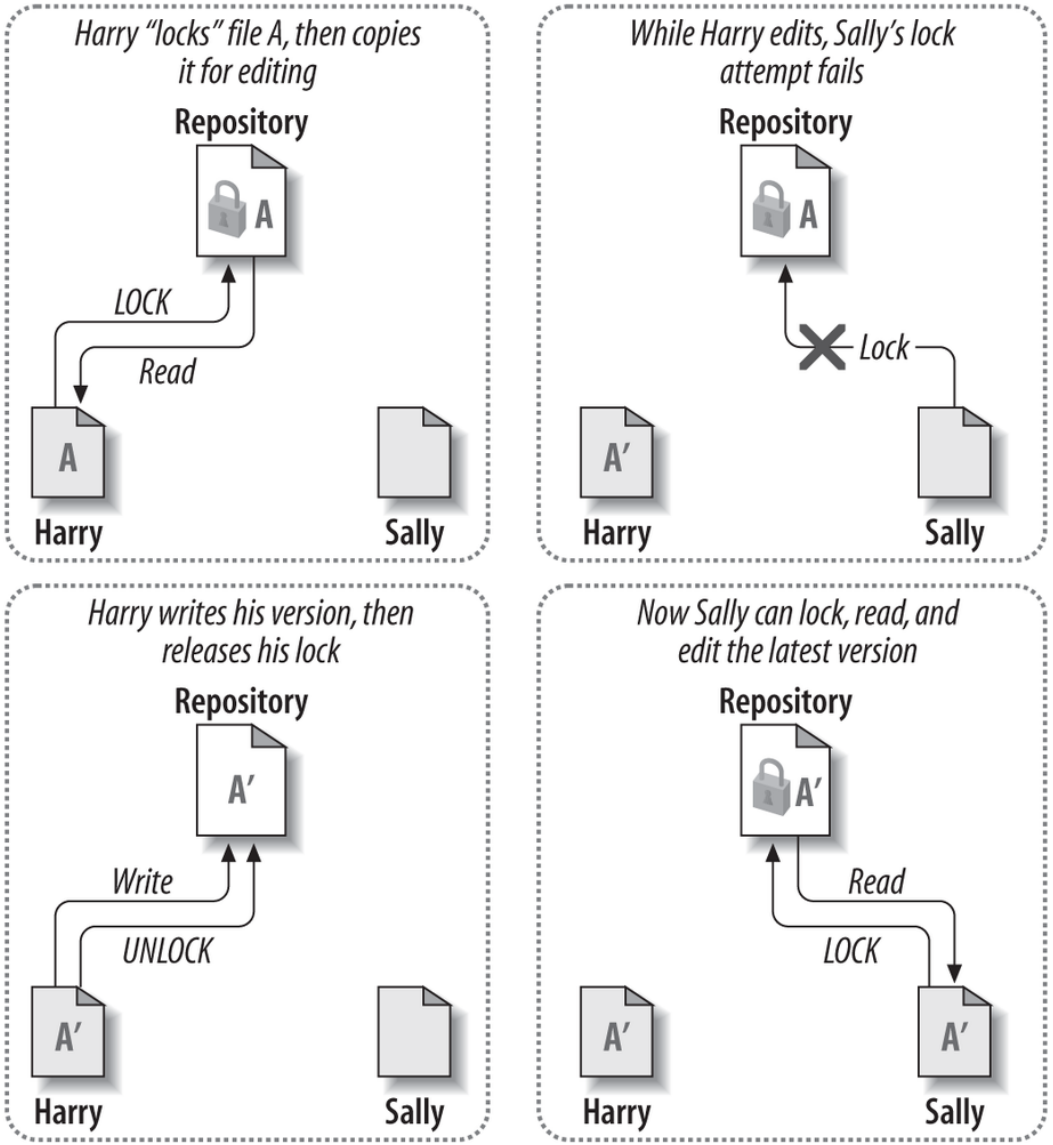
- **First Generation:**
  - One file at a time
  - Lock: only one person could be working on a file at a time.
  - Example: RCS, SCCS



