Version Control with GIT: an introduction

Muzzamil LUQMAN (L3i) and Antoine FALAIZE (LaSIE)

23/11/2017
LaSIE Seminar
Université de La Rochelle

Version Control with GIT: an introduction

- Why Git?
- What is Git?
- How to use Git locally?
- Some Graphical User Interfaces to use Git humanly
- How to use Git remotely?
- Can we use Git at Université de La Rochelle?
- Python Package Index (PyPI)
    - What is it?
    - How to use it?
    - Can we use Git with it?

What is version control (VC)?

From http://en.wikipedia.org/wiki/Revision_control :

- Revision control [...] is the **management of changes** to documents, computer programs, large web sites, and other collections of information.
- Changes are usually **identified by a number or letter code**, termed the ”revision number” [...].
- For example, an initial set of files is ”revision 1”. When the first change is made, the resulting set is ”revision 2”, and so on.
- Each revision is **associated with a timestamp and the person making the Change**.
- Revisions can be compared, restored, and with some types of files, merged.

Example of VC systems:

- GIT (today’s topic)
- SVN
- Mercurial

A History of Version Control

Three Generations of Version Control

Generation	Networking	Operations	Concurrency	Examples
First	None	One file at a time	Locks	RCS, SCCS
Second	Centralized	Multi-file	Merge before commit	CVS, SourceSafe, Subversion, Team Foundation Server
Third	Distributed	Changesets	Commit before merge	Bazaar, Git, Mercurial

The forty year history of version control tools shows a steady movement toward more concurrency.

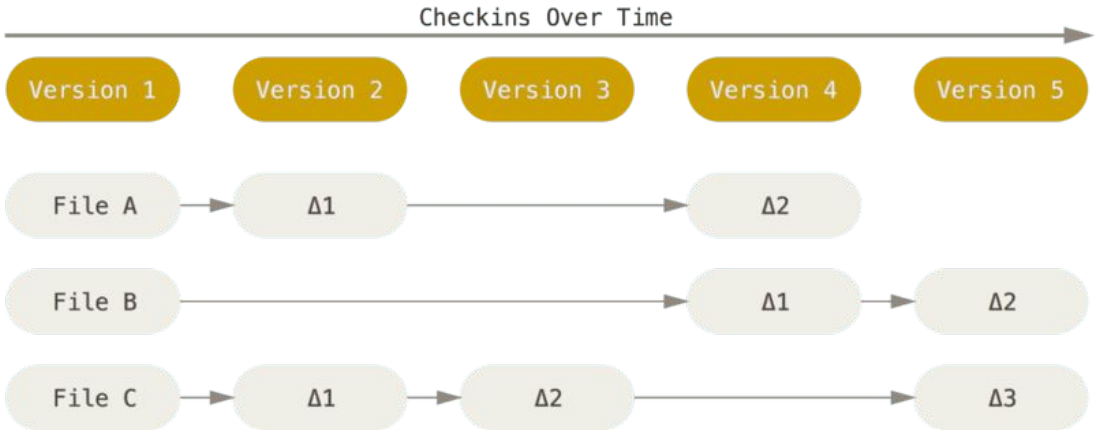
http://ericsink.com/vcbe/html/history_of_version_control.html
https://www.lynda.com/ALMTFS-tutorials/history-version-control/106788/115979-4.html

As you learn Git, try to clear your mind of the things you may know about other VCSs, such as CVS, Subversion or Perforce — doing so will help you avoid subtle confusion when using the tool.

History of Git

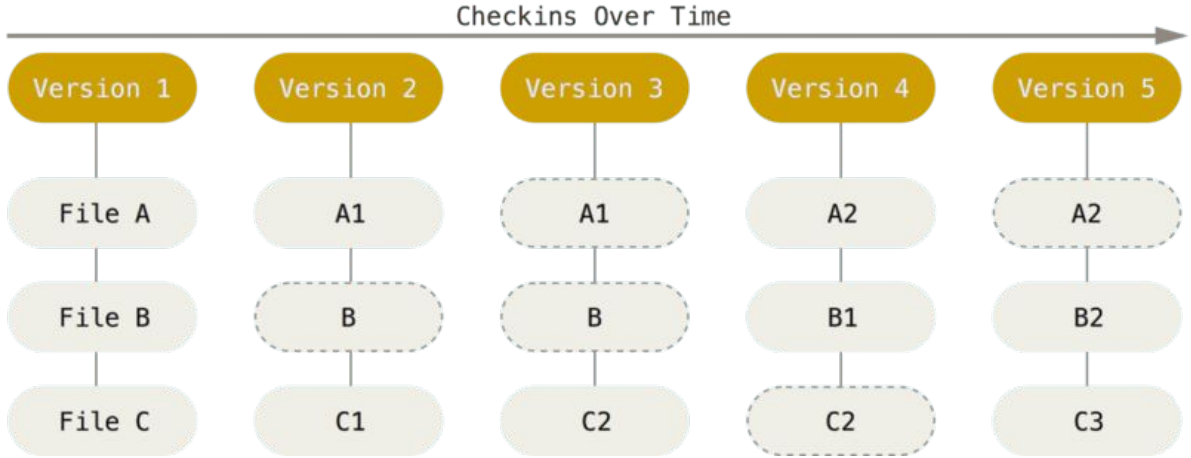
- Git is created by **Linus Torvalds**, the creator of Linux
- It was developed initially to manage the Linux development community
- Linux code has been managed,
  - 1991-2002: Using an archive of patches
  - 2002-2005: Using **BitKeeper**
  - 2005-Now: Using **Git**
- Target was,
  - Speed
  - Simple design
  - Fully distributed
  - Strong support for non-linear development (thousands of parallel branches)
  - Able to handle large projects like the Linux kernel efficiently

Snapshots, Not Differences



https://git-scm.com/book/en/v2/Getting-Started-Git-Basics

Snapshots, Not Differences

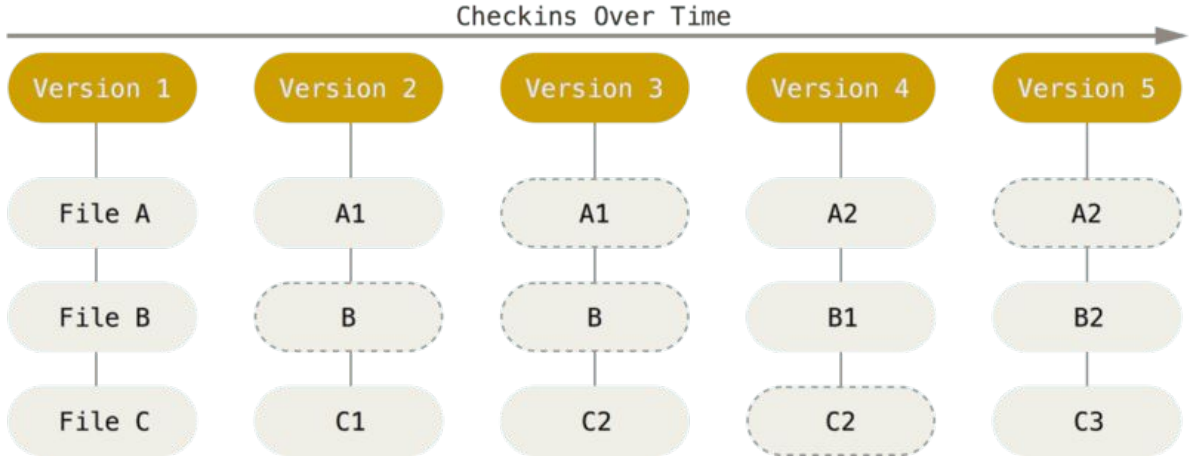


Nearly Every Operation Is Local

- Most operations in Git need only local files and resources to operate — generally no information is needed from another computer on your network.
- If you get on an airplane or a train and want to do a little work, you can commit happily (to your local copy, remember?) until you get to a network connection to upload.

