Introduction to Git & Gitlab

Robin Passama

January 2016, 13-14 th

What is Git ?

- Git: a distributed version control system (DVCS)
  - A set of software tools used to:
    - Register and retrieve different versions of a project.
    - Manage collaborative work.
  - Initially designed by Linus Torvalds to ease the development of the linux kernel,
    - free and open source.
  - Supported on most platforms.

What is Gitlab ?

- Gitlab: a project management web application
  - used to manage:
    - the life cycle of git projects.
    - the users of projects (roles, groups, etc.).
    - the communication between users.
  - Developed by a private company
    - “community edition” is free of use.
  - Deployed on a server at LIRMM: gite.lirmm.fr.

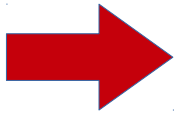
Why Git and Gitlab ?

- Postulate: you need a robust system to manage life cycle of your software.
  - Manage versions of your sources.
  - Add / Remove developers, change their roles within your project, etc.
  - Manage Bugs and other issues.
  - Manage online documentation.
  - Manage visibility of your projects.
  - Etc.

Why Git and Gitlab ?

- Git: currently the most widely used version control system
  - Very popular in the open source community.
  - Very popular in start-ups and software companies.

- Gitlab: like Github but private
  - ~ same functionalities as the very popular GitHub online service.
  - Private instances of gitlab can be deployed in restricted access environments.
  - Lots of public institutions and private companies use it.



Very mature, stable and fast improving solution

What can they be used for ?

- Software development
  - Made for controlling version of source code.
- Writing documents
  - Papers (latex sources).
  - Web pages (HTML, markdown, etc.).
  - Any kind of text document (ascii format).
- Archive projects binaries (less common use)
  - Released executable and libraries
  - CAD files, etc.

only for saving snapshots !

Objectives of this training day

- Learning:
  - root concepts of *git*
  - projects version control
  - collaborative work
  - project management using *Gitlab*
  - good practices

- Using:
  - basic commands of git (work station side)
  - Main functionalities of Gitlab (server side)

Plan

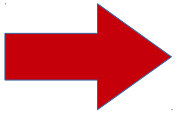
- **Installation**
- A brief history of version control systems
- GIT concepts
- GITLAB Server
- Step by step tutorial

Installation

- Installing git on a workstation
  - Linux (Ubuntu/Debian):
    - Base: `sudo apt-get install git-core`
    - GUI: `sudo apt-get install gitk`
    - Subversion interoperability : `sudo apt-get install git-svn`
  - MAC
    - Base + subversion: `sudo port install git-core +svn`

Installation

- Configuring git on a workstation
  - Configuring information about the user
    - `git config --global user.name "Robin Passama"`
    - `git config --global user.email "passama@lirmm.fr"`



Email will be used by Gitlab to identify the commits

  - Configuring git behavior
    - `git config --global color.diff auto`
    - `git config --global color.status auto`
    - `git config --global color.branch auto`

Installation

- Configuring SSH on a workstation
  - `cd ~/.ssh`
  - `ssh-keygen -b 2048 -t rsa`

    Enter a name for the key (e.g. name of key = name of target server)

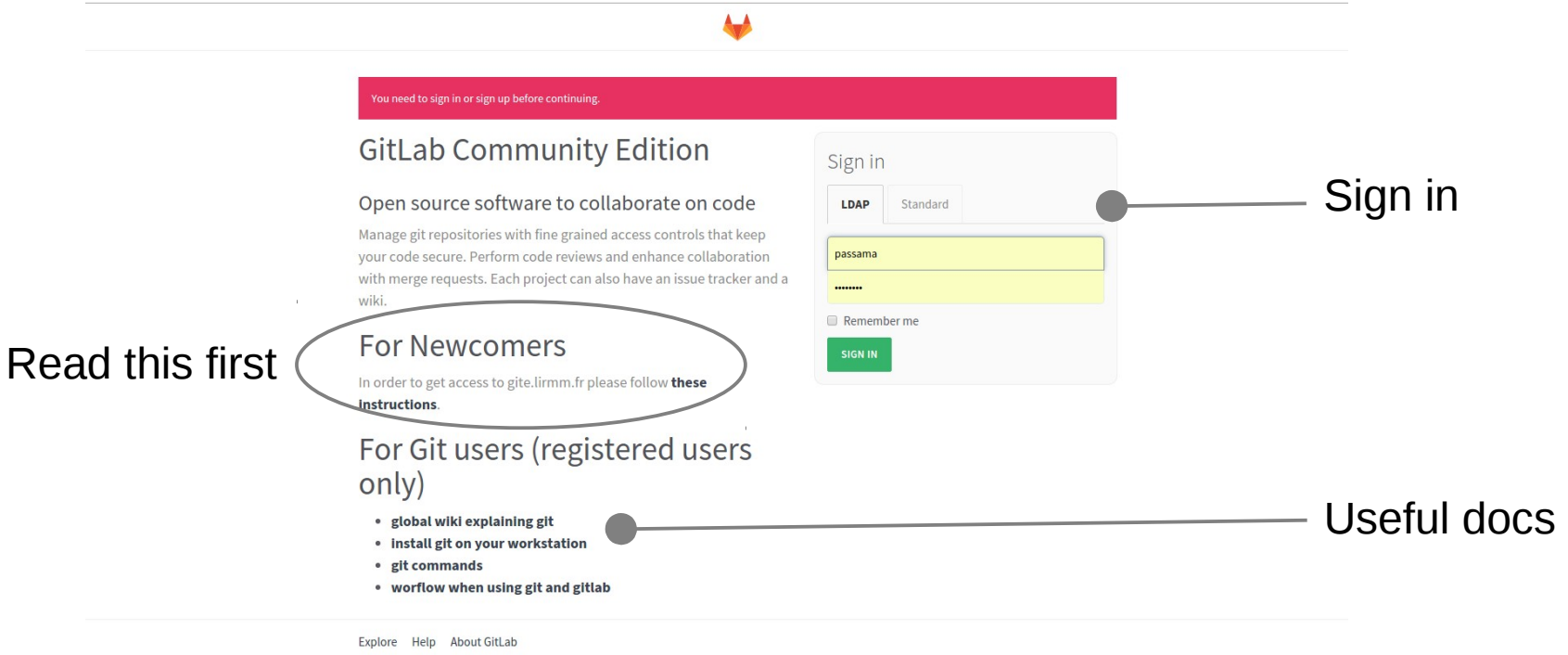
    Enter a pass phrase

    Result: **private and public keys generated**
  - `chmod 600 ~/.ssh/<your_private_key>`
  - Should be unnecessary on recent systems
  - `ssh-agent`
  - `ssh-add ~/.ssh/<your_private_key>`