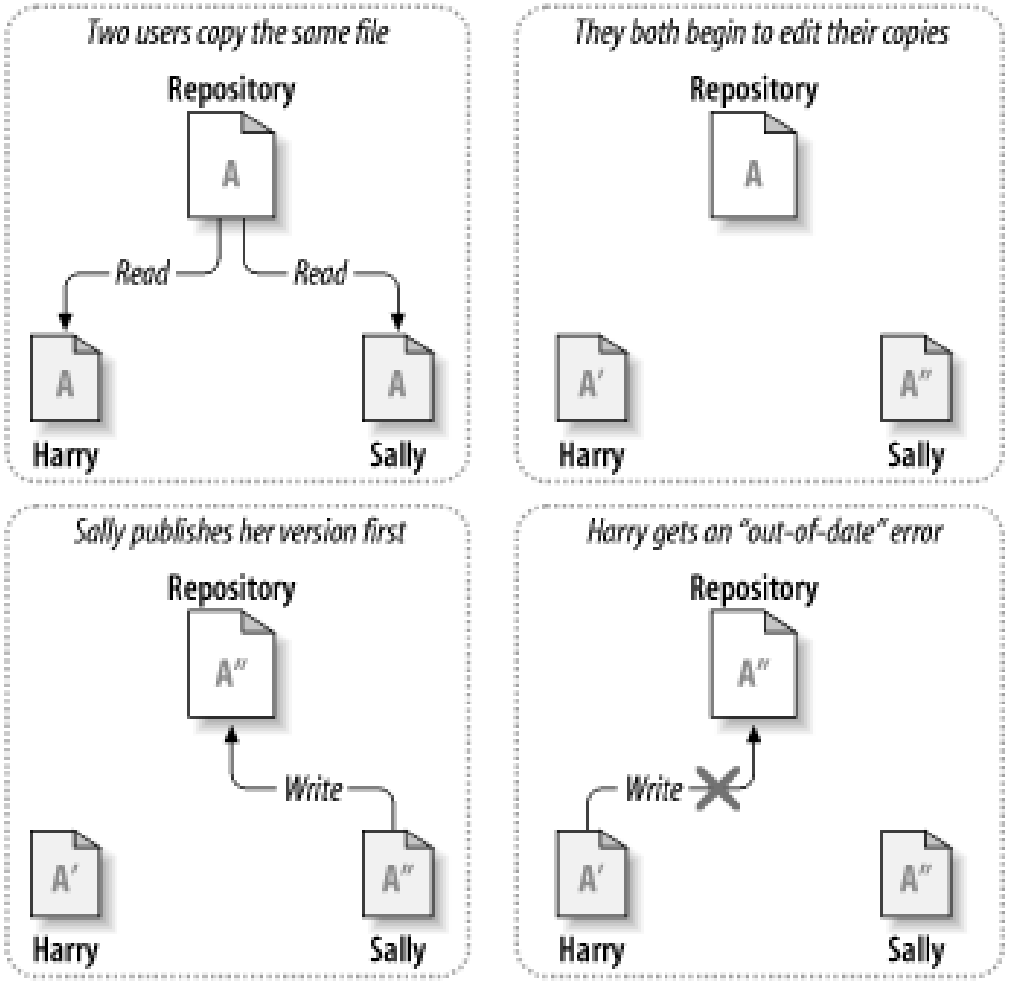
First Generation: The Problem to Avoid



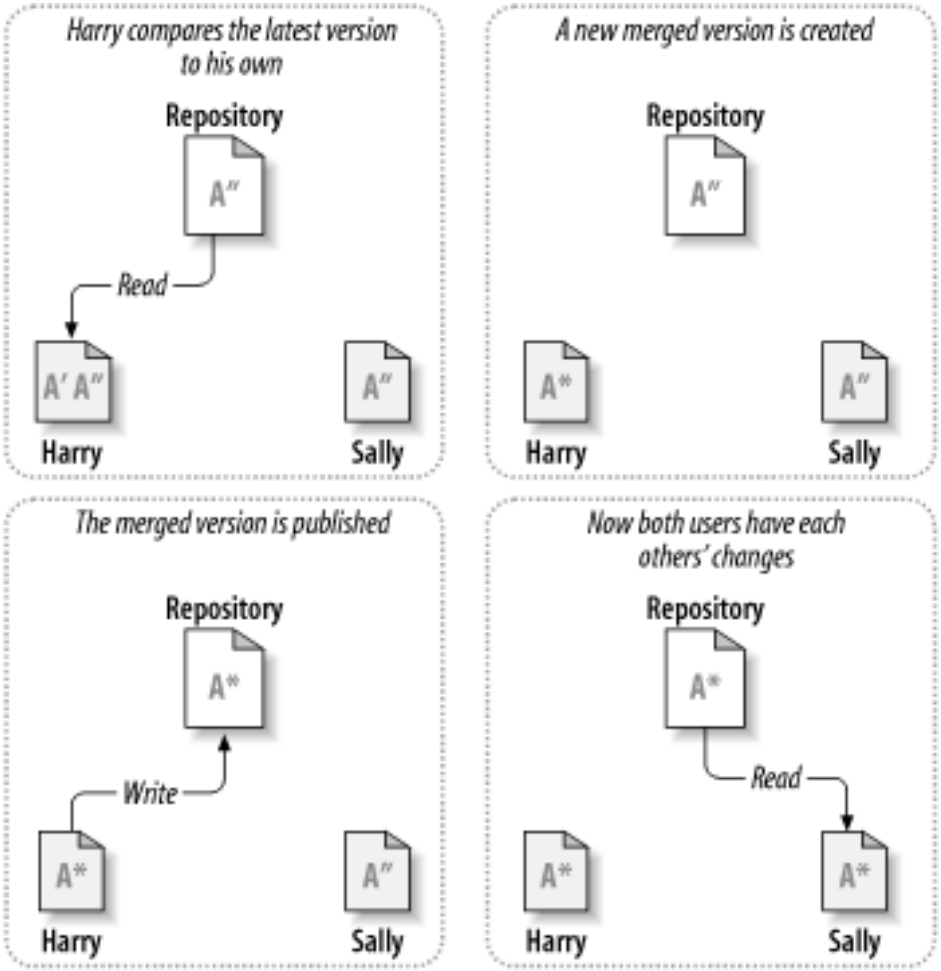
First Generation: *lock-modify-unlock*



First Generation: The Copy-Modify-Merge Solution 1

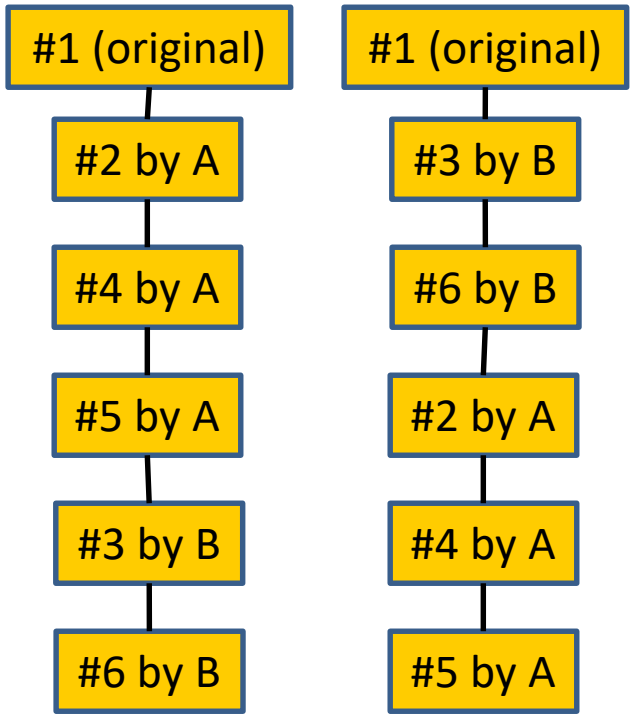


First Generation: The Copy-Modify-Merge Solution 2



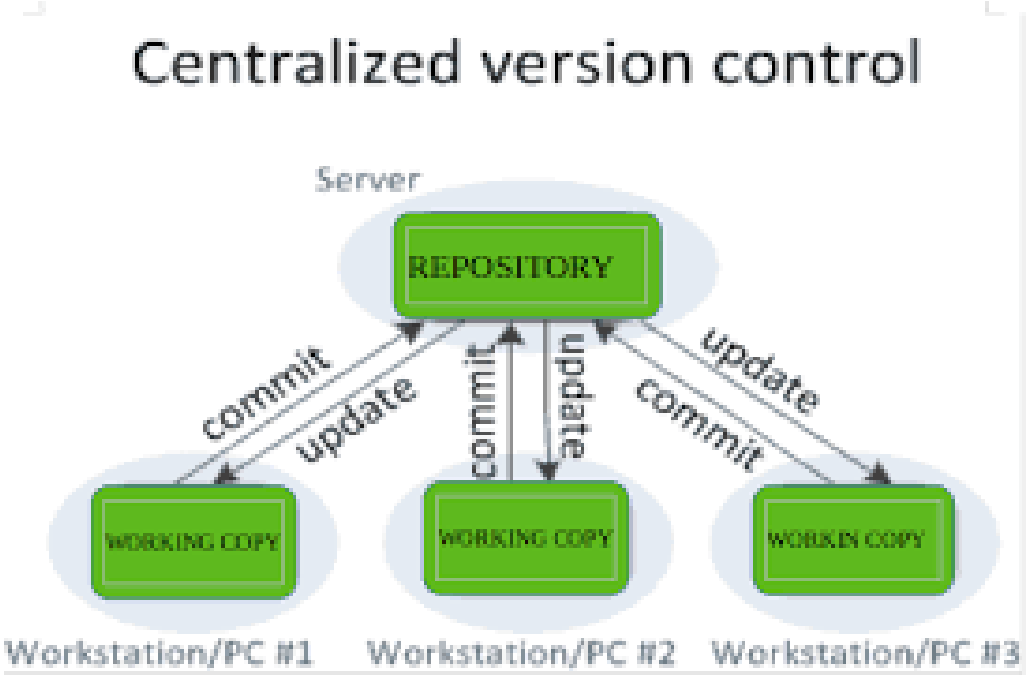
Three Generations of Version Control

- **Second Generation:**
  - Centralization: one central shared repository
  - MultiFile
  - Merge before commit
  - Rewrites history
  - Example:
    - CVS,
    - SourceSafe,
    - Subversion,
    - Team Foundation Server



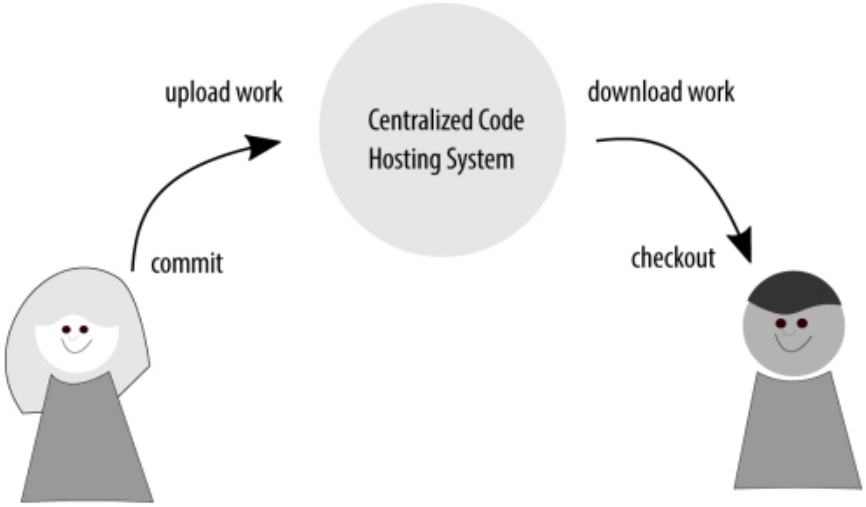
Centralized version control

- **Centralized version control systems** are based on the idea that there is **a single "central" copy** of your project on a single server and programmers "commit" their changes to this central copy.



Centralized version control

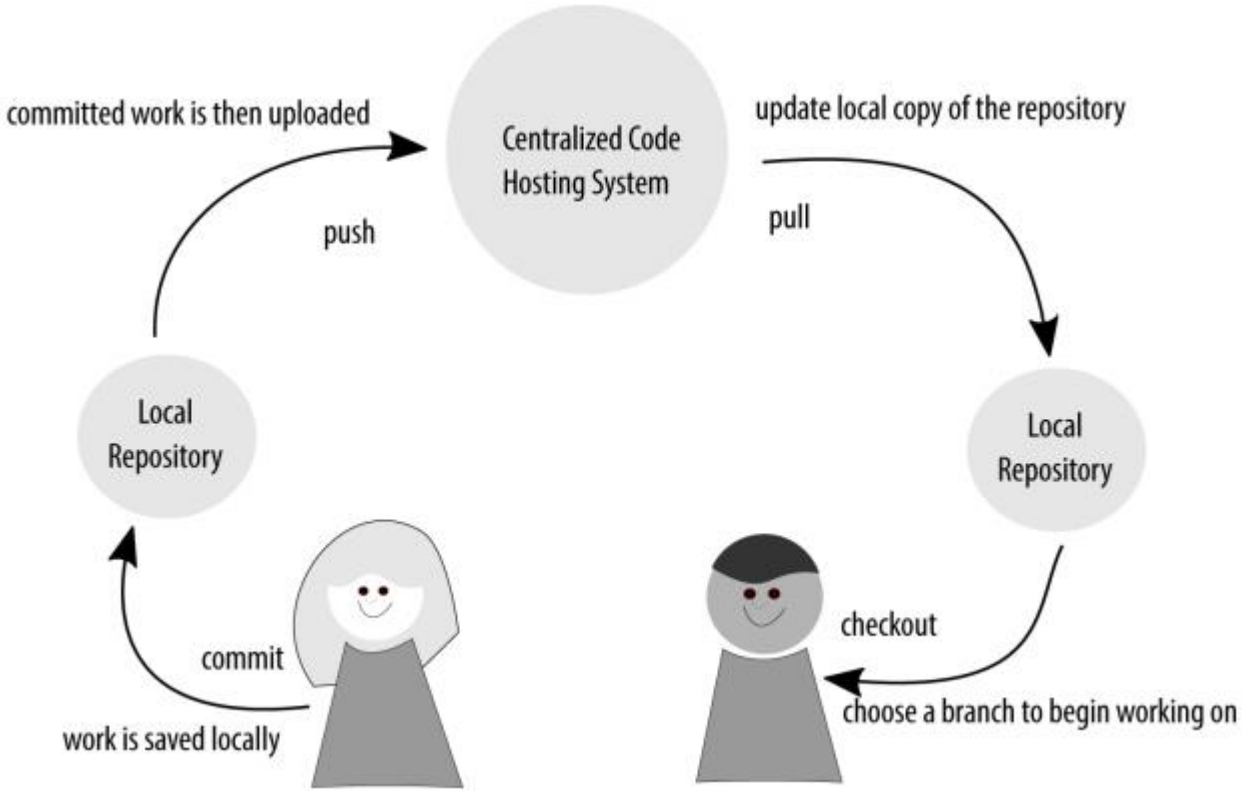
- Just when you thought you were ready to share your work, or request a code review, **you would sometimes be prevented from doing so if someone else had recently updated the same branch with their own work.**