https://git-scm.com/book/en/v2/Getting-Started-Git-Basics

Snapshots, Not Differences

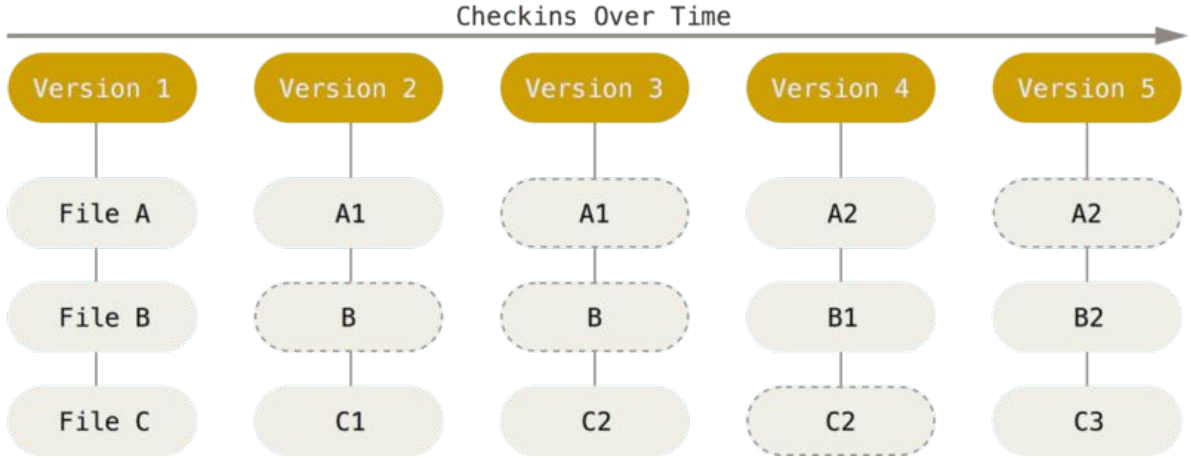


Git Has Integrity

- Everything in Git is check-summed before it is stored and is then referred to by that checksum. This means it's impossible to change the contents of any file or directory without Git knowing about it.
- You can't lose information in transit or get file corruption without Git being able to detect it.

https://git-scm.com/book/en/v2/Getting-Started-Git-Basics

Snapshots, Not Differences



Git Generally Only Adds Data

- When you do actions in Git, nearly all of them only add data to the Git database. It is hard to get the system to do anything that is not undoable or to make it erase data in any way.

https://git-scm.com/book/en/v2/Getting-Started-Git-Basics

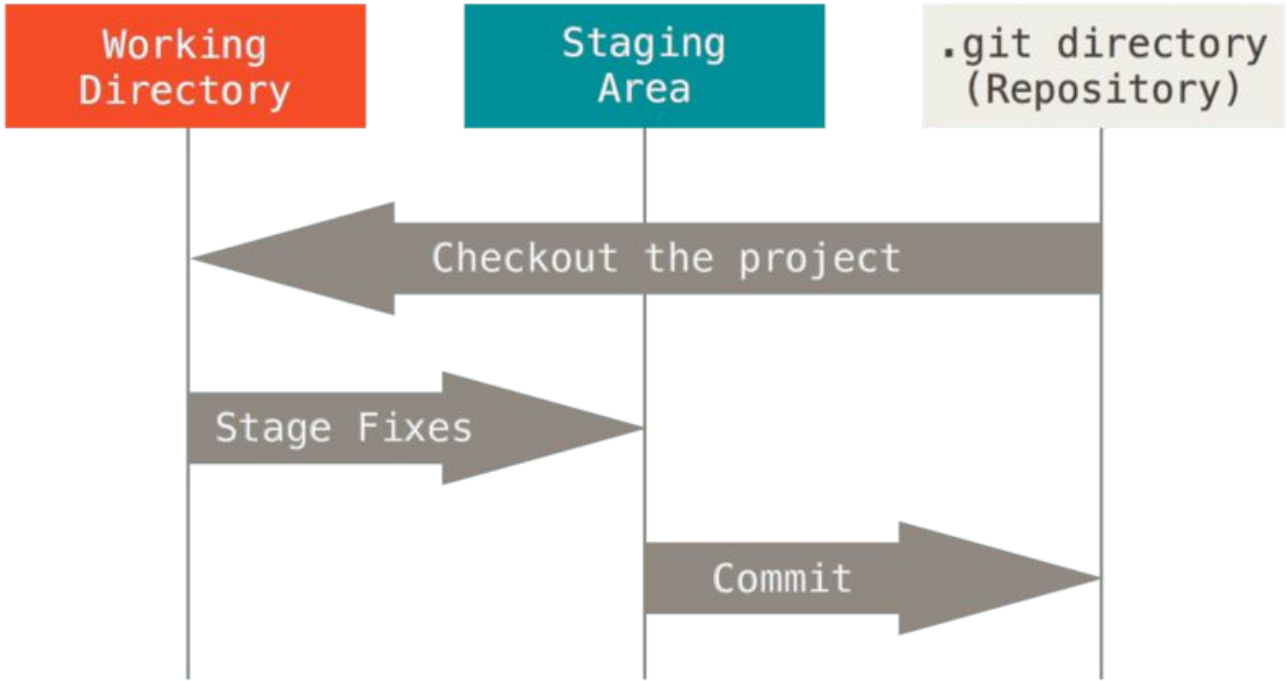
The Three States

Pay attention now — here is the main thing to remember about Git if you want the rest of your learning process to go smoothly.

Git has three main states that your files can reside in: committed, modified, and staged:

- Committed means that the data is safely stored in your local database.
- Modified means that you have changed the file but have not committed it to your database yet.
- Staged means that you have marked a modified file in its current version to go into your next commit snapshot.

The Three States



Example 1: local history

- The life of your software/article is recorded from the beginning: at any moment you can **revert to a previous revision**
- the history is browseable, you can **inspect any revision**
  - when was it done ?
  - who wrote it ?
  - what was change ?
  - why ?
  - in which context ?
- all the **deleted content remains accessible** in the history

Example 2: collaborative development

VC tools help you to:

- **Share** a collection of files with your team
- Merge changes done by other users
- Ensure that **nothing is accidentally overwritten**
- ~~Know who you must blame when something is broken~~

VC tools help working with third-party contributors:

- it gives them visibility of **what is happening** in the project
- it helps them to **submit changes** (patches) and it helps you to integrate these patches
- forking the development of a software and merging it back into mainline

Taxonomy of VC systems

Architecture:

- centralised → everyone works on the same unique repository
- decentralised (GIT) → everyone works on his own repository

Concurrency model:

- lock before edit → mutual exclusion
- merge after edit (GIT)→may have conflicts (GIT)

History layout:

- tree → merges are not recorded
- direct acyclic graph (GIT)

Atomicity scope: file Vs whole tree (GIT)

Other technical aspects of VC systems

Space efficiency:

- Storing the whole history of a project requires storage space (**storing every revision of every file**).
  - Do not commit datasets nor useless material (code that is not working)
- Most VC tools use delta compression to optimise the space (except Git which uses object packing instead)

Access method: A repository is identified with a URL. VC tools offer multiple ways of interacting with remote repositories.

- dedicated protocol (svn:// git://)
- direct access to a local repository (file://path or just path)
- direct access over SSH (ssh:// git+ssh:// svn+ssh://)
- over http (http:// https://)

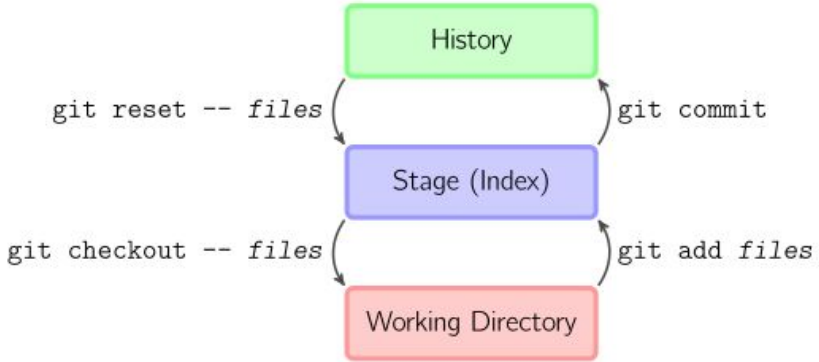
Part 1: Local usage

Create a new repository

```
git init myrepo
```

- This command creates the directory **`myrepo`** in the local directory.
- the repository (history) is located in **`myrepo/.git`**
- the (initially empty) working copy is located in **`myrepo/`**
- **`WARNING:`** The **`/.git/`** directory contains the whole history (**do not delete**)