    Enter pass phrase again

Installation

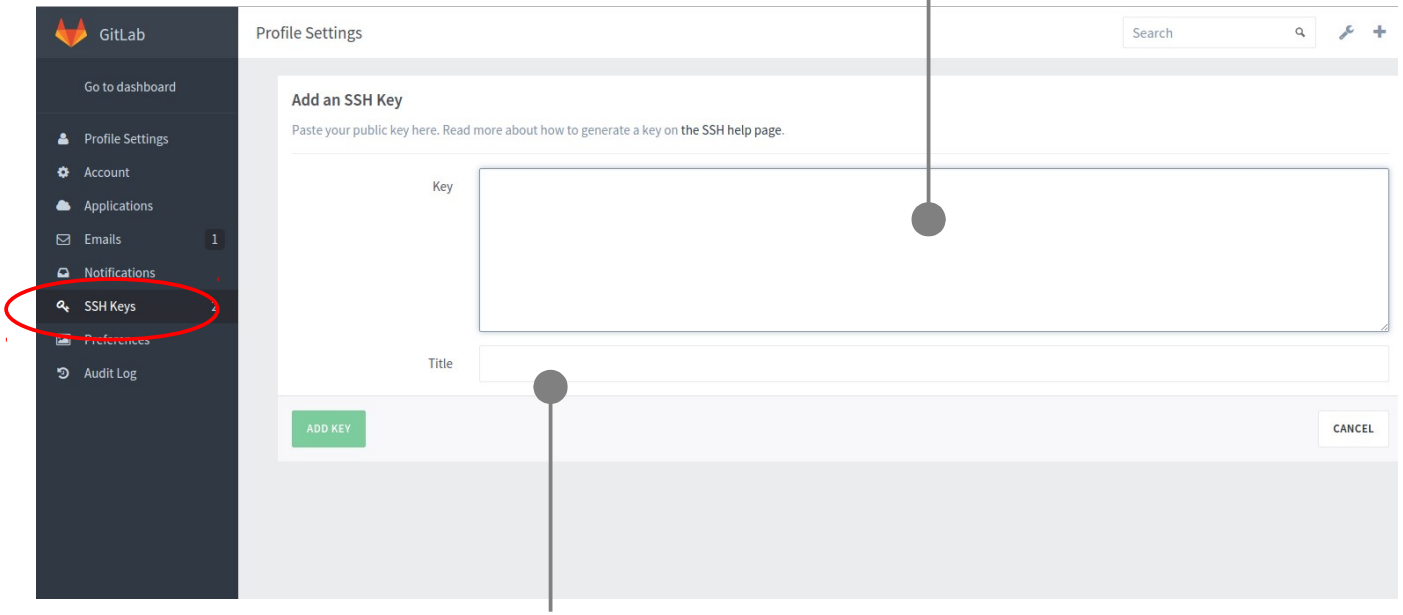
- Sign in into gitlab
  - for LIRMM members: use LIRMM LDAP
  - for external people: ask for a login/passwd, use “standard”



Installation

- Add a public ssh key on server
  - Go into your profile settings...

Copy/paste the content of your public ssh key



Title identifying your machine

Plan

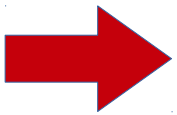
- Installation
- **A brief history of version control systems**
- GIT concepts
- GITLAB Server
- Step by step tutorial

History of VCS

- During project development, you need to register some snapshots, called **version** or revision.
  - Finding a specific version (last stable, last released, etc.).
  - Being able to test some new ideas without “loosing” previous functional code.
- Collaborative work: you need to share your changes and get changes made by others.
  - Manage **merging** of changes.
  - Understand what has been modified by another person.

History of VCS

- The forerunner of VCS: CPOLD
  - Just a development “methodology”, no specific tool.
  - A folder = a project **repository**.
  - A file/subfolder name = a **file/folder** of the project.
  - A file/folder extension = a **version** of the file/folder.
  - The unique command to know: `cp`
    - `cp file1 file1.old`
    - `cp -R folder folder.1.4 //suppose 1.3 exists`
    - **Require strong naming and sharing (access to file server) conventions**.



Quickly become difficult to manage (large or collaborative projects).

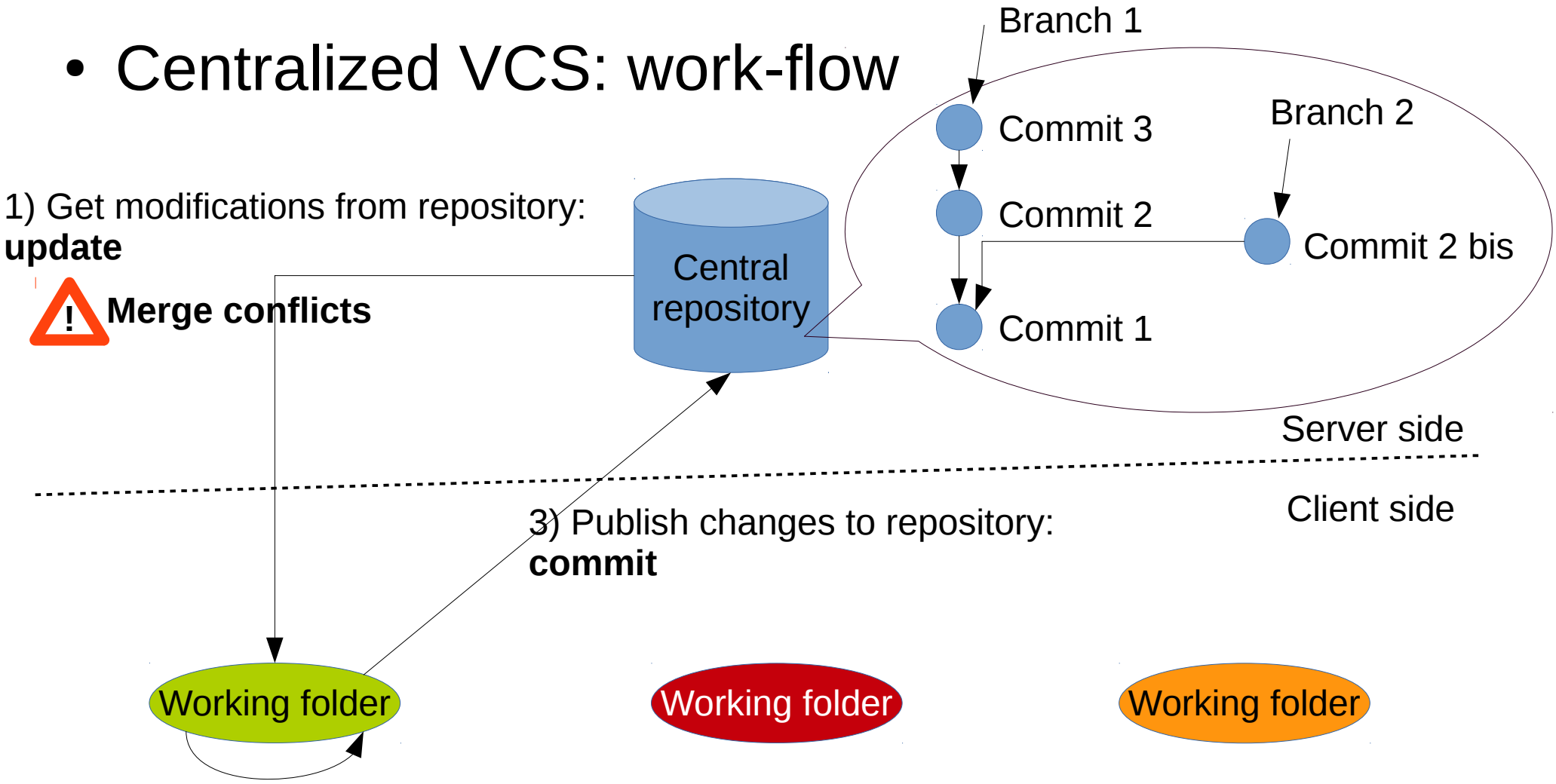
Really not optimal in term of project size on disk.