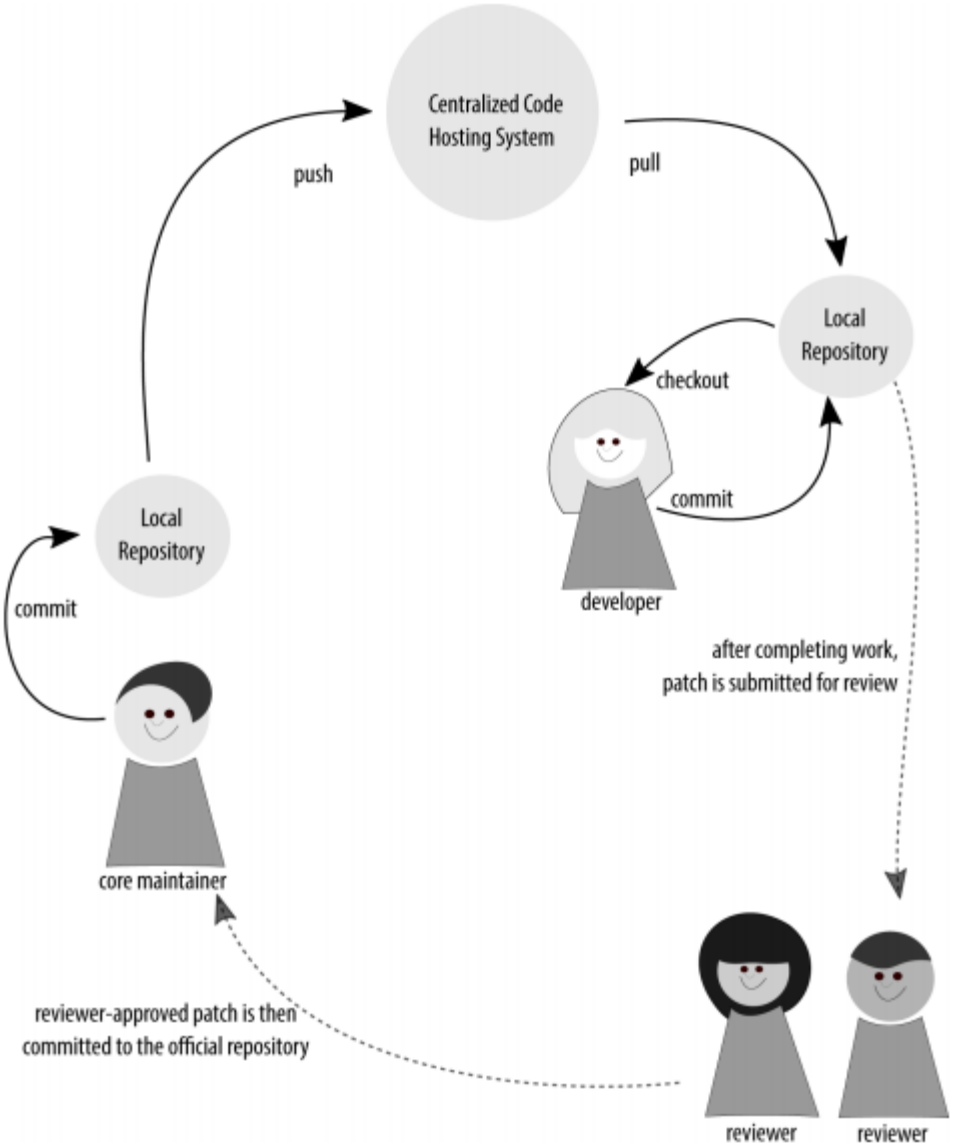
- Working with files in Subversion

Centralized version control

Centralized is simple, and what you'd first invent:
a single place everyone can check in and check out.



The community review process for patches



Centralized version control

- This model works **for backup, undo and synchronization** but **isn't great for merging and branching changes** people make.

- As projects grow, you want to split features into chunks, developing and testing in isolation and **slowly merging changes into the main line.**

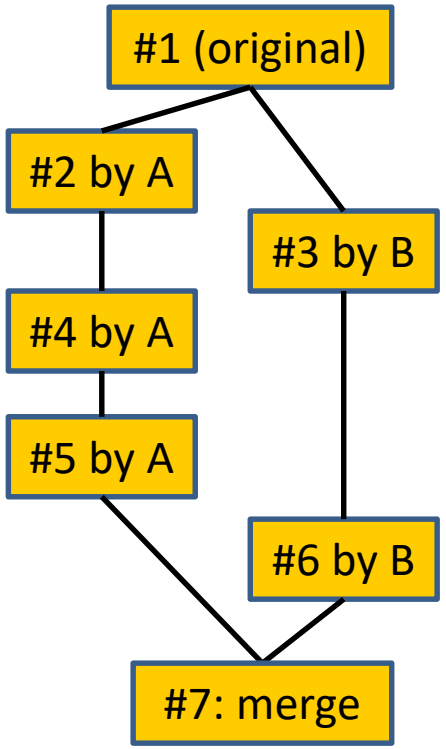
- **Recording/Downloading** and **applying** a change are separate steps

- Centralized version control focuses on **synchronizing, tracking, and backing up files.**

Version Control Examples

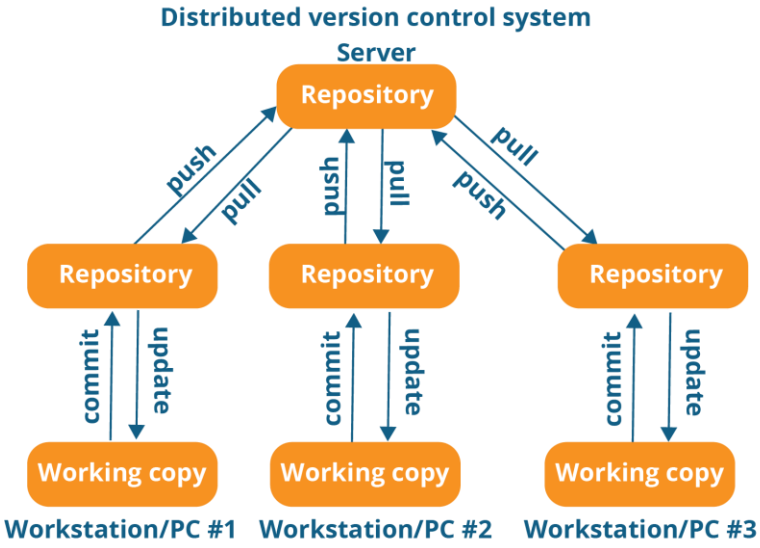
Three Generations of Version Control

- **Third Generation**:
  - Changesets
  - Commit before merge
  - Example: Bazaar, Git, Mercurial



Three Generations of Version Control

- Repository - including the **whole history** - is held on each developer's own machine
- **New revisions are checked in to local repository**
- Local repository can be synchronised with other remote repositories to send your changes to other people

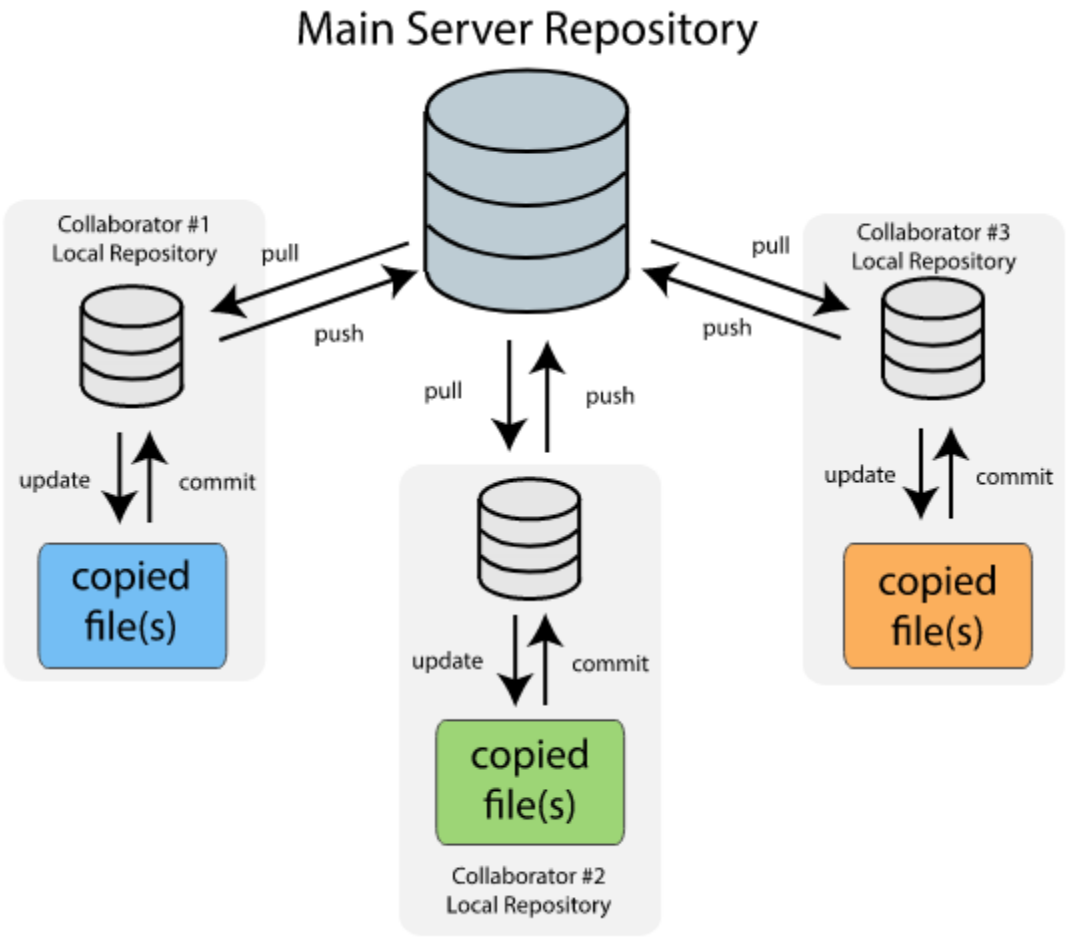


Advantages of DVCS

- Most operations done on the local repository
  - Much faster
  - Possible offline
- Each developer has the full history of the project
  - If anyone loses theirs - they just need to clone someone else's copy
  - Changes can easily be sent to different remote repositories

E.g. always send to "development" repository, only send to "production" repository when code is tested

Distributed Version Control Systems



Distributed Version Control Systems use a peer-to-peer approach to version control, which allows users to work productively when not connected to a network

Additional

- Understanding Version-Control Systems Eric Raymond
  - http://www.catb.org/~esr/writings/version-control/version-control.html
- A brief history of version control
  - https://www.red-gate.com/blog/database-devops/history-of-version-control

Why use version control

- A version control system (VCS) is a tool for **managing a collection of program code** that provides you with three important capabilities:
  - ***Reversibility (grįžtamumas/atgaminimas),***
  - ***Concurrency*** **(išlygiagretinimas)**
  - ***annotation*** **(anotavimas)**

Why use version control: **reverssibilty**

- The ability to **back up to a saved**, known-good state when you discover that some modification you did was a mistake or a bad idea.

Why use version control: **concurrency**

- The ability to have many people modifying the same collection of code or documents knowing that **conflicting modifications can be detected and resolved.**

Why use version control: **annotate**

- The capability to *annotate* your data, **attaching explanatory comments** about the intention behind each change to it and a record of who was responsible for each change.
  - Even for a **programmer working solo**, change histories are an important **aid to memory**;
  - for a **multi-person project** they become a vitally important **form of communication among developers**.

Why is version control important?