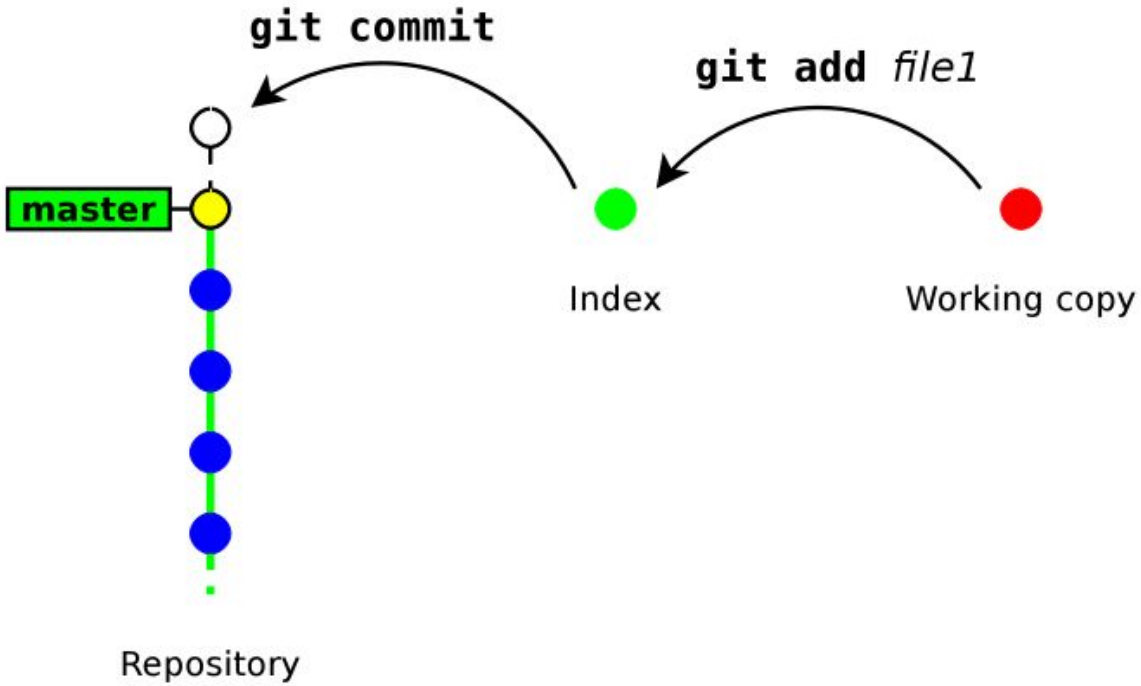
The different Areas: **working**, **stage** and **history**



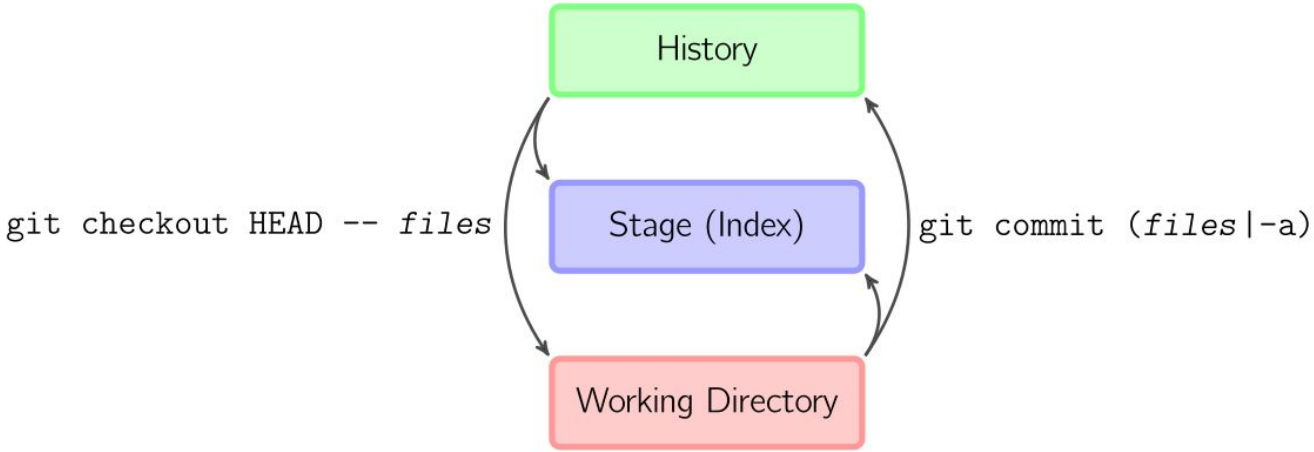
The commands above copy files between the working directory, the stage (also called the index), and the history (in the form of commits).

- **`git add 'files'`** copies files (at their current state) to the stage.
- **git commit** saves a snapshot of the stage as a commit.
- **`git reset -- 'files'`** unstages files ("undo" a **`git add files`**). You can also **`git reset`** to unstage everything.
- **`git checkout -- 'files'`** copies files from the stage to the working directory. Use this to throw away local changes.

Visually

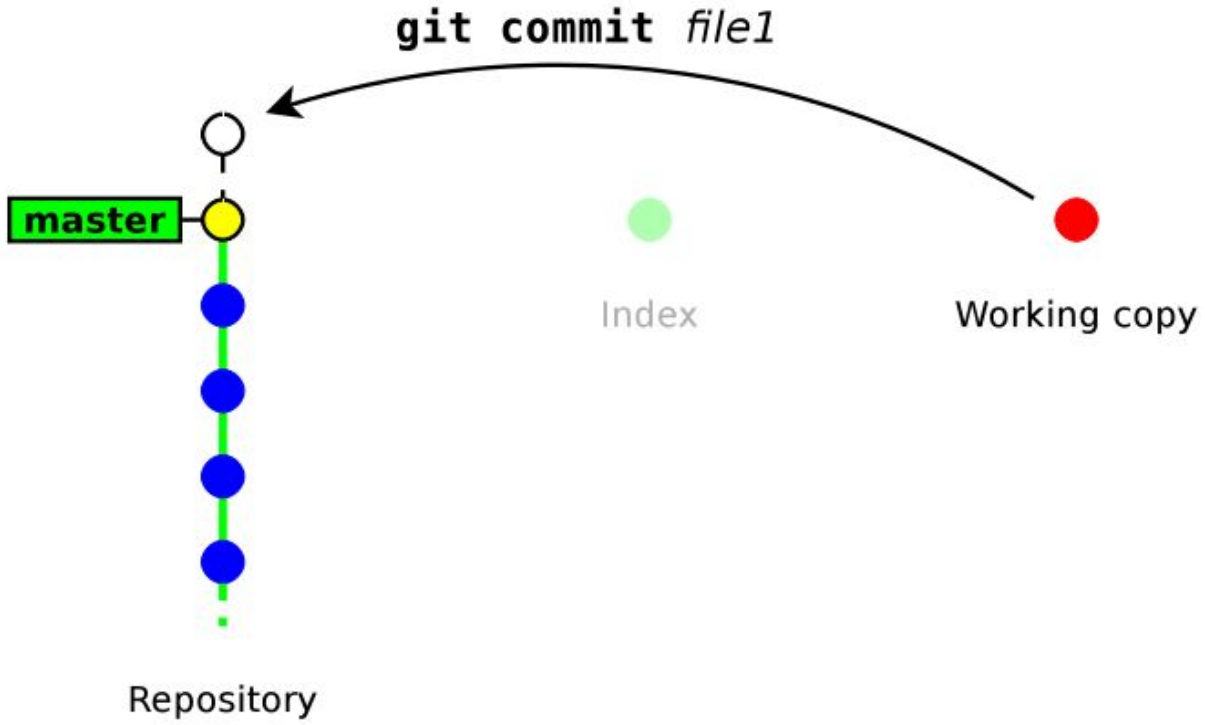

git commit
git add file1
master
Index
Working copy
Repository

Bypassing the index



- **`git commit -a`** is equivalent to running **`git add`** on all filenames that **existed** in the latest commit, and then running **`git commit`**.
- **`git commit 'files'`** creates a new commit containing the contents of the latest commit, plus a snapshot of files taken from the working directory. Additionally, files are copied to the stage.
- **`git checkout HEAD -- files`** copies files from the latest commit to both the stage and the working directory.

Visually



Deleting files

`git rm 'file'`

→ remove the file from the index **and** from the working copy

`git commit`

→ commit the index to the history

Check for differences

- **`git diff`**
  - show deltas between the working directory and the stage
- **`git diff --cached`**
  - show deltas between the stage and the last commit (HEAD)
- **`git diff HEAD`**
  - show deltas between the working directory and the last commit (HEAD)
- **`git diff 'branchA'`**
  - show deltas between the working directory and last commit in branch '`branchA`'
- **`git diff 'revisionA' 'revisionB'`**
  - show deltas between two revisions (commits) in the current branch

Each of theses commands can optionally take extra filename arguments that limit the differences to the named files

Other useful (local) commands

- **`git status`** → show the status of the index and working copy
- **`git show`** → show the details of a commit (metadata + diff)
- **`git log`** → show the history
- **`git mv`** → move/rename a file 9
- **`git tag`** → creating/deleting tags (to identify a particular revision)

Branching and Merging: Why?

You may have **multiple variants of the same software**, materialised as branches, e.g.