History of VCS

- Centralized VCS: CVS, subversion, etc.
  - Set of tools based on client/server approach.
    - Use kind of *diff* and *patch* commands to automate the management of difference between files.
    - A **specific file server** manages access to and operation on files/folders.
  - A **repository** = a specific folder on the server.
  - Any modification of the repository's content is registered.
    - Files and folders modifications are registered as **kind of patches**, patch are ordered.
    - Modifications of many files/sub-folders can be registered as a unique revision, called **commit**.
  - On a user workstation, only a **working copy** of the repository.

History of VCS

- Centralized VCS: work-flow



History of VCS

- Centralized VCS: known limitations
  - Committing
    - Impossible to work locally on a isolated set of commits, each commit requires to synchronize with the server.
    - Need to resolve **merge conflicts** BEFORE committing (painful in collaborative work).
    - Network access required to create commits !
  - Robustness
    - Loosing a repository (corruption of data, server HD crash, etc.) is loosing the history of commits.
    - requires additional backup procedures for the server, not managed by the VCS.

History of VCS

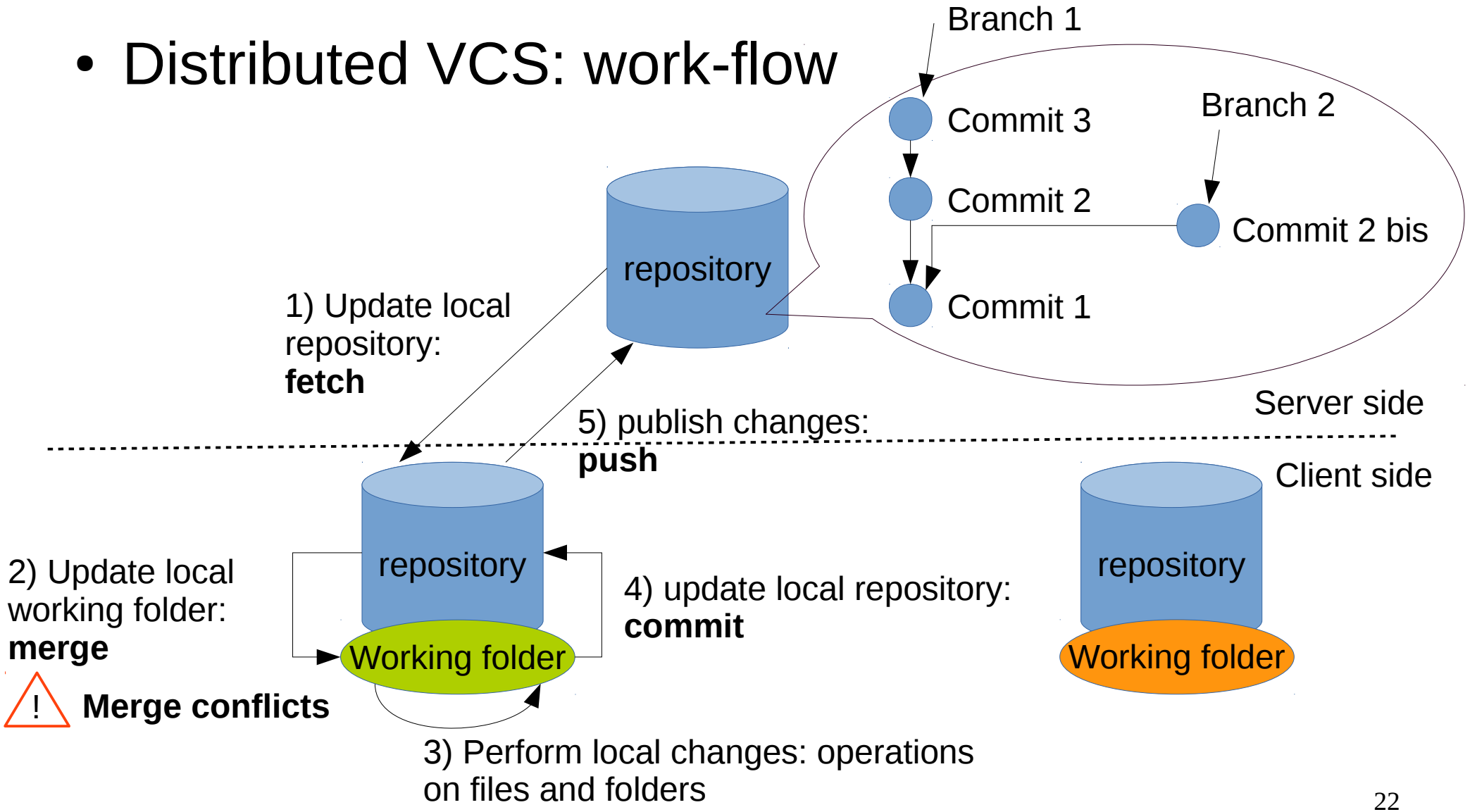
- Centralized VCS: known limitations
  - Centralization
    - Network latency on many operations.
    - Access rights are fixed, not easy to organize into sub-groups with specific rights (may be critic for large projects).
  - Branching model (SVN)
    - Merge of branches is a complex operation.
    - Drastically increases size of the repository.

History of VCS

- Distributed VCS: git, Bazaar, mercurial, etc.
  - Like centralized VCS but ...
  - using a peer-to-peer approach for synchronization operations.
  - A **repository** = a specific folder either on a server or on a workstation.
  - On a user workstation, a **working directory connected to a local copy of the repository**.