- **Version control** allows us to:
  -- Keep everything of importance in one place
  – Manage changes made by the team
  – Track changes in the code and other items
  – Avoid conflicting changes
  – Revert to a previous state of data

Benefits of Version Control System

- Individual benefits
  - Backups with tracking changes
  - Tagging – marking the particular version in time
  - Branching – multiple versions
  - Tracking changes
  - Revert (undo) changes

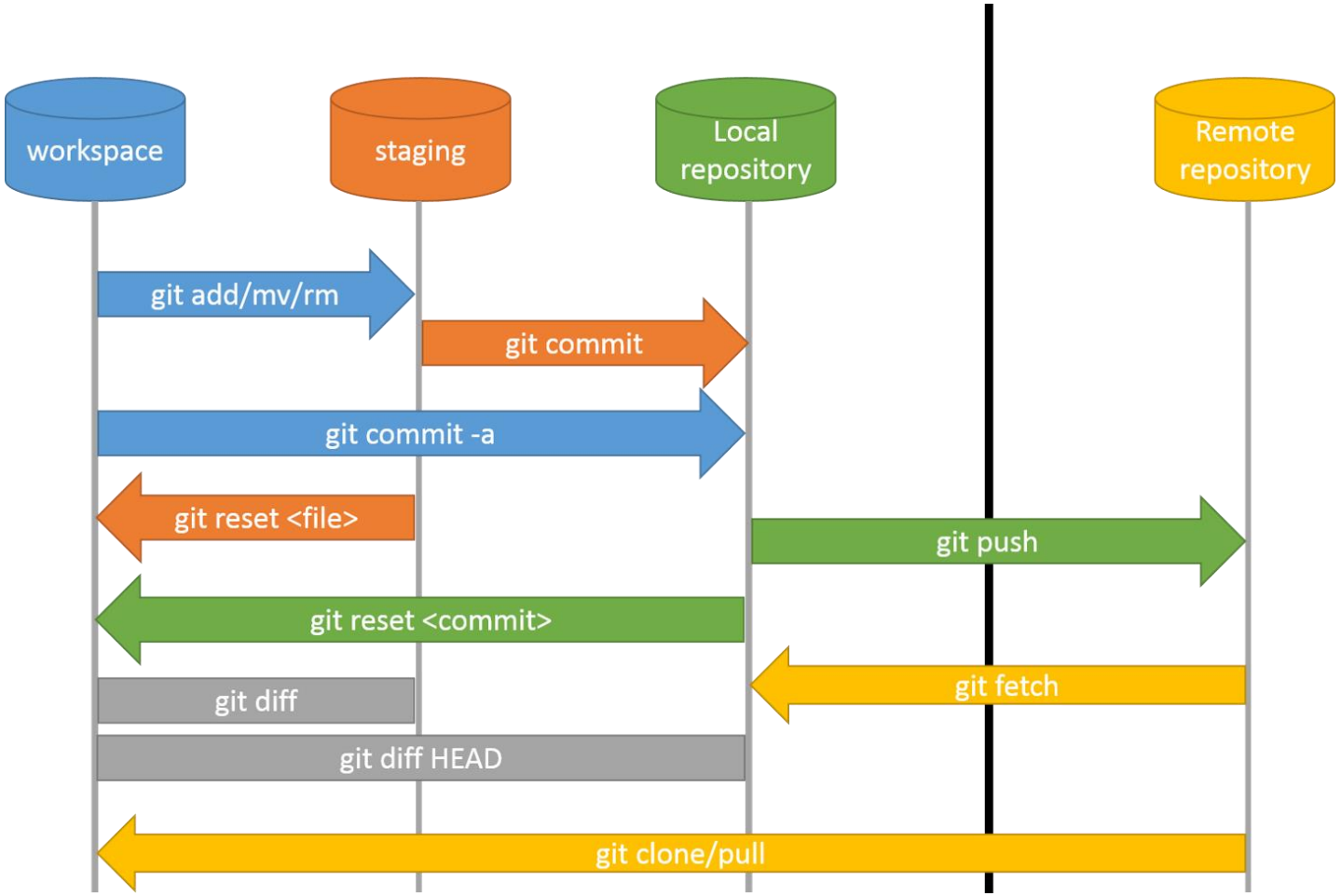
Benefits of Version Control System

- Team benefits
  - Working on the same code sources in a team of several developers
  - Merging concurrent changes
  - Support for conflicts resolution when the same file (the same part of the file) has been simultaneously changed by several developers
  - Determine the author and time of the changes

Version Control Systems

https://www.tekkiwebsolutions.com/blog/github-vs-bitbucket-vs-gitlab/

GIT processes

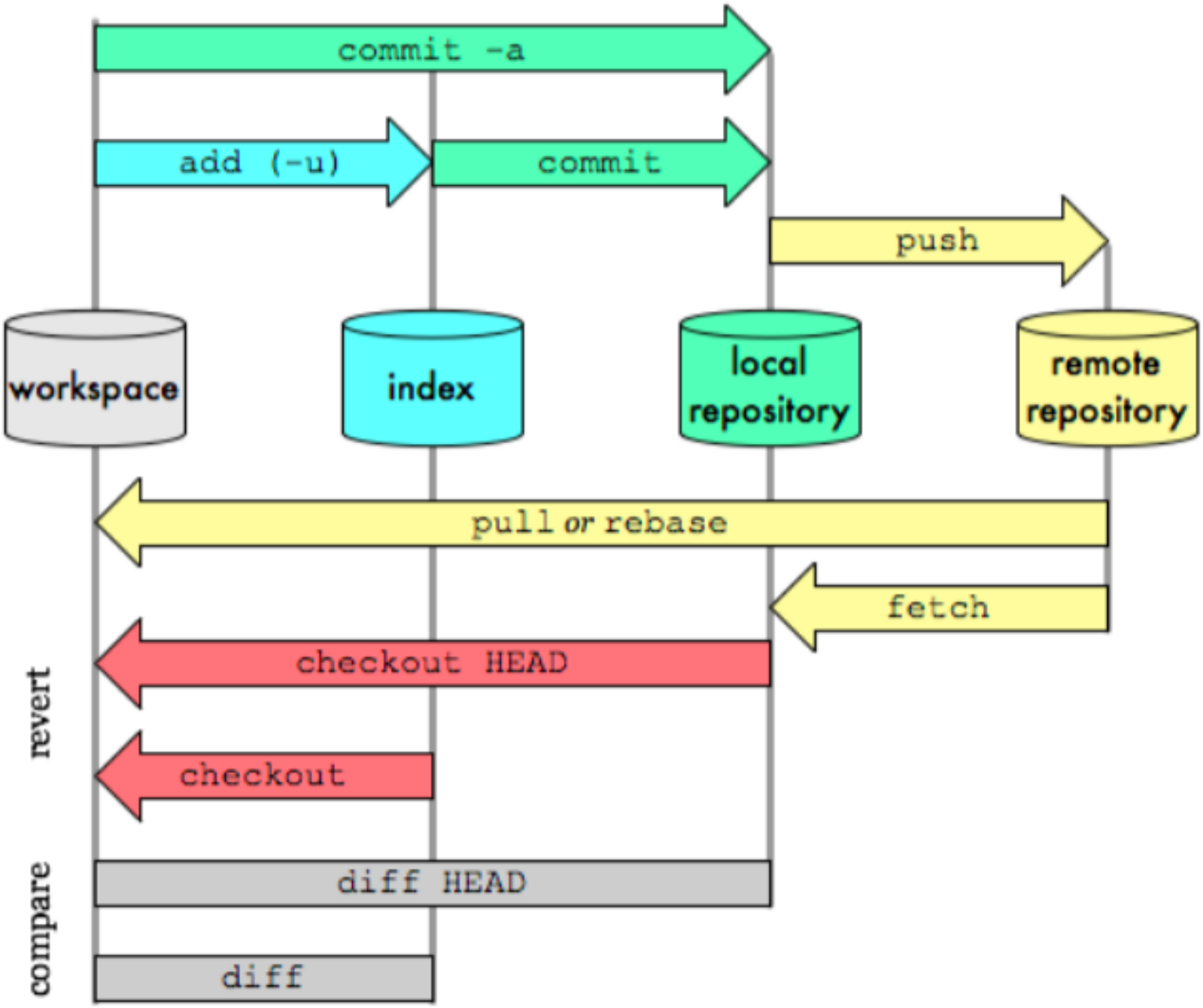


Main terminology I

- **Repository (repo)**: a database containing the history
- **Server**: The computer storing the repo.
- **Client**: The computer connecting to the repo.
- **Working Set/Working Copy**: your local directory of files/ current state of the data, where you on
- **Revision / commit** (noun)- the state of data at a given time
- **Check out** - to retrieve a revision from the repository
- **Check in / commit** (verb) - to send a revision to the repository changes.

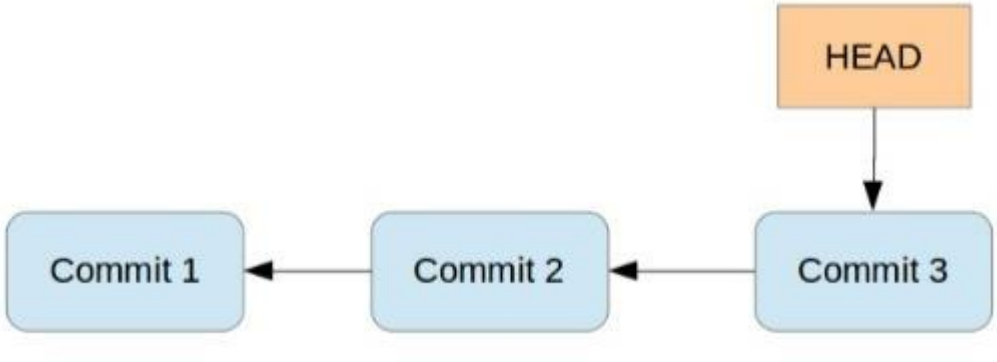
Main terminology II

- **Staging area** is a place to record things before committing.
  - Physically it's the .git/index file that makes up the content of the Staging Area
- **Master**: Default name for initial branch. When a new repository is initialized with $ git init, a branch named master is automatically created.
- **Branch:** Lightweight moveable pointer referencing one particular *commit*.
- **Hash (SHA-ID, "commit hash"):** Unique commit identifier, derived from the content and metadata provided at time of commit, using the cryptographic SHA
- **Working Tree (**Work Space**):** The actual files and folders on disk currently checked out for editing, e.g. what you see in your editor or file explorer. It also contains metadata about any changes made, that are not yet staged or ignored, and can be shown using $ git status.


commit -a
add (-u)
commit
push
workspace
index
local repository
remote repository
pull or rebase
fetch
checkout HEAD
checkout
diff HEAD
diff
revert
compare