- a main branch
- a maintenance branch (to provide bugfixes in older releases)
- a development branch (to make disruptive changes)
- a release branch (to freeze code before a new release)

VC tools will help you to:

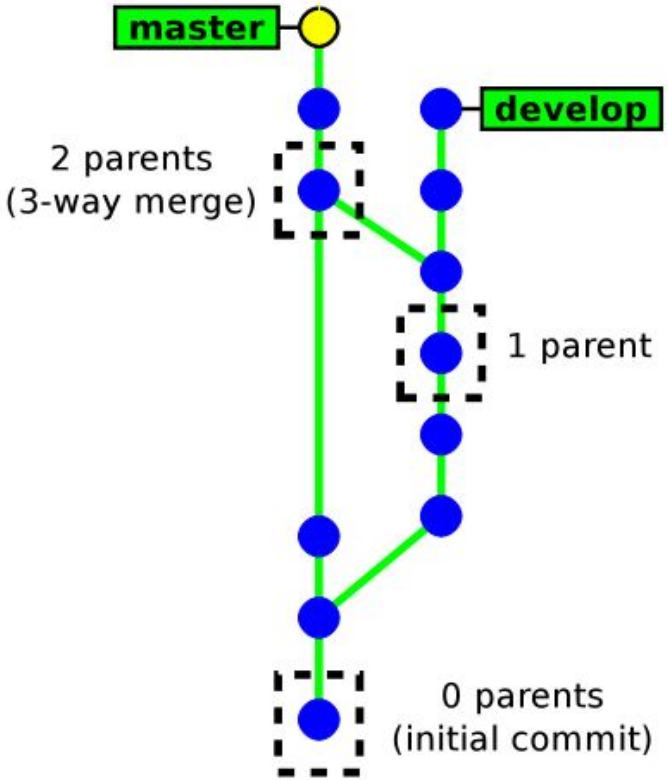
- handle multiple branches **concurrently**
- **merge** changes from a branch into another one

Branching and merging: Overview

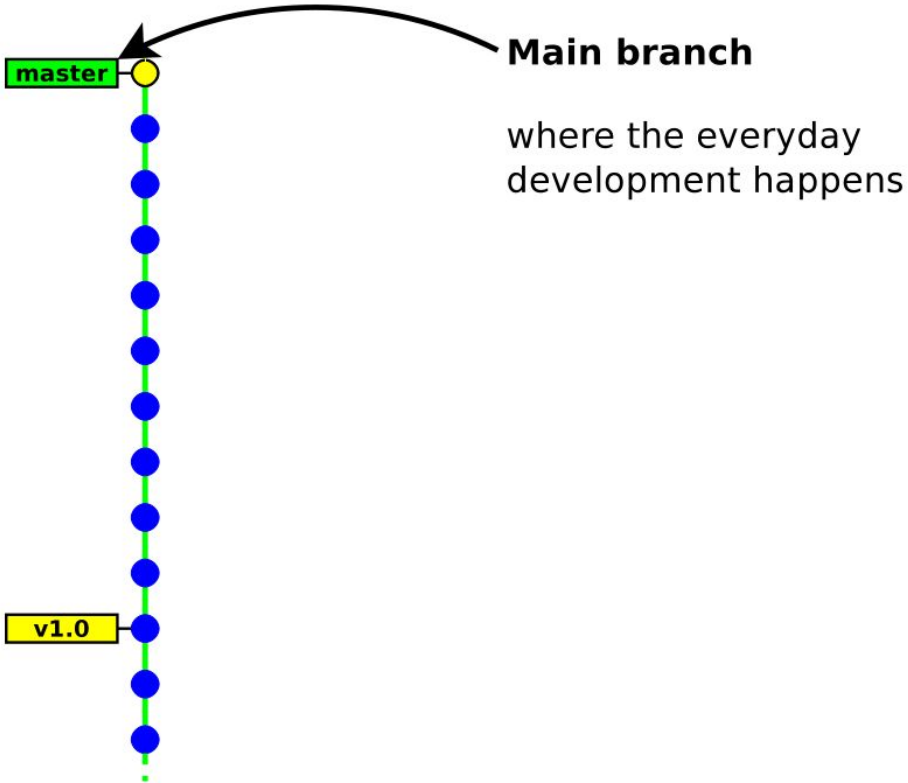
Each commit object has a list of parent commits

- 0 parents → initial commit
- 1 parent → ordinary commit
- 2+ parents → result of a merge