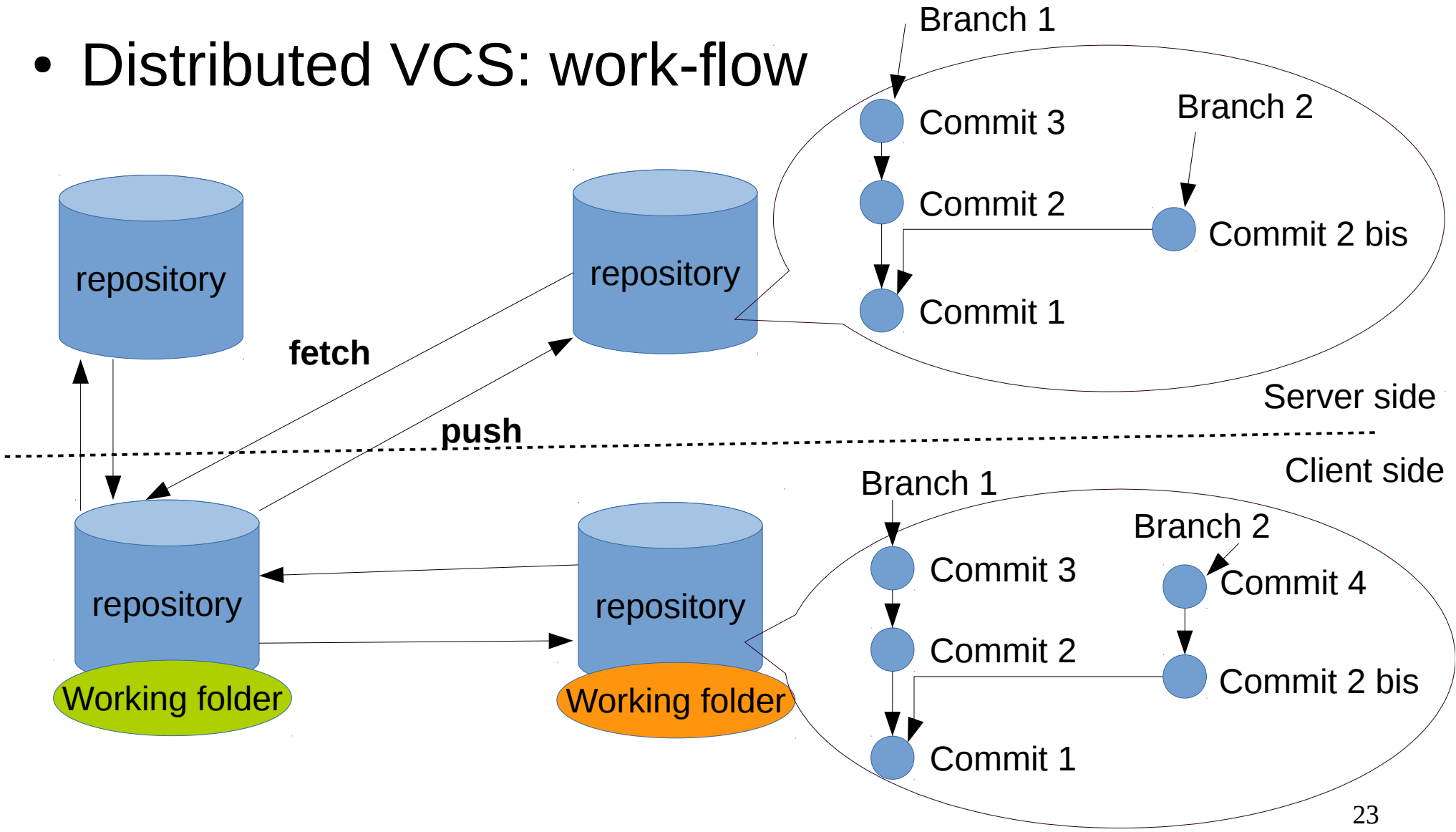
History of VCS

- Distributed VCS: work-flow



History of VCS

- Distributed VCS: work-flow

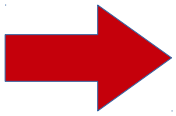


History of VCS

- Distributed VCS: Advantages VS Centralized VCS
  - Robustness
    - Any workstation repository is a full copy of the server repository.
  - Distributed
    - Very fast, no network latency on many operation.
    - Local commits
    - Each local repository can define its own access rights.
  - Branching model (Git)
    - nearly no additional cost in disk space.
    - Merging branches is a basic operation.
  - Team work-flow organization
    - Any model model is possible (even “centralized”)

History of VCS

- Distributed VCS: limitations VS Centralized VCS
  - More commands to know and understand.
  - Simpler to organize around a centralized well known server rather than a collection of distributed repositories.
    - Require to define a clear team work-flow when using DVCS.
  - **Enforcing a policy for access control is impossible with DVCS**
    - Every repository has a full copy of whole history of commits (from last **fetch**).
    - Every repository can define its own access rights to others.



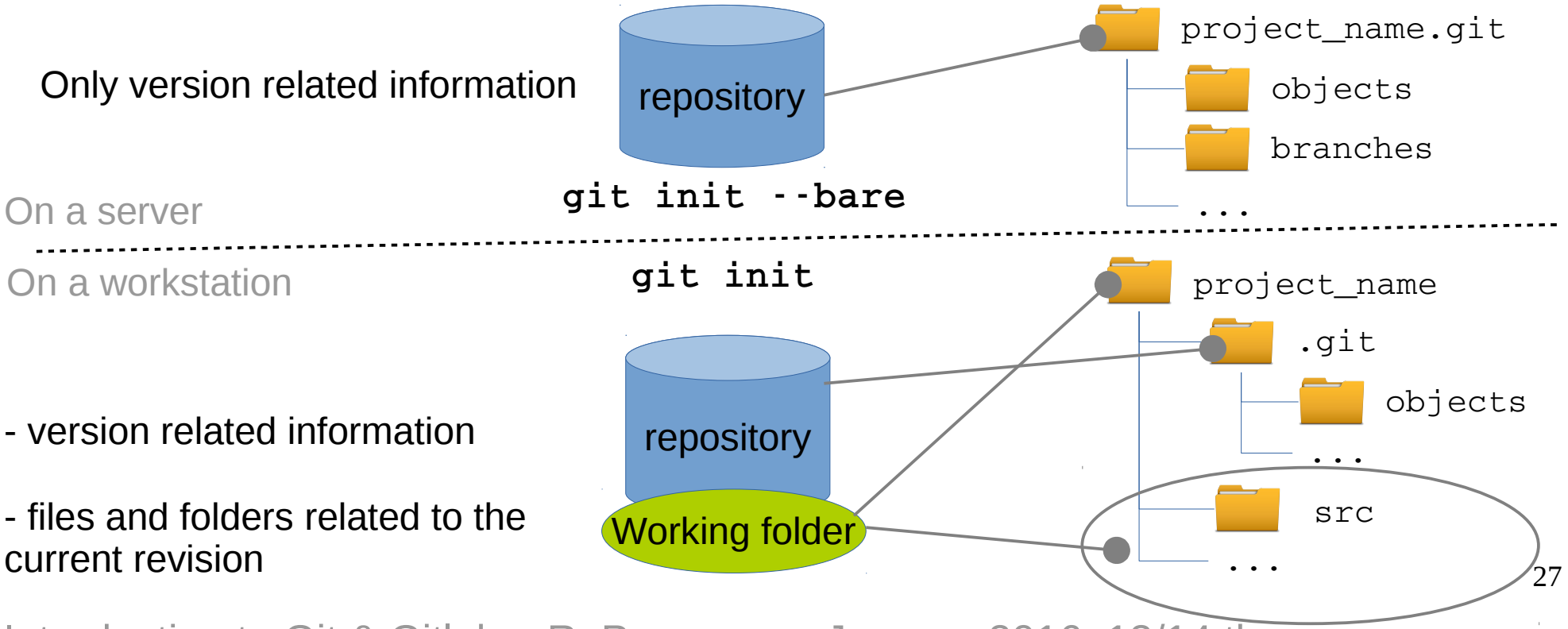
DVCS are intensively used in open source world, but less used in industry.