Commit/Revision

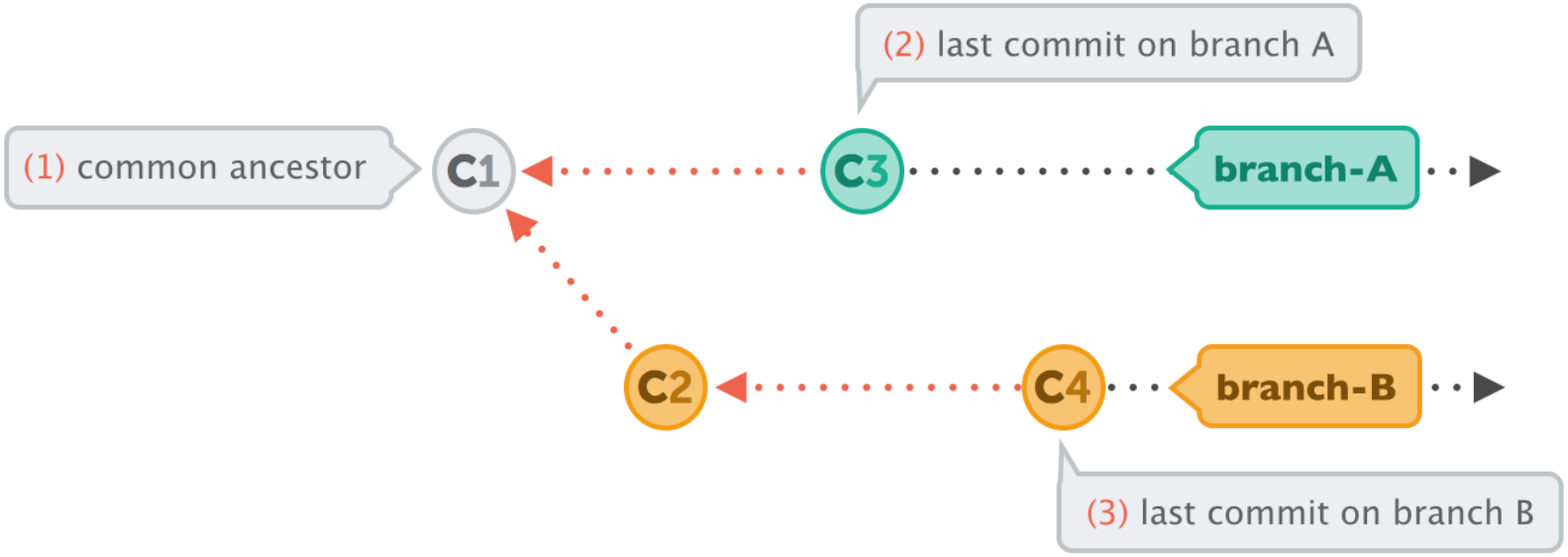
- When you commit your changes into a repository this creates a new *commit object* in the Git repository.
  - This *commit object* uniquely identifies a new revision of the content of the repository.
- Every revision has: a revision ID/number; message; author; time
- Commits in Git are **immutable snapshots**
  - ***Immutable object:*** *In object-oriented and functional programming, an immutable object is an object whose state cannot be modified after it is created. This is in contrast to a mutable object, which can be modified after it is created.*
- Every revision (except the first) has at **least one parent** revision that it builds upon.



Commit/Revision

- Commit structure, where child commits reference parent commits, is technically a directed acyclic graph (DAG) and is what makes up the repository history.
- The DAG is referred to as a history graph or commit-graph.
- SHA algorithm produces a 40 digit long (20 byte) hash value, also known as message digest. In Git, a commit can be referenced using its entire hash

```
commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date: Sat Mar 15 16:40:33 2008 -0700
```



Add/Restore

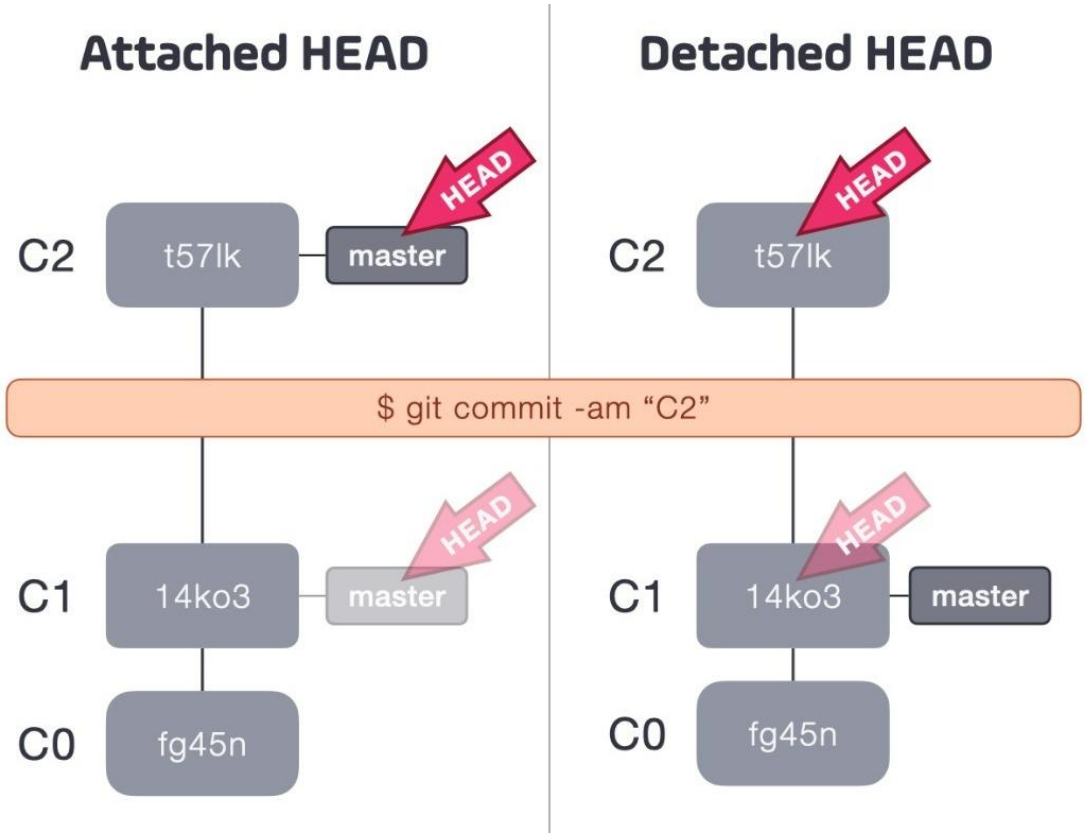
- **Add:** Moves files from *Working Tree* to ***Staging Area.***
  - Updates **the index** using the current content found in the working tree, to prepare the content staged for the next commit.
  - `git add`
- **Restores** changes made to files in *Staging Area* or *Working Tree*.
  - When used for files in *Working Tree* it resets them to their initial state based on *HEAD*, discarding any changes made.
  - https://git-scm.com/book/en/v2/Git-Tools-Reset-Demystified

HEAD

- *HEAD* is a symbolic reference most often pointing to the currently checked-out branch or commit
  - **Where am I right now in the repository?**
  - it's Git's way of knowing on which commit to mirror your local Working Tree on, and whether you're currently working on a branch (attached) or not (detached).
- Default state is attached, where any manipulation to the history is **automatically recorded to the branch HEAD is currently referencing.**
- Technically HEAD is a plain text reference, but unlike branches and lightweight tags that are stored in **.git/refs**, it is instead stored inside: **.git/HEAD**

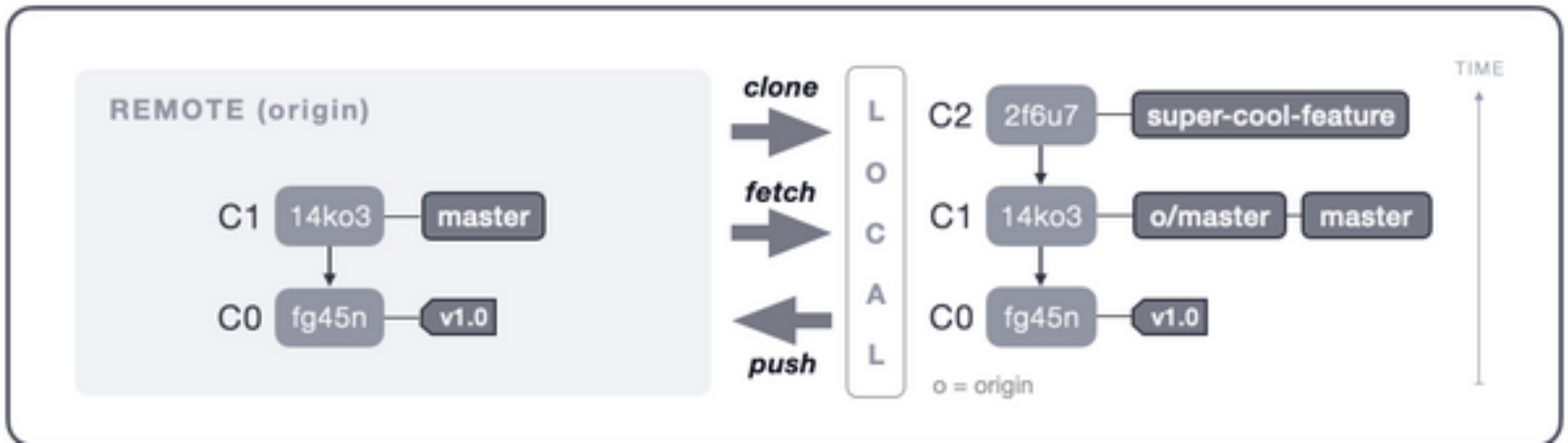
HEAD

- The main difference is that **in the attached state the change is automatically recorded in the master branch**.
- In the detached state, **the change did not impact any existing branch and master** remained to reference C1.