→ This is a Direct Acyclic Graph



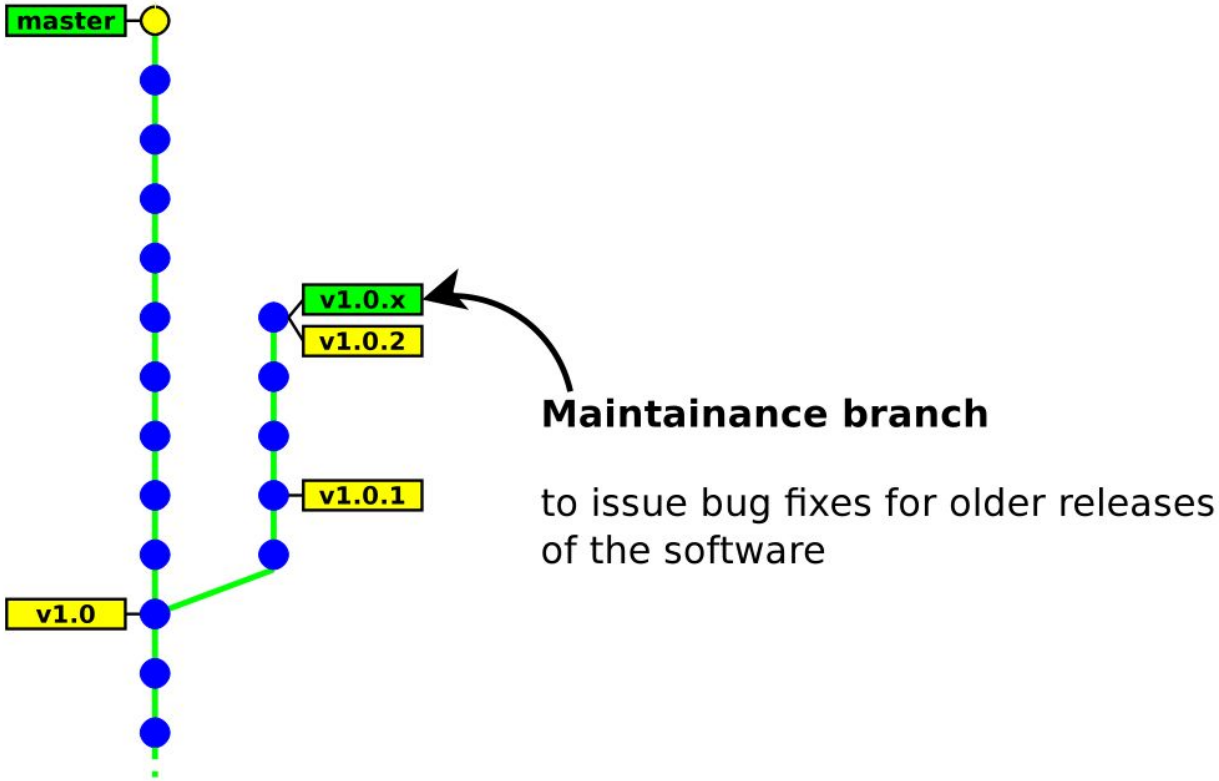
Branching and merging: Overview



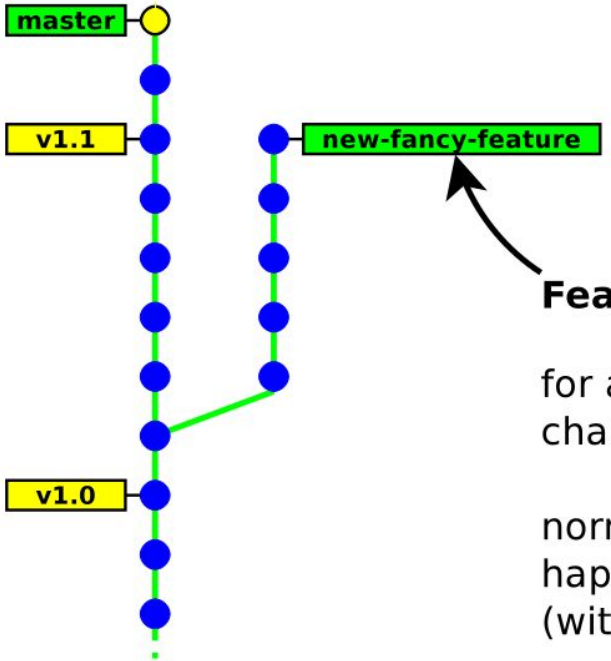
Main branch

where the everyday
development happens

Branching and merging: Overview



Branching and merging: Overview

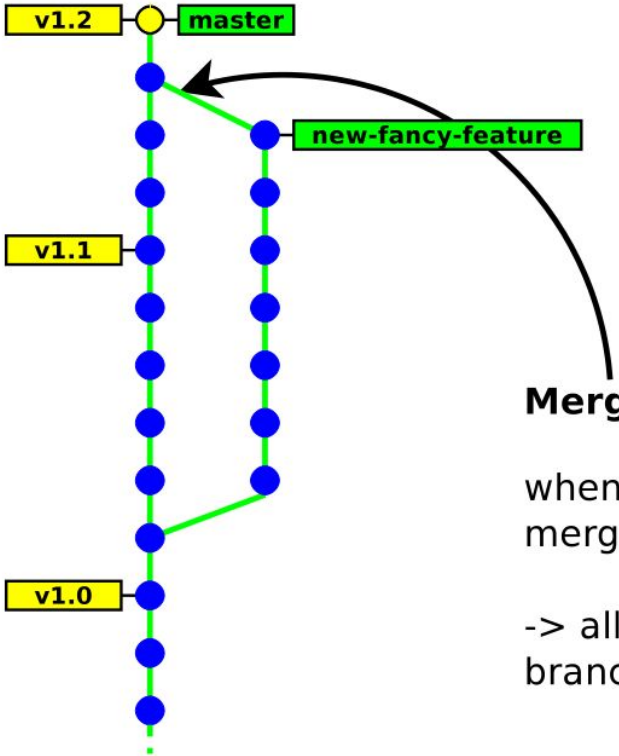


Feature branch

for a new feature requiring intrusive changes in the code

normal development continues to happen in the master branch (without disturbance)

Branching and merging: Overview

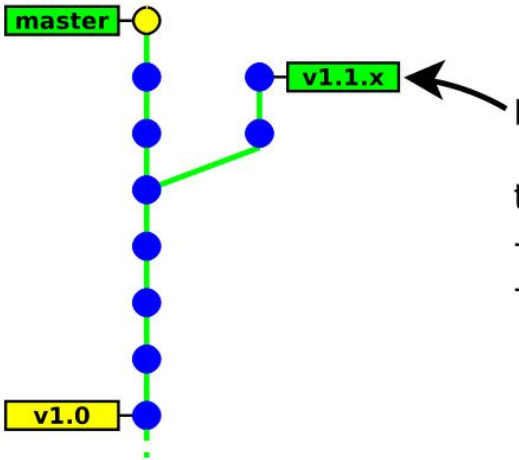


Merging

when the new feature is ready, it can merged back into the master branch

-> all changes done in the feature branch are imported

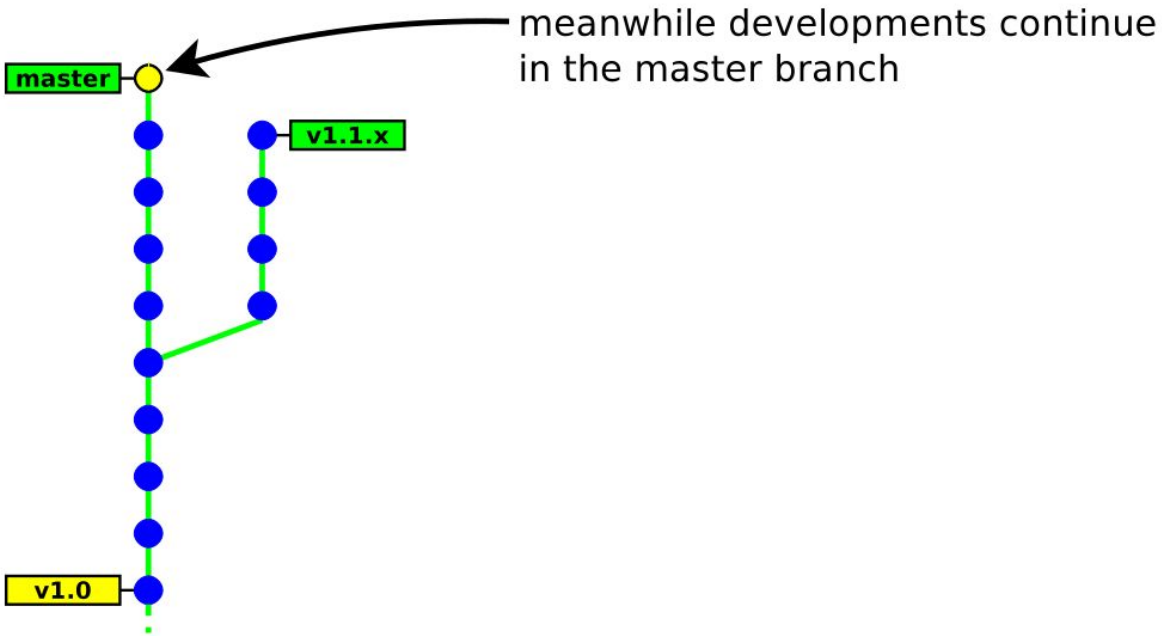
Branching and merging: Overview



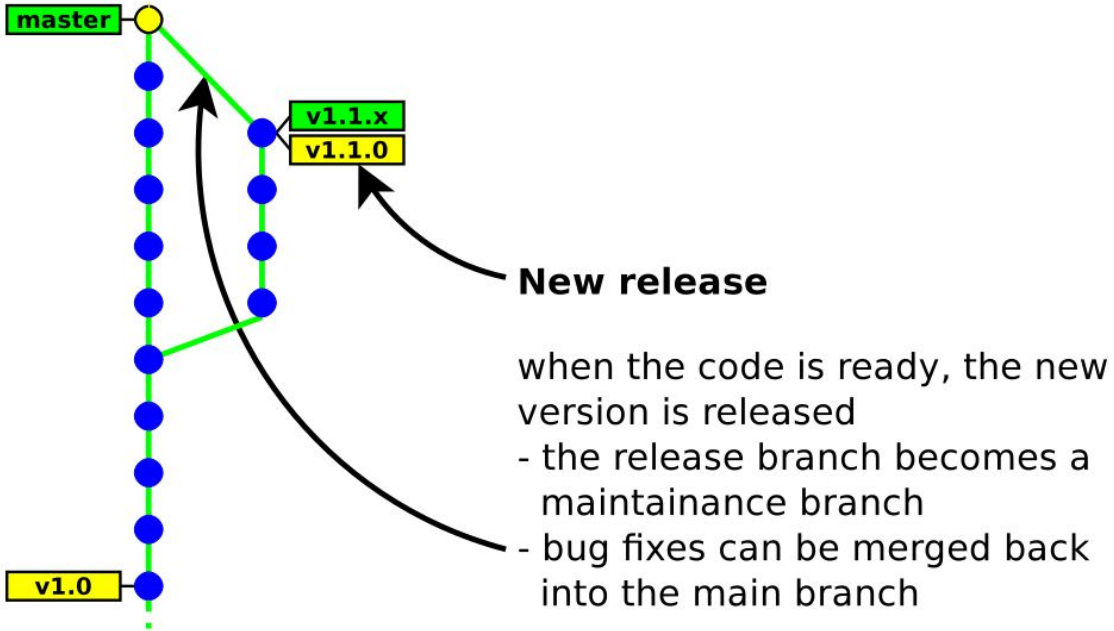
Release branch

to prepare the next release
- the code is frozen
- only bug fixes are accepted

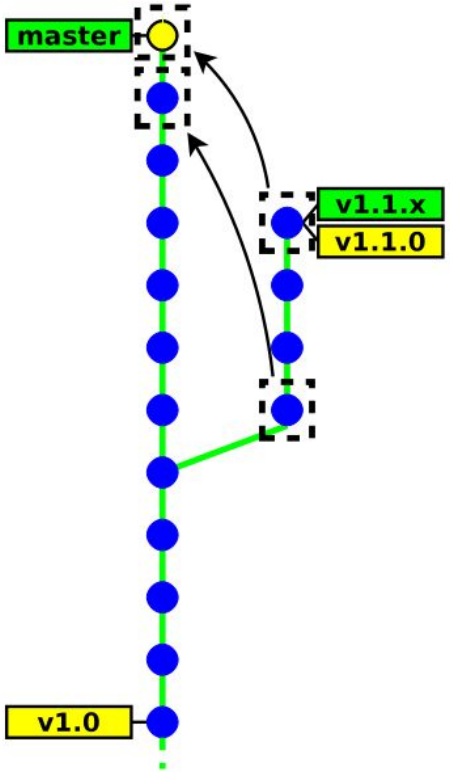
Branching and merging: Overview



Branching and merging: Overview



Branching and merging: Overview



Cherry picking

it may not be desirable to merge all the commits into the other branch (e.g. a bug may need a different fix)

-> it is possible to apply each commit individually

Branching and merging: Overview

There is no “branch history” → a branch is just a pointer on the latest commit.