Plan

- Installation
- A brief history of version control systems
- **GIT concepts**
- GITLAB Server
- Step by step tutorial

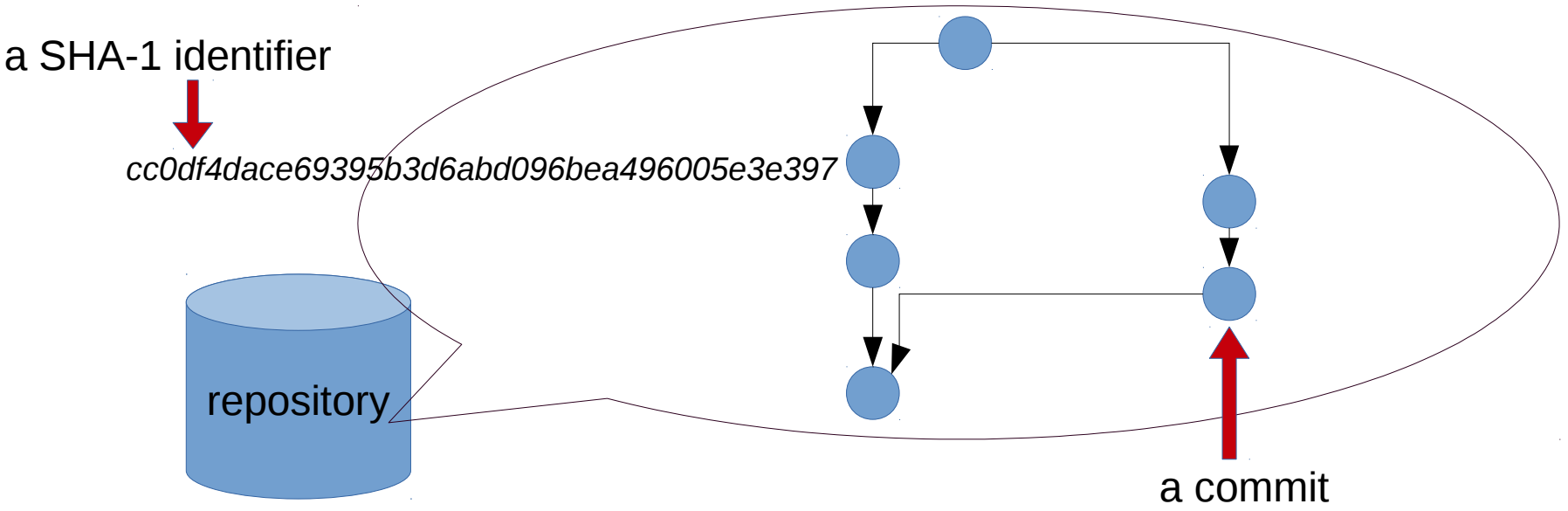
GIT Concepts

- Base element: the **repository**
  - A folder with specific information related to versions
  - Creating a repository from a folder: **`git init`**



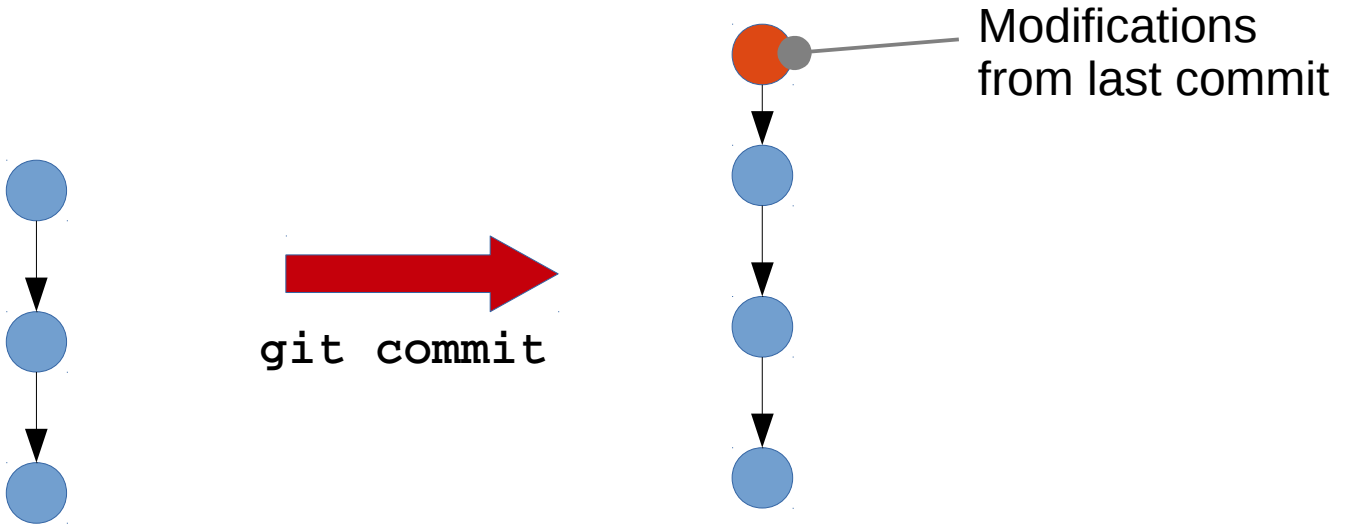
GIT Concepts

- Atomic element in repository: the **commit**
  - A kind of **patch** operation (a set of modifications on files and folders content)
  - Uniquely identified by a SHA-1 hash code
  - Ordered as an acyclic oriented graph