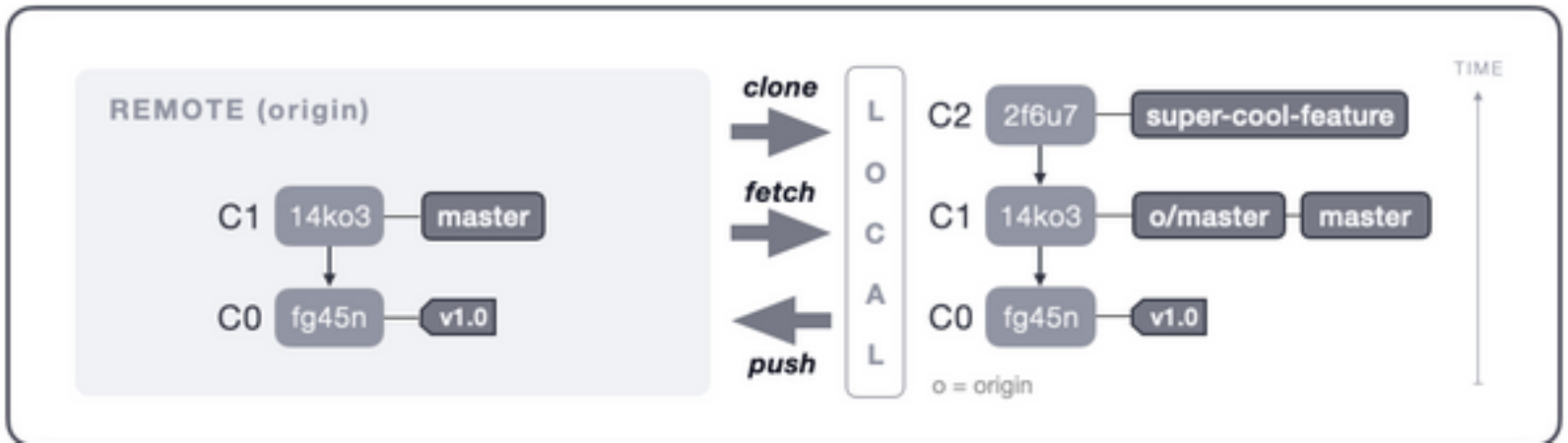
Clone/Fetch

- **Clone:** Copies an entire remote repository down to your local machine, setting up a cloned version and checks out the default branch (generally master); this action is done only once.
- **Fetch:** Updates remote references in your *cloned* local repository.
  - if a developer has pushed changes to a remote branch, those changes will be pulled down to your repository whenever *fetch* is performed. Note: *fetch* won't automatically merge any changes, only update references.



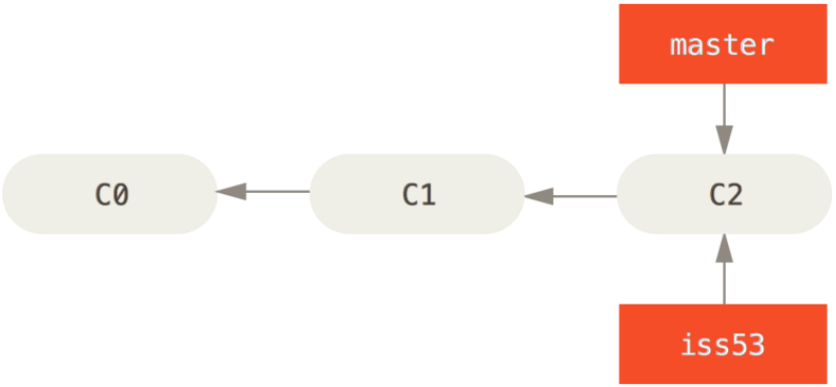
Push

- **Push:** Makes your local changes publicly available in a remote repository
  - Updates remote refs using local refs, while sending objects necessary to complete the given refs.
  - `git push <repo name> <branch name>`



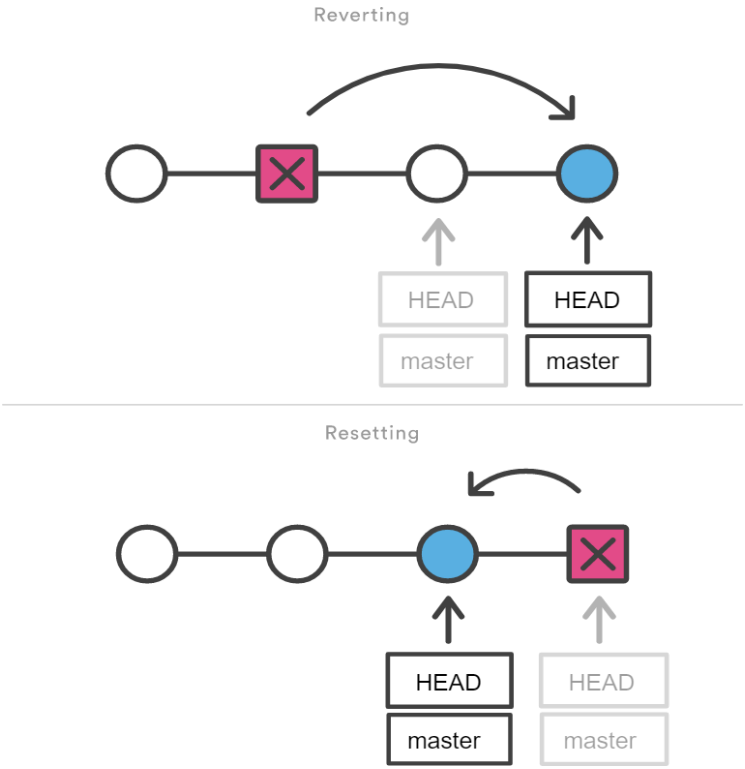
Checkout

- **Check out-** download a file from the repo.
- hecks out a particular version of one or multiple files from history into your *Working Tree*
- `git checkout iss53`



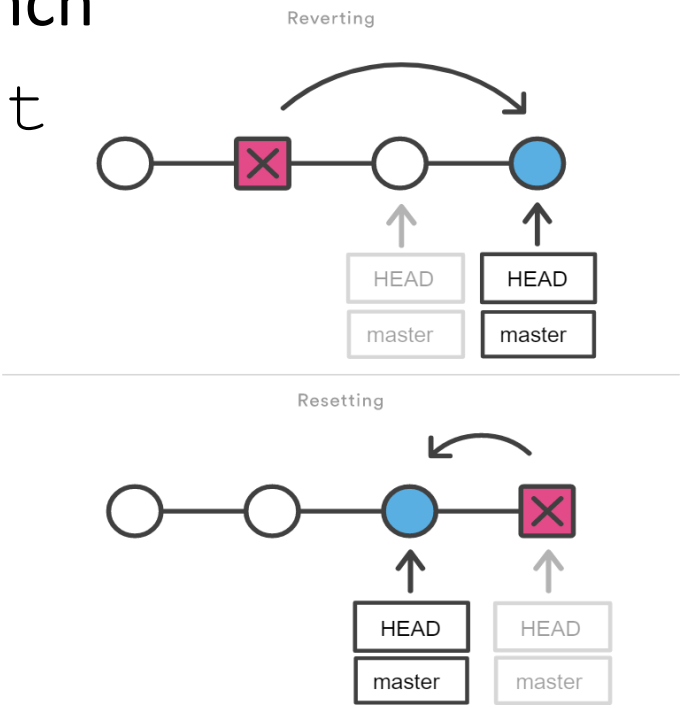
Checkout/revert

- If you don’t like your changes and want to start over, you can **revert** to the previous version and start again (or stop).
- Reverting undoes a commit by creating a new commit. This is a safe way to undo changes, as it has no chance of re-writing the commit history.



Checkout/reset

- Resetting is a way to move the tip of a branch to a different commit.
  - This can be used to remove commits from the current branch
  - `git reset`



Git commands

```
git init    # initialize new repository
git add     # add files or stage file(s)
git commit  # commit staged file(s)
git commit -m <title> -m <description>
git status  # see what is going on
git log     # see history
git diff    # show unstaged/uncommitted
       modifications
git show    # show the change for a specific
      commit
git mv      # move tracked files
git rm      # remove tracked files
```

Conventional Commits

- Commitizen is a tool designed for teams
- Template:

```
<type>[optional scope]: <description>

[optional body]

[optional footer(s)]
```

Conventional Commits

- feat – a new feature is introduced with the changes
- fix – a bug fix has occurred
- chore – changes that do not relate to a fix or feature and don't modify src or test files (for example updating dependencies)
- refactor – refactored code that neither fixes a bug nor adds a feature
- docs – updates to documentation such as a the README or other markdown files
- style – changes that do not affect the meaning of the code, likely related to code formatting such as white-space, missing semi-colons, and so on.
- test – including new or correcting previous tests
- perf – performance improvements
- ci – continuous integration related
- build – changes that affect the build system or external dependencies
- revert – reverts a previous commit

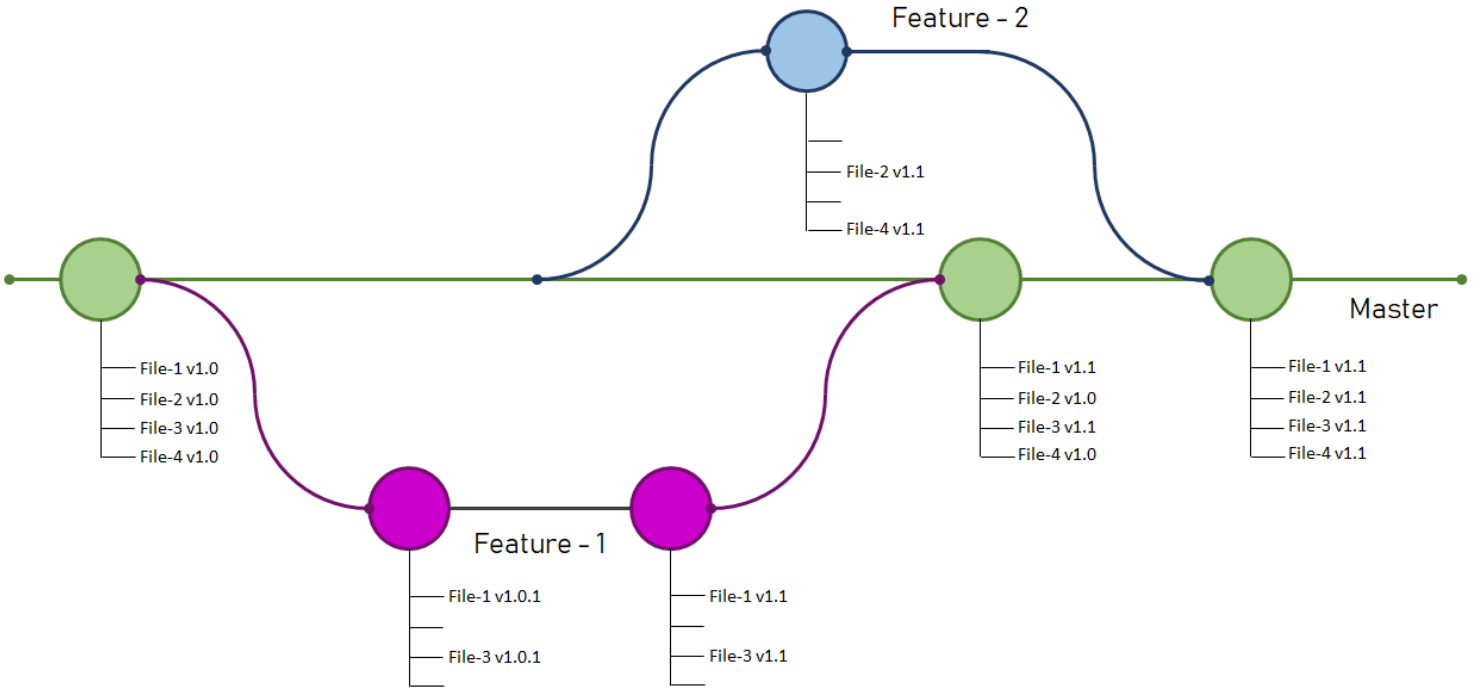
Conventional Commits

- Good examples of commit
  - feat: improve performance with lazy load implementation for images
  - Fix bug preventing users from submitting the subscribe form
  - Update incorrect client phone number within footer body per client request