Commits are identified with SHA-1 hash (160 bits) computed from:

- the commited files
- the meta data (commit message, author name, . . .)
- the hashes of the parent commits

→ A commit id (**hash**) identifies securely and reliably its content and all the previous revisions.

Manage branches: Basic commands

`git checkout -b 'new_branch' [ 'starting_point' ]`

- `'new_branch'` is the name of a new branch
- `'starting_point'` is the starting location of the branch (commit id, a tag, a branch, etc.). If not present, git will use the current location

`git checkout [-m] 'branchA'`

- Switch to `'branchA'`
- may fail when the working copy is not clean. Add -m to request merging your local changes into the destination branch.

`git branch -d 'branchA'`

- Delete (safely) `'branchA'`
- cannot delete: the current branch (HEAD) or a branch that has not yet been merged into the current branch

Merging: Basic command

```
git merge 'branchA'
```

- This will merge the changes in `'branchA'` into the current Branch.
- The result of `git merge` is immediately committed (unless there is a **conflict**)
- The new commit object has two parents→ the merge history is recorded
- `git merge` applies only the changes since the last common ancestor in the other branch→ if the branch was already merged previously, then only the changes since the last merge will be merged.

How Git merges files ?

If the same file was independently modified in the two branches, then Git needs to merge these two variants

- **textual files** are merged on a per-line basis:
  - lines changed in only one branch are **automatically merged**
  - if a line was modified in the two branches, then Git reports a **conflict**.
- **binary files** (figures, compiled objects, etc) always raise a **conflict** and require manual merging

In case of a conflict:

- unmerged files (those having conflicts) are left in the working tree and marked as “unmerged”. Git will refuse to commit the new revision until all the conflicts are explicitely resolved by the user
- the other files (free of conflicts) and the metadata (commit message, parents commits, ...) are automatically added into the index (the staging area)

Resolving conflicts

There are **two** ways to resolve conflicts:

- either edit the files **manually**, then run
  - `git add 'file'` to check the file into the index, or
  - `git rm 'file'` to delete the file
- or with a **conflict resolution tool** (xxdiff, kdiff3, emerge, ...)
  - `git mergetool [ 'file' ]`

Then, once all conflicting files are checked in the index, you just need to run

```
git commit
```

to commit the merge.

But everything is simple: several GUIs

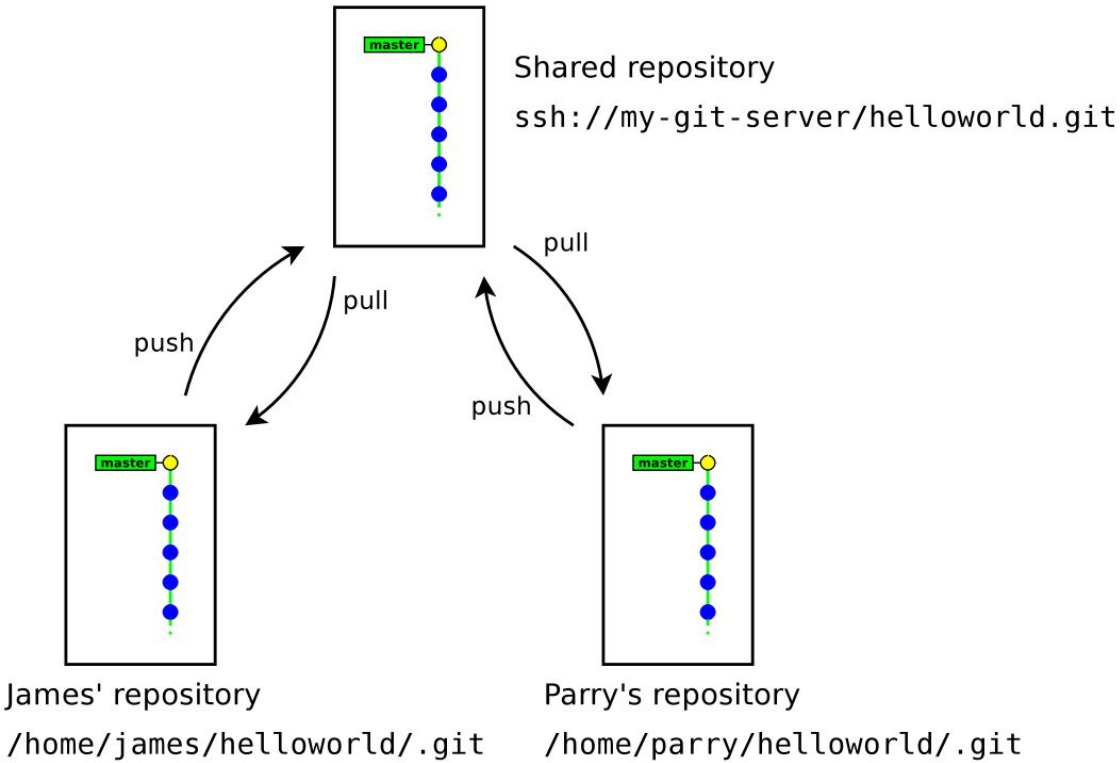
Platform dependent :

- Linux https://www.slant.co/topics/242/~best-graphical-git-clients-for-linux
- Mac OS X https://www.slant.co/topics/5574/~gui-git-clients-for-mac
- Windows https://www.slant.co/topics/2089/~git-clients-for-windows

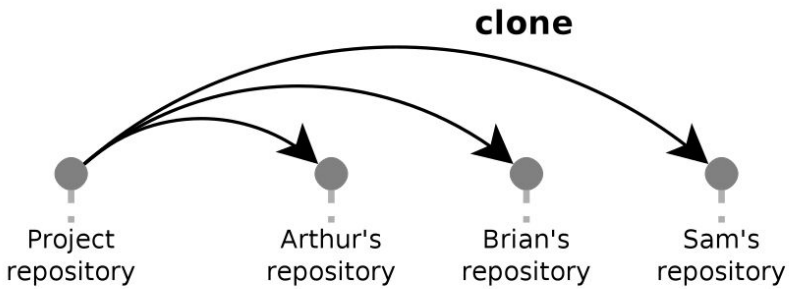
Mutlti-platform:

- E.g. gitkraken https://www.gitkraken.com/

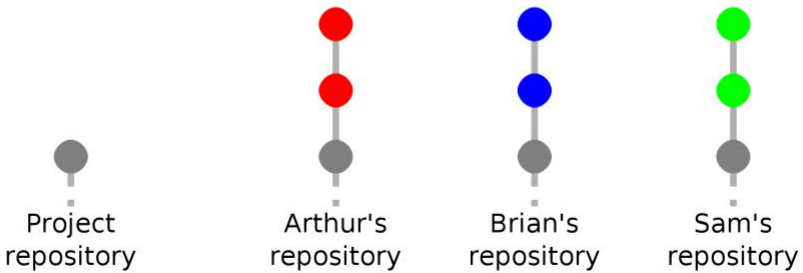
Part 2: Collaborative (distant) usage



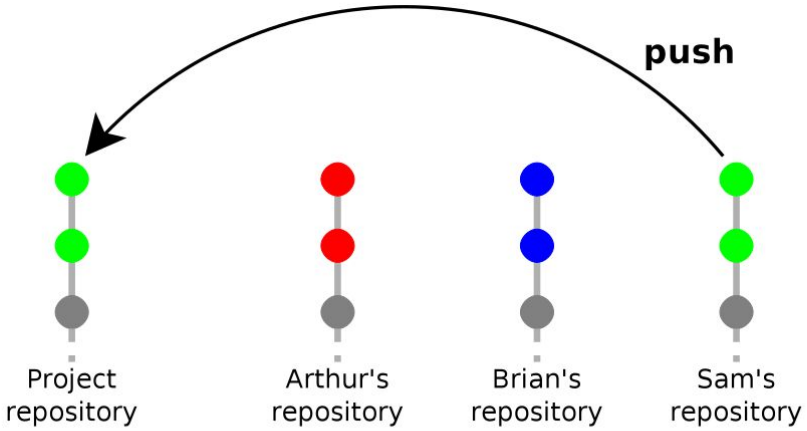
Part 2: Collaborative (distant) usage



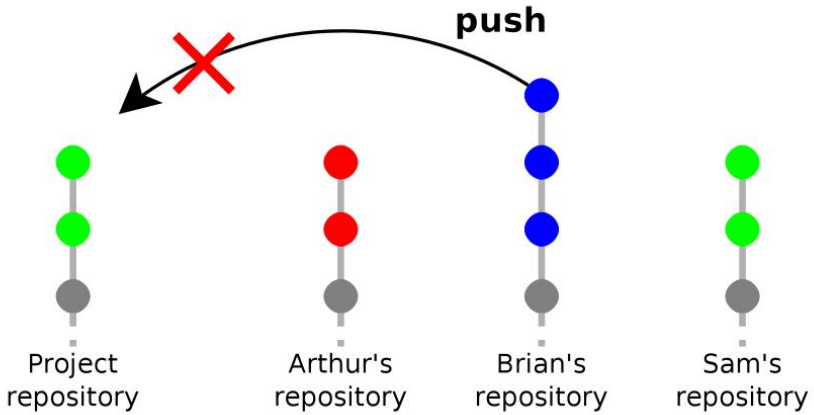
Part 2: Collaborative (distant) usage



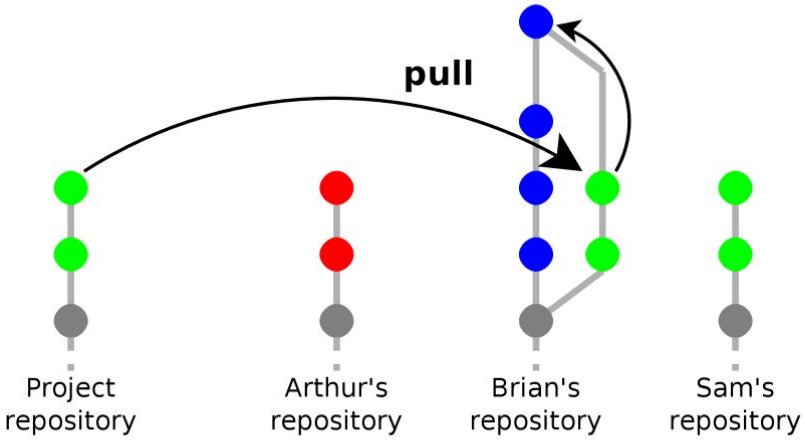
Part 2: Collaborative (distant) usage



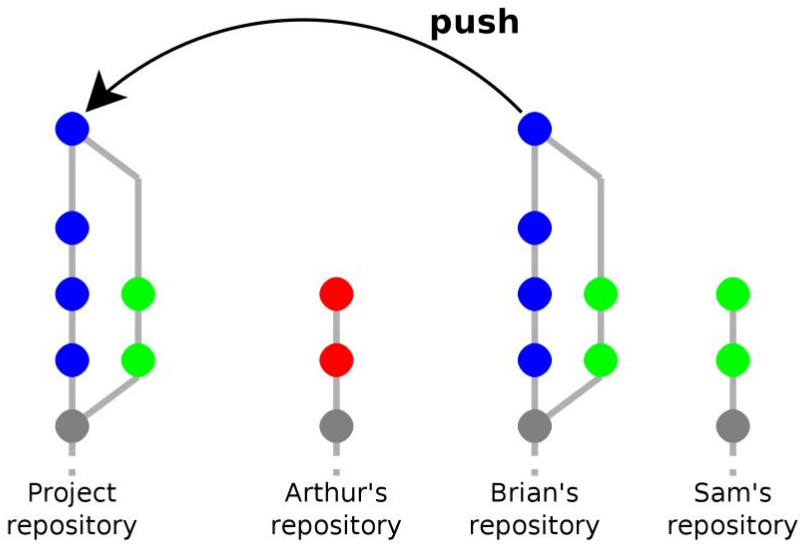
Part 2: Collaborative (distant) usage



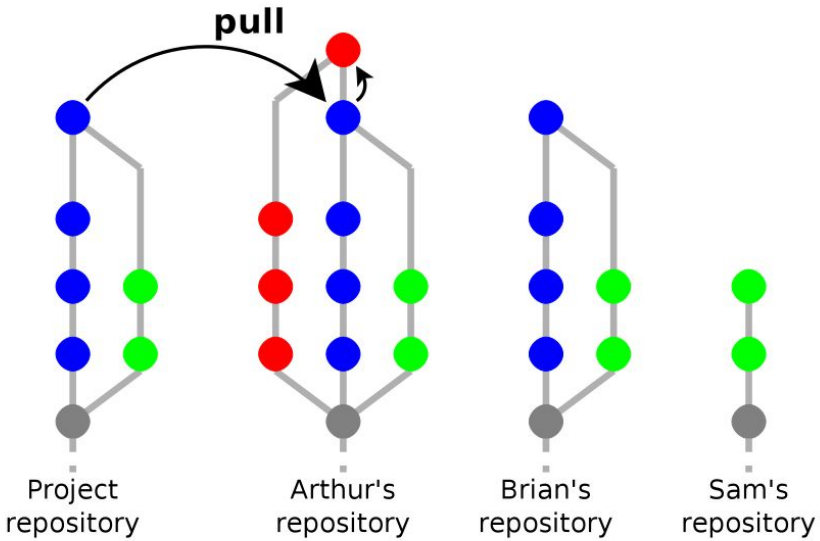
Part 2: Collaborative (distant) usage



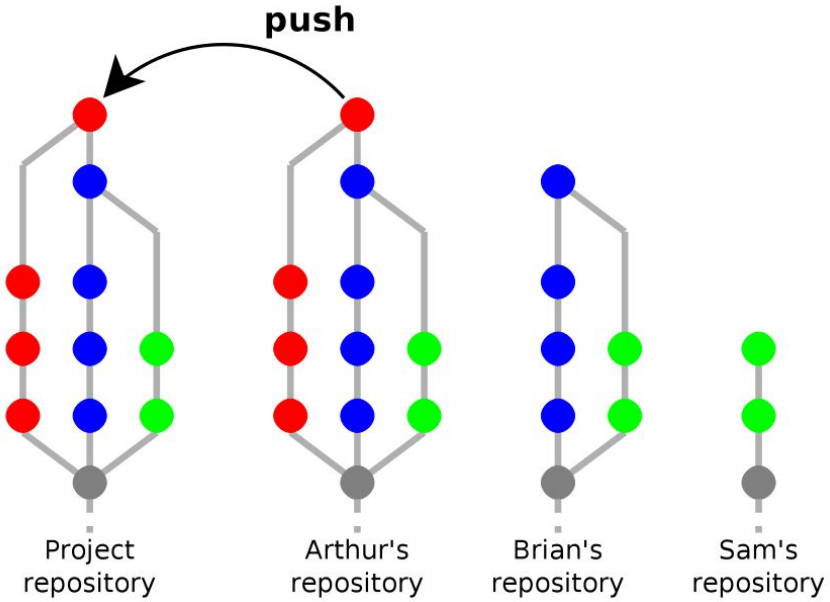
Part 2: Collaborative (distant) usage



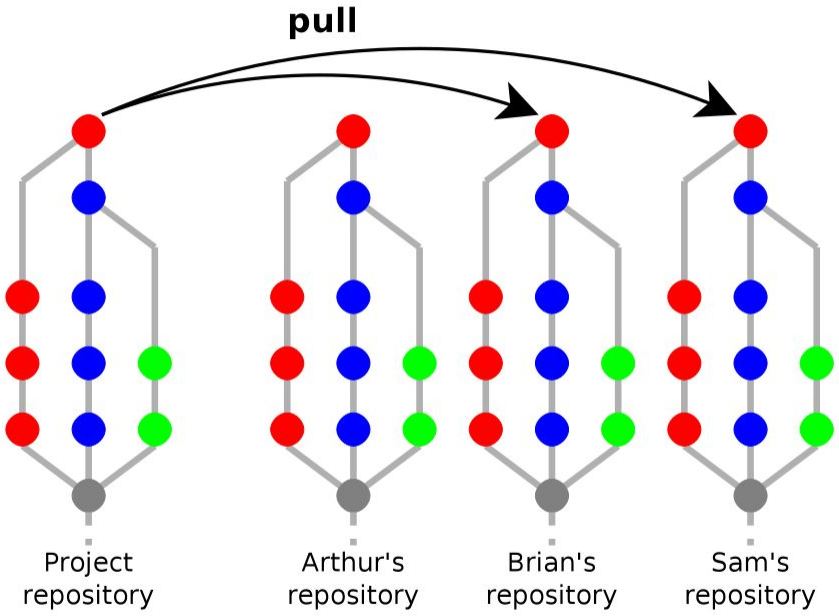
Part 2: Collaborative (distant) usage



Part 2: Collaborative (distant) usage



Part 2: Collaborative (distant) usage



How git handles remote repositories

- Remote repositories are **mirrored** within the local repository
- It is possible to work with **multiple** remote repositories
- Each remote repository is identified with a local **alias**.
  - When working with a unique remote repository, it is usually named **origin**
- Remote branches are mapped in a separate namespace: `remote/name/branch`.
  - Examples:
    - `master` refers to the local master branch
    - `remote/origin/master` refers to the **master** branch of the remote repository named **origin**