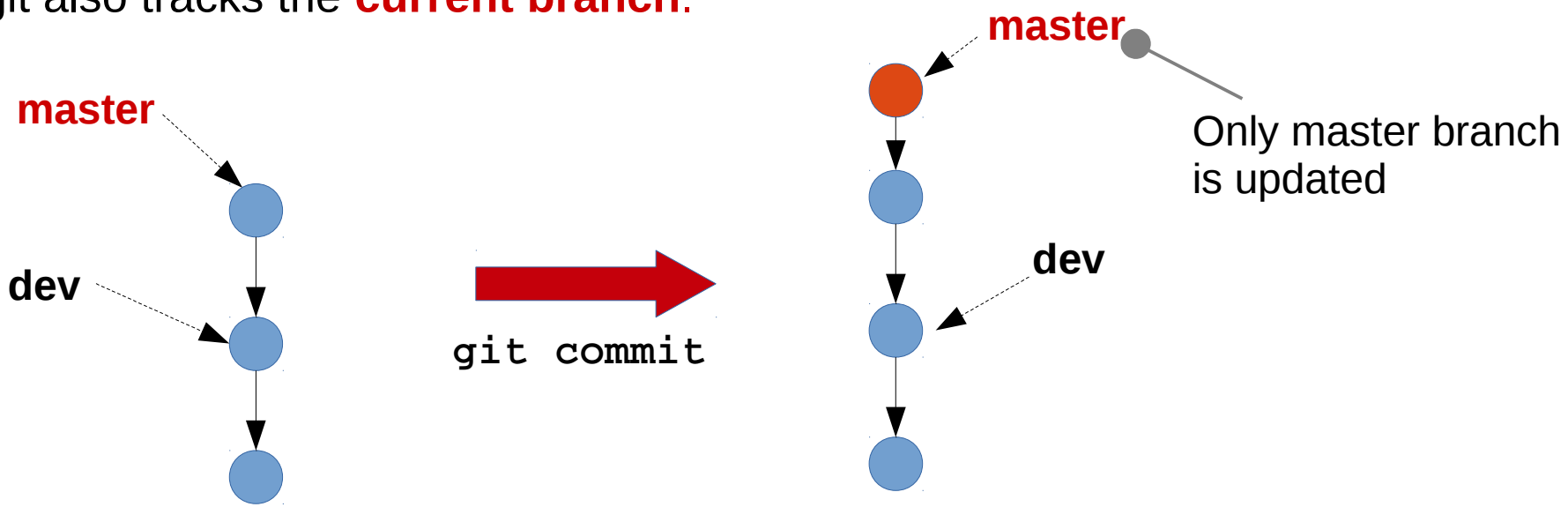
GIT Concepts

- Atomic element in repository: the **commit**
  - Graph is updated when **committing**
  - The new commit contains the modifications performed from last commit



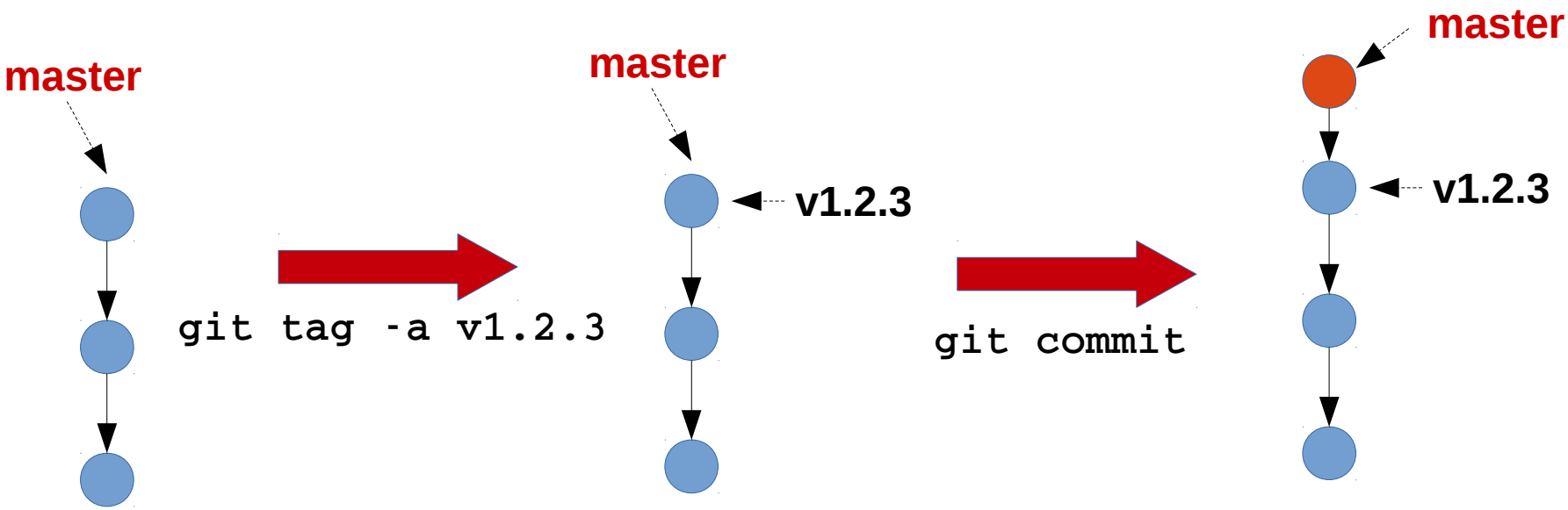
GIT Concepts

- Managing graph of commits with **branches**
  - A branch is a **pointer** on (i.e. a reference to) a commit.
  - This pointer is updated when **committing:** it points to the new commit.
  - The default branch is called **master.**
  - As many branches as you want**.**
  - git also tracks the **current branch**.



GIT Concepts

- Memorizing interesting states with **tags**
  - A tag is a **pointer** on (i.e. a reference to) a commit.
  - Once created this pointer is **never updated**.