- Bad examples of commit
  - fixed bug on landing page
  - Changed style
  - I think I fixed it this time?
  - empty commit messages

Branch

- The branch is a lightweight movable **pointer referencing a specific commit** in the repository's commit history.
- They **allow multiple developers to work on the same codebase** simultaneously without interfering with each other's work



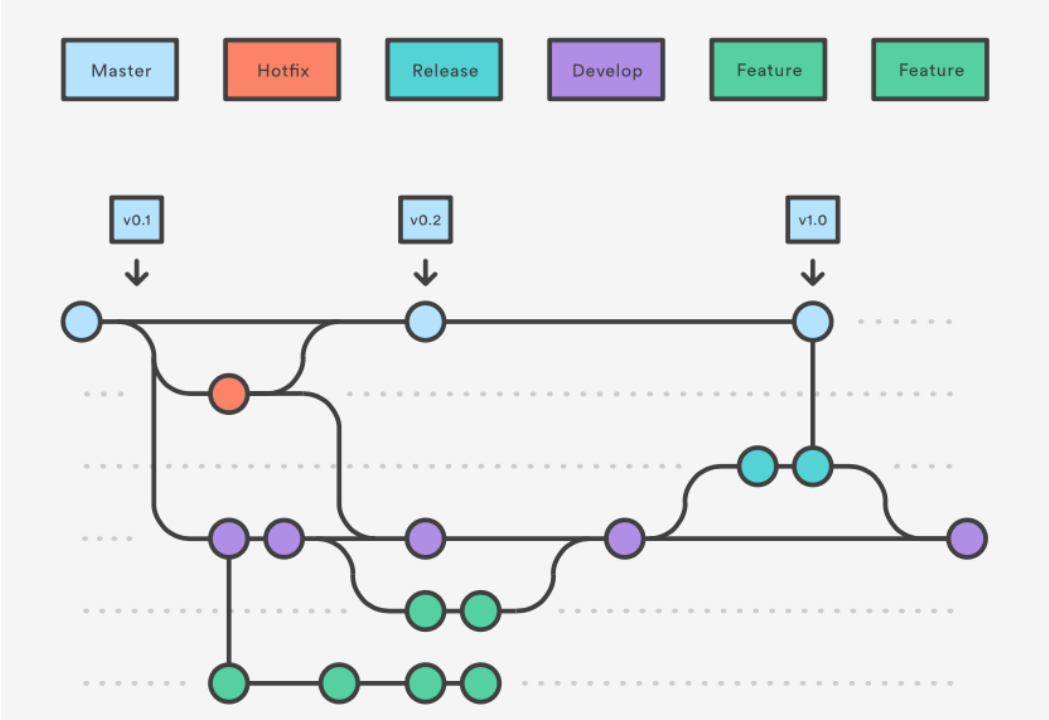
Branch

- Branches enable you to have multiple parallel versions of data
- Separate development of unrelated features
- Generally the main branch (“master”) should have the latest tested & stable version of the software
- Operations with branch:
  - Create
  - Delete
  - Shows a list
  - Rename
  - Checkout
  - Visualise
  - ...
- Git mantra: **“branch early; branch often”**

Types of Branches

Create GitFlow:

- Production branch: master
- Develop branch: develop
- Feature prefix: feature/
- Release prefix: release/
- Hotfix prefix: hotfix/
- Depending on the project



Types of Branches

Smaller or solo projects:

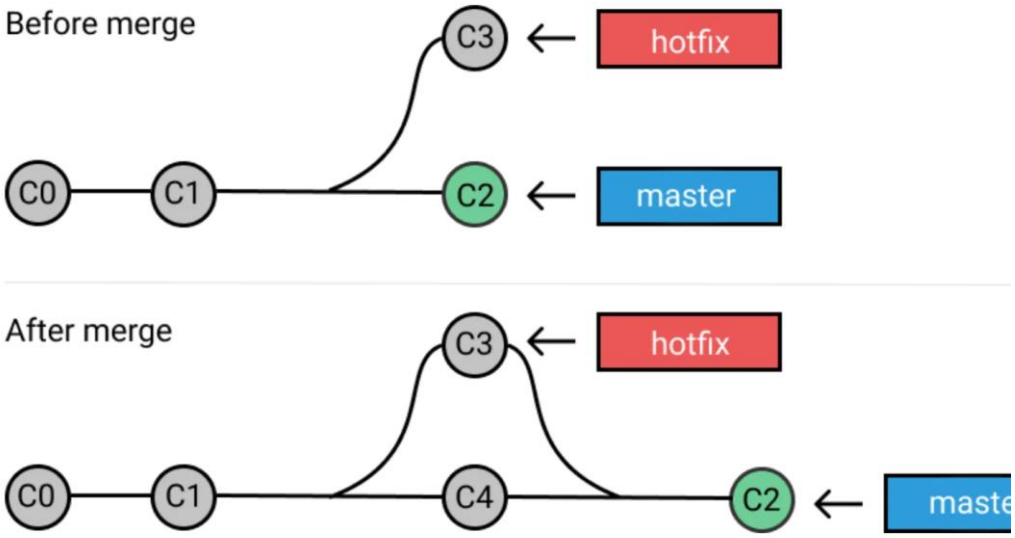
- Doing most of work in master
- Creating branches for bigger tasks
- Naming branches as seems appropriate

Companies or bigger open-source projects usually have special guidelines:

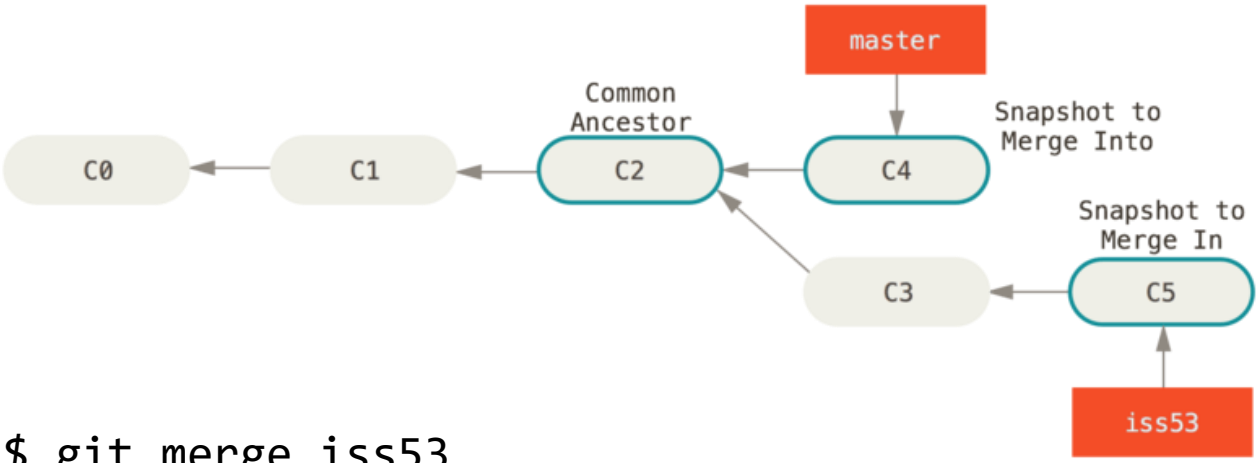
- How to name branches
- How often to create branches
- How to merge branches into master

Merge

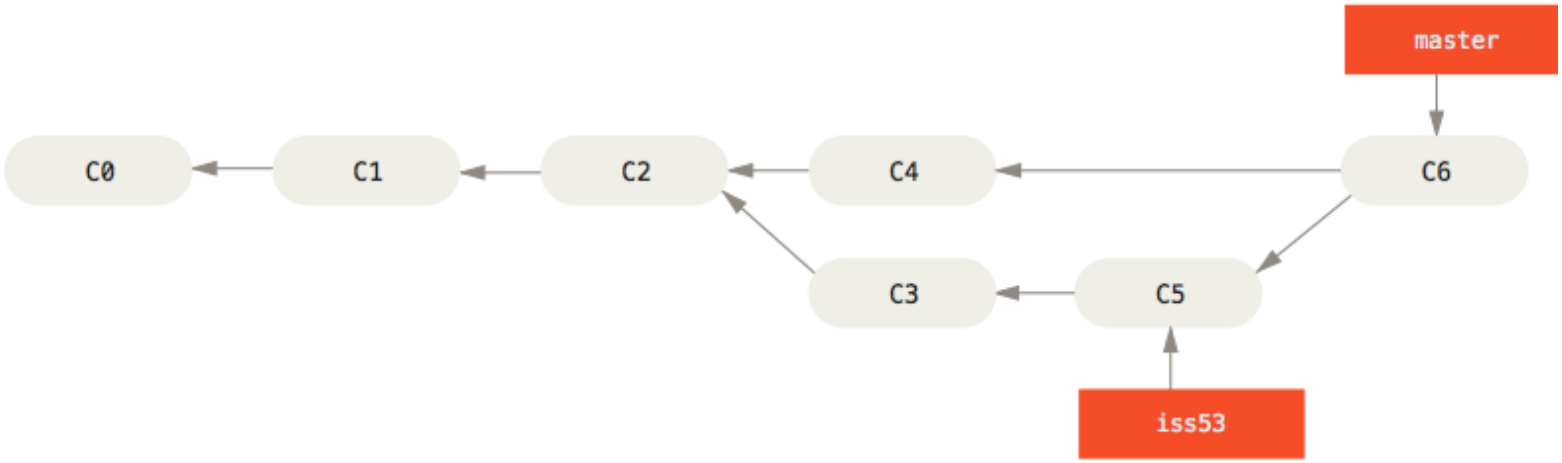
- Merge - where one branch connects with others, one revision has 2+ parents
- Git merges most changes automatically