Add a remote

```
git remote add ‘name’ ‘url’
```

- `‘name’` is a local alias identifying the remote repository
- `‘url’` is the location of the remote repository

Pushing (uploading) local changes to the remote repository

```
git push [ --tags ]
```

- **`git push`** examines the current branch, then:
- if the branch is tracking an upstream branch, then the local changes (commits) are propagated to the remote branch
- if not, then nothing is uploaded (new local branches are considered private by default)
- In case of conflict **`git push`** will fail and require to run **`git pull`** first

Pushing a new branch to the remote repository

```
git push -u destination repository ref [ref . . . ]
```

- explicit variant of git push: the local reference ref (a branch or a tag) is pushed to the remote destination repository
- -u/--set-upstream configures the local branch to track the remote branch so that git pull an git push work with that repository by default (this is usually what you want)

Fetching (downloading) changes from the remote repository

`git fetch`

updates the local mirror of the remote repository:

- it downloads the new commits from the remote repository
- it updates the references `remote/remote_name/*` to match their counterpart in the remote repository

Example: the branch `remote/origin/master` in the local repository is updated to match the new position of the branch master in the remote repository

Merging remote changes into the current local branch

Changes in the remote repository can be merged explicitly into the local branch by running

```
git merge
```

In practice, it is more convenient to use

```
git pull
```

which is an alias to `git fetch + git merge`

Importing a new remote branch

```
git checkout 'branchA'
```

If the `'branchA'` does not exist locally, then GIT looks for it in the remote repositories. If it finds it, then it creates the local branch and configures it to track the remote branch.

Cloning a repository

```
git clone url [ directory ]
```

- It makes a local copy of a remote repository and configures it as its origin remote repository.
- It is a shortcut for the following sequence:
  1. `git init directory`
  2. `cd directory`
  3. `git remote add origin url`
  4. `git fetch`
  5. `git checkout master`
- In practice you will rarely use `git init`, `git remote` and `git fetch` directly, but rather use higher-level commands: `git clone` and `git pull`.

About third party contributions

Third-party contributors can submit their contributions by:

1. sending patches (the traditional way)
2. publishing their own (unofficial) repository and asking an official developer to merge from this repository (**pull request** or **merge request**)

Generating patches

`git format-patch`

converts you history (commits) into a series of patches (one file per commit) and it records the metadata (author name, commit message)

`git format-patch ‘rev_origin’ [ ‘rev_final’ ]`

generates patches from revision **`‘rev_origin’`** to **`‘rev_final’`** (or to the current version if not given)

Applying patches

```
git am file1 [ file2 ...]
```

- applies a series of patches generated by **`git format-patch`** into the local repository (**`am`** originally stands for “apply mailbox”)
- each patch produces one commit
- the authorship of the submitter is preserved (actually GIT distinguishes between the author and the committer of a revision; usually they refer to the same person, but not when running git am)

Explicit pull/push

`push` and **`pull`** can work on any arbitrary repository

- **`git push 'url' 'local' 'branch'`**
  - push the local **`'branch'`** to the repository **`'url'`**
- **`git pull 'url' 'remote' 'branch'`**
  - merge the **`'remote'`** branch from the repository **`'url'`** into the current local branch

Reviewing a remote branch

- **`git pull`** merges immediately the remote branch into the current local branch.
- In practice you may prefer to review it before merging

- **`git fetch 'url' 'branchA'`**
  - Fetch the branch **`'branchA'`** from the repository **`'url'`** and store it temporarily as **`FETCH_HEAD`**
  - The **`FETCH_HEAD`** reference remains valid until the next time **`git fetch`** is run

Advices

Do not put large data sets in your repository (e.g. Finite Elements results)

Commit as often as you can (keep independent changes in separate commits)

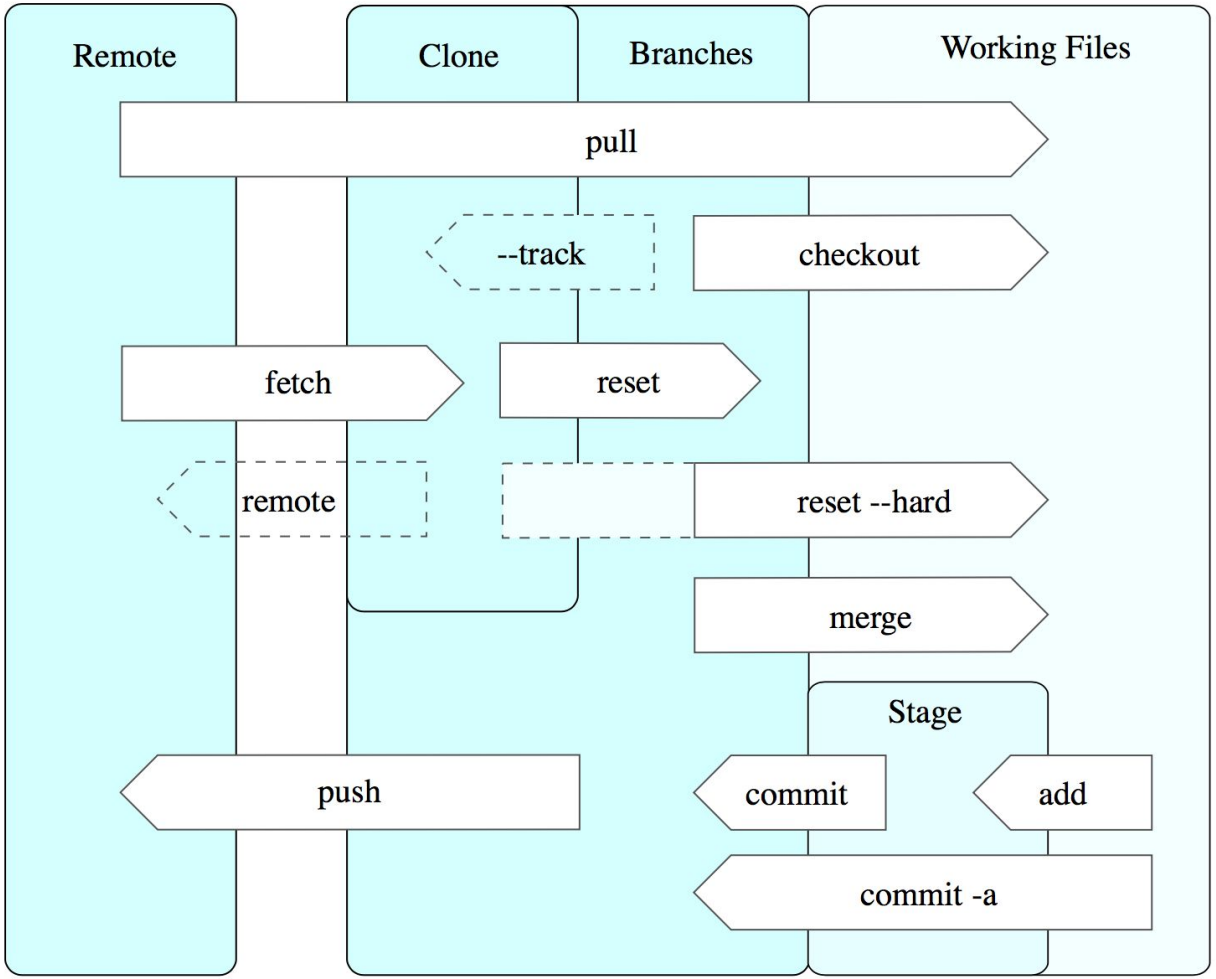
Run `git diff` before preparing a commit

In commit messages, describe the rationale behind of your changes (it is often more important than the change itself)

Do not forget to run `git push`

Version Control @ Université de La Rochelle

- The DSI provides two VC systems: SVN and GIT
- Use the SSH protocol
  1. Get the SSH key associated with your computer
     a. Check for an existing SSH key, if none
     b. Create a new SSH key (need of a passphrase)
  2. Add your SSH key to your ULR-GitLab account


Remote
Clone
Branches
Working Files
pull
--track
checkout
fetch
reset
remote
reset --hard
merge
Stage
push
commit
add
commit -a

Acknowledgement

Most of the following material (figures, definitions, commands) was taken from the following two presentations:

- GIT for Beginners (with courtesy of Anthony Baire)
- Visual Git Guide (with courtesy of Mark Lodato)

Further documentation

- `man git cmd` (tough & exhaustive)
- `man gitglossary`
- The Git book
- The Git community book
- Github learning materials
- Atlassian learning materials
  - Tutorial
  - Workflows
- GIT for Beginners (with courtesy of Anthony Baire)
- Visual Git Guide (with courtesy of Mark Lodato)