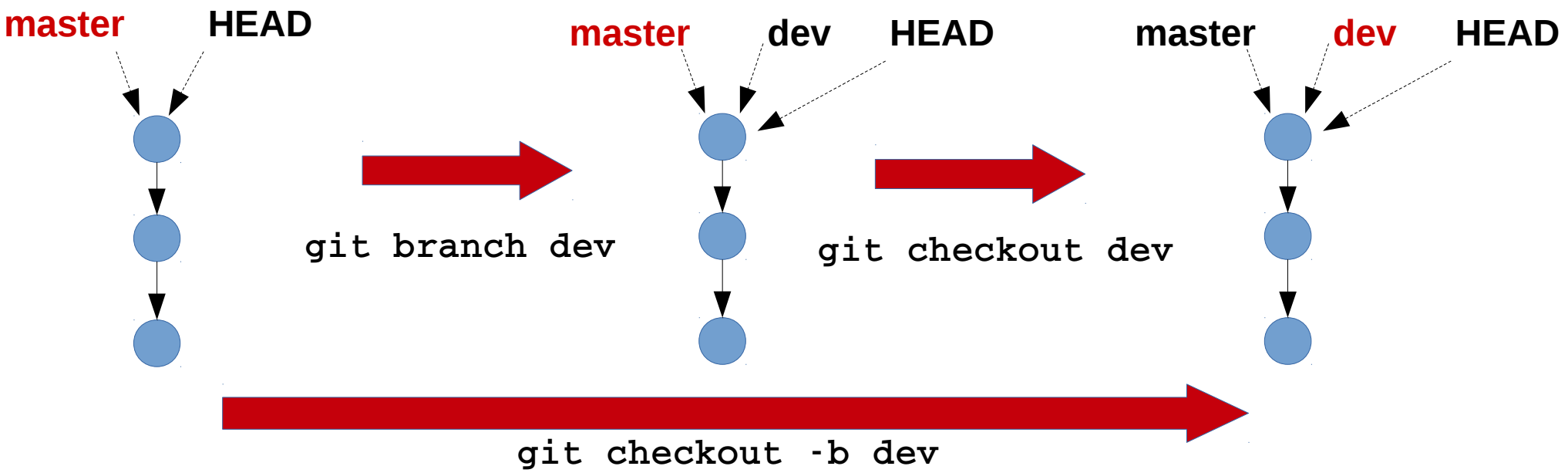


GIT Concepts

- Memorizing **current state** and navigating in the graph
  - A specific pointer is called **HEAD**
    - Represents the **current commit**.
    - Used to build **current state of the working folder**.
    - New commits will be created on top of the commit pointed by **HEAD**.

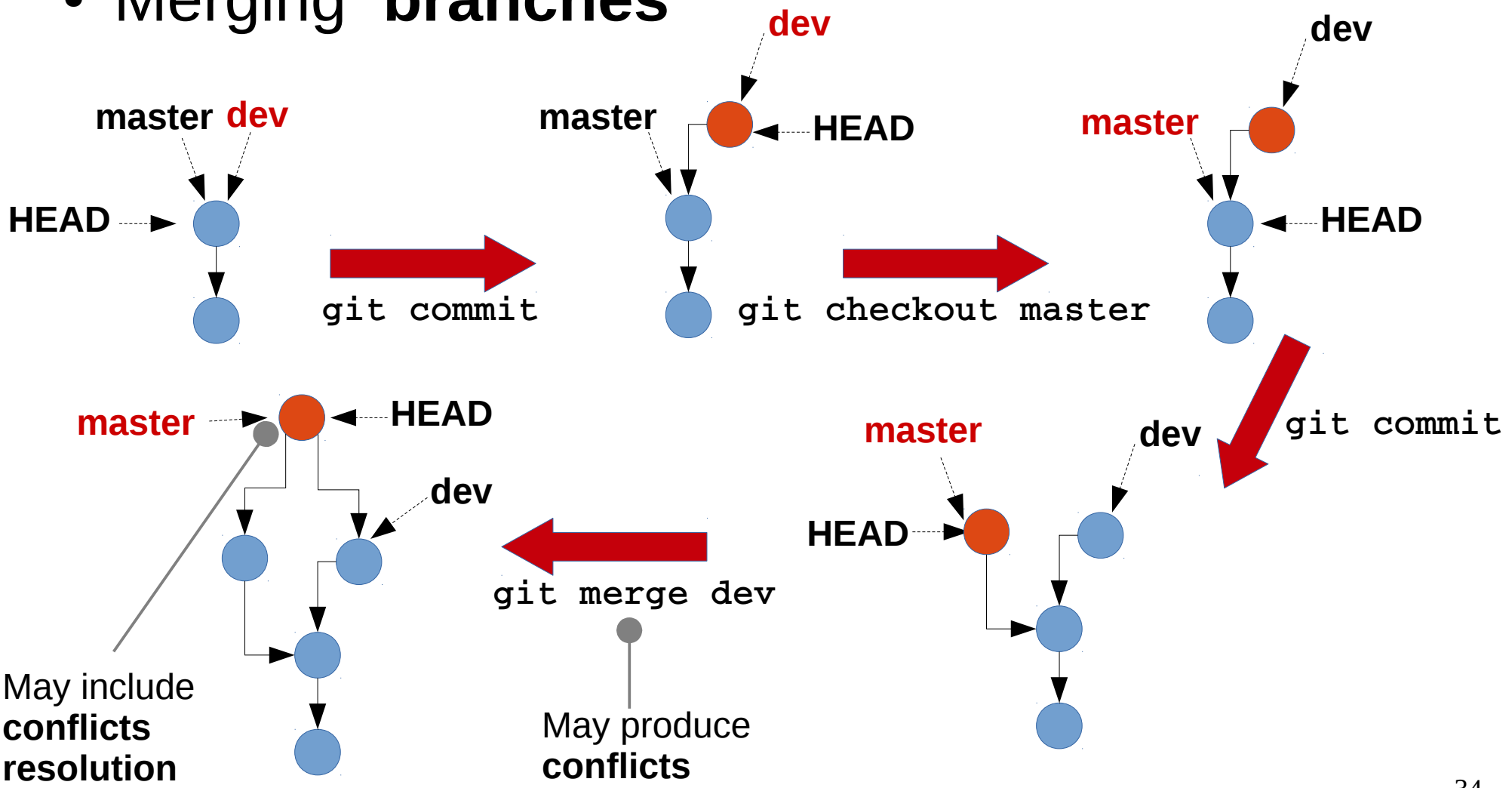
GIT Concepts

- Creating **branches**
  - Many branches can live in the same repository
  - Creating a branch = creating a new pointer on **HEAD**