Merging Code




```
$ git merge iss53
Merge made by the 'recursive'
strategy.
```



Branches merge

Types of merge strategies :

- Recursive (Fast Forward)
- Ours
- Octopus
- Resolve
- Subtree

https://www.geeksforgeeks.org/merge-strategies-in-git/

Branches merge strategies: Recursive

- Recursive merges **are the default** for any merges that aren't fast-forward merges
- These types of merges operate **on two different heads** using a three-way merge algorithm.
- The merge commit ends up having two parents once the merge is complete.
- The recursive strategy can detect and handle **merges that involve renaming**, but **it can't make use of detected copies.**

Branches merge strategies: Recursive

- Recursive merges **are the default** for any merges that aren't fast-forward merges
- These types of merges operate **on two different heads** using a three-way merge algorithm.
- The merge commit ends up having two parents once the merge is complete.
- The recursive strategy can detect and handle **merges that involve renaming**, but **it can't make use of detected copies.**

Merge Conflicts

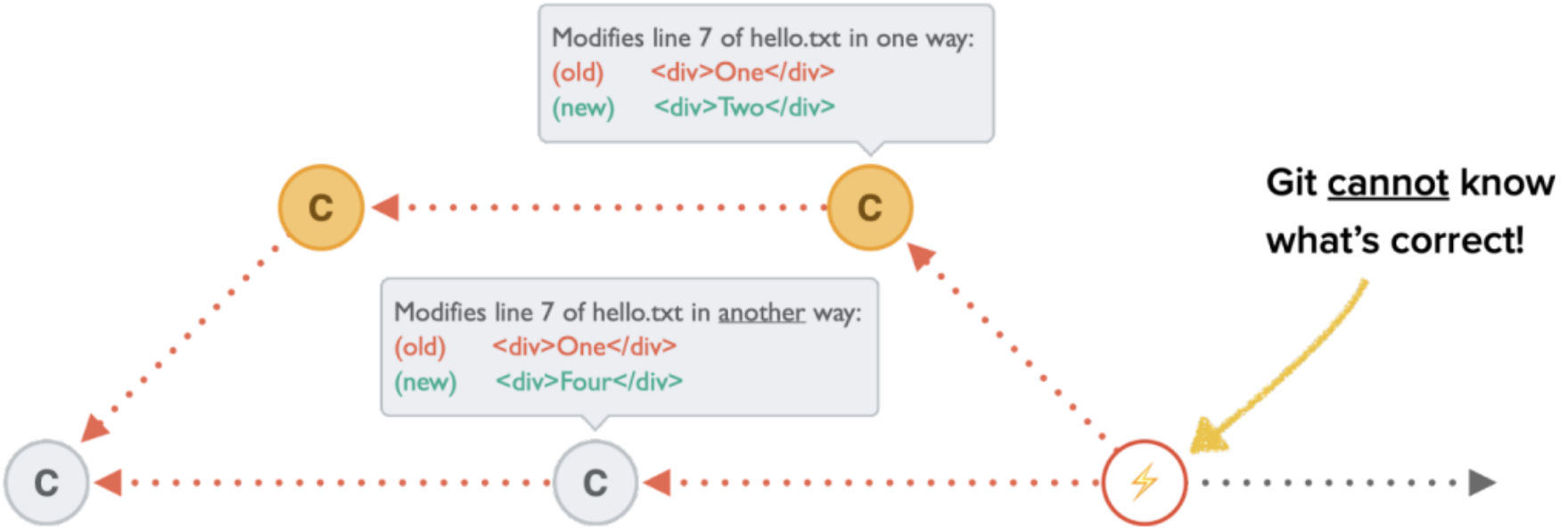
- “Merge conflict” - when Git is not sure how to combine changes in a file
- A **conflict** occurs when two **developers will change the same line of code in two different ways**, and the system is **unable to decide the changes.**

```
$ git merge new-branch
Auto-merging test.txt
CONFLICT (content): Merge conflict in test.txt
Automatic merge failed; fix conflicts and then commit the
result.
```

Merge Conflicts: when and how

- When more than one person **changes the same line in a file** and tries to merge the change to the same branch
- When a developer **deletes a file**, **but another developer edits it**, and they both try to merge their changes to the same branch.
- When a **developer deletes a line, but another developer edits it**, and they both try to merge their changes to the same branch
- When a developer choose and take only the most beneficial a commit, which is the act of picking a **commit from a branch and applying it to another**
- When a developer **is rebasing a branch**, which is the process of moving a sequence of commits to a base commit

Merge Conflicts




```
$ git status
On branch master
You have unmerged paths.
(fix conflicts and run "git commit")
(use "git merge --abort" to abort the merge)
Unmerged paths:
(use "git add <file>..." to mark resolution)
both modified: test.txt
no changes added to commit (use "git add" and/or "git commit -a")
```

Merge Conflicts: sequence

1. Resolving a merge conflict
2. Find conflicts in file.
3. Decide what is the desired final state.
4. Delete the conflict markers, leaving only the desired state.
5. Add the resolved files and commit.

Why is version control important?

- **Version control** allows us to:
  - Keep everything of importance in one place
  - Manage changes made by the team
  - Track changes in the code and other items
  - Avoid conflicting changes

VCS software

- Automates the heavy lifting behind version management
- Enables you to:
  - Track changes in data
  - View history of data
  - Revert to a previous state of data
- Not just for code

History of Git

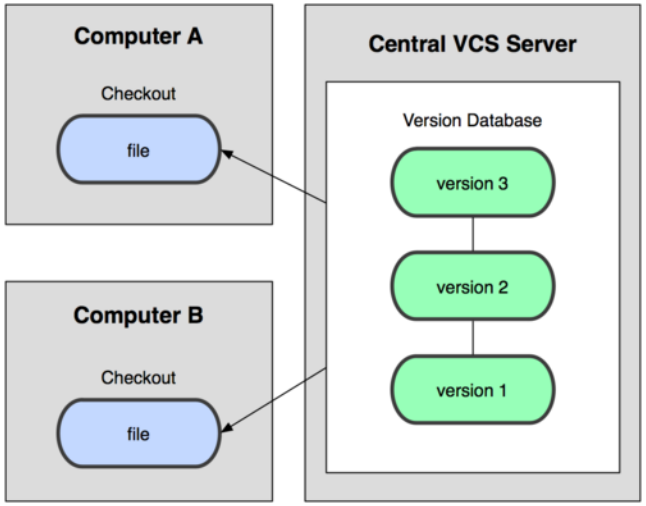
- Came out of Linux development community
- Linus Torvalds, 2005
- Initial goals:
  - Speed
  - Support for non-linear development (thousands of parallel branches)
  - Fully distributed
  - Able to handle large projects like Linux efficiently

Git – Properties and Features

- Local repository allows versioning without network connection
- Commit only adds the changes to the local repository therefore it is necessary to propagate the changes to the upstream using git push
- Files are stored as objects in a database (INDEX)
  - SHA1 fingerprints as file identifiers
- Support development a high usage of branches
- Support for applying path sets , e.g., delivered by e-mails
- Tags and Branches are marked points/states of the repository
- Suitability of the Git deployment depends on the project and model of the development

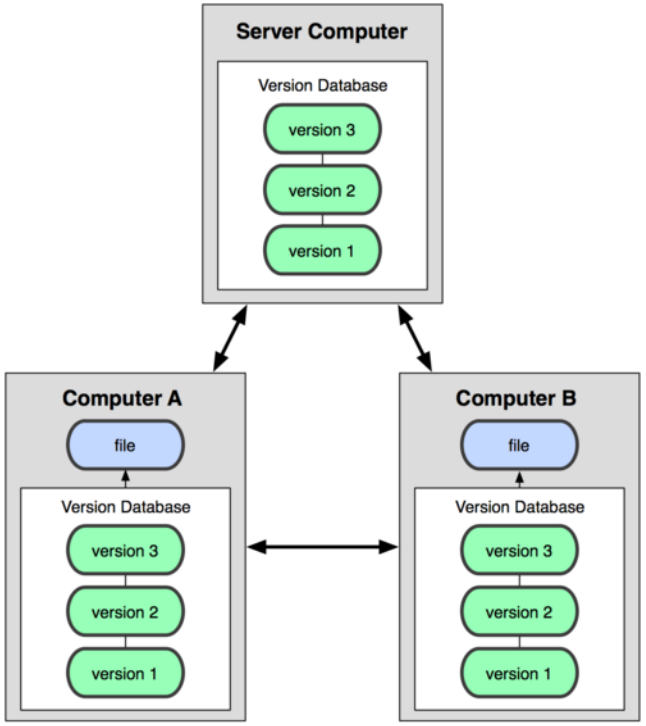
Git uses a distributed model

Centralized Model



(CVS, Subversion, Perforce)

Distributed Model



(**Git**, Mercurial)

Result: Many operations are local

VCS software: GitHub

- GitHub Documentation https://help.github.com and https://www.atlassian.com/git/tutorials/
- GitHub is the most popular open source code repository site

VCS software: Bitbucket

- Bitbucket is a web-based hosting service for projects that use either the Git or Mercurial revision control systems
- Bitbucket https://bitbucket.org
- Bitbucket Documentation
  - https://confluence.atlassian.com/display/BITBUCKET/Bitbucket+Documentation+Home

Additional

- Subversion (SVN) - http://subversion.apache.org
- TortoiseSVN (Windows) - http://tortoisesvn.tigris.org
- Concurrent Version Systems (CVS) - http://savannah.nongnu.org/projects/cvs
- Git - http://git-scm.com
- TortoiseGit (Windows) - http://code.google.com/p/tortoisegit
- Mercurial - http://mercurial.selenic.com
- RabbitVCS (Linux) - http://www.rabbitvcs.org