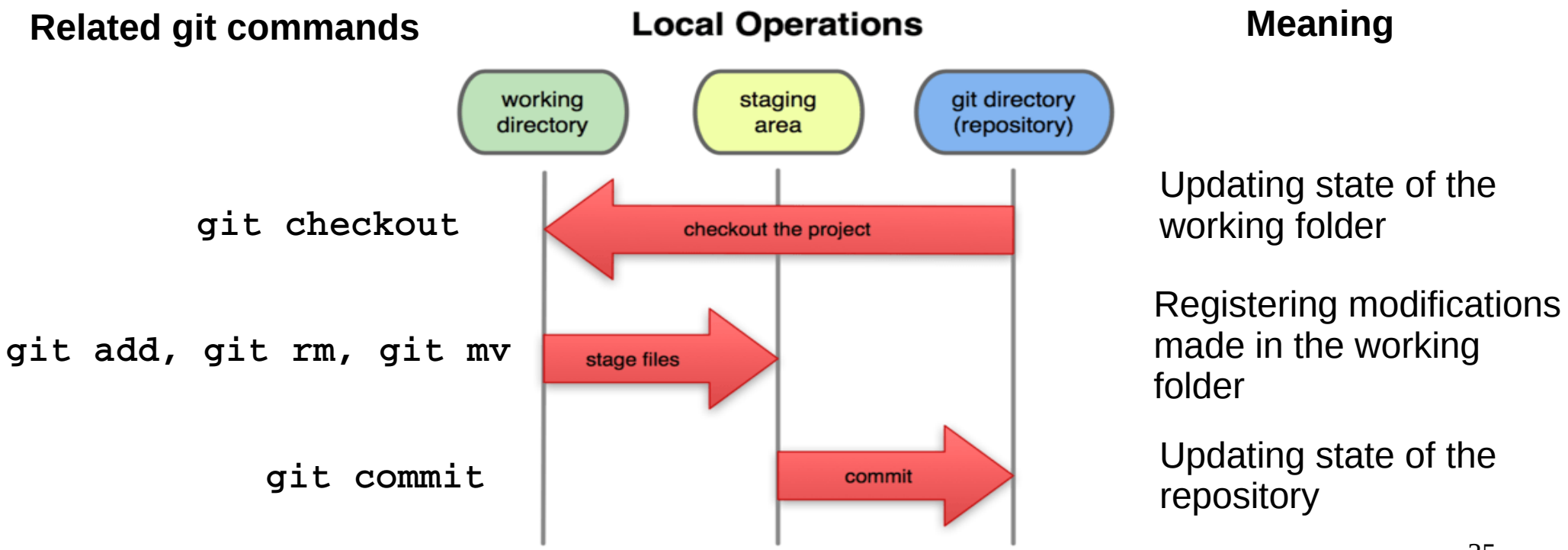
GIT Concepts

- Merging **branches**



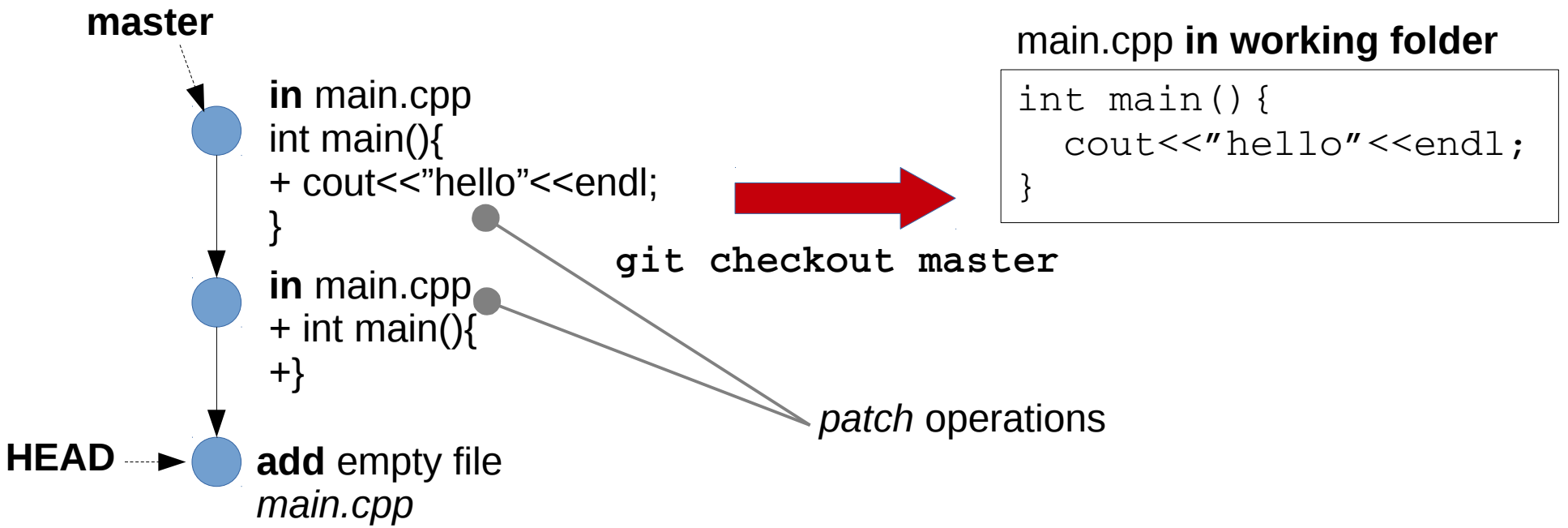
GIT Concepts

- Local work-flow overview
  - Goal: “synchronize” the content of the working folder with the local repository
  - Originality of git: 2 phases



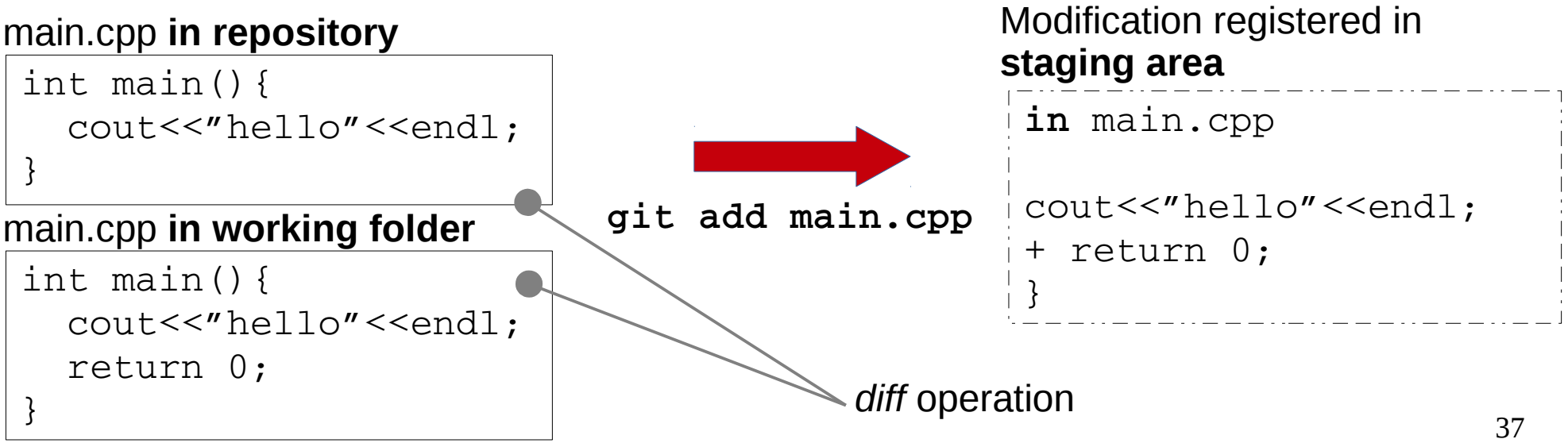
GIT Concepts

- Updating state of the working folder
  - Any time when `checkout (or merge)` is performed
  - ~ applying the sequence of patches starting from first commit to the current commit (**HEAD**)



GIT Concepts

- **Staging**: registering modifications of the working folder
  - Selecting modifications to be part of the next commit
    - Files content modifications can be selected or not (`git add -p`)
    - Entire file modification can be selected or new files tracked (`git add filename`)
    - Removed files must be untracked (using `git rm filename`)
    - Registering all modifications (using `git add --all`)
  - Undo staging (deselecting): `git reset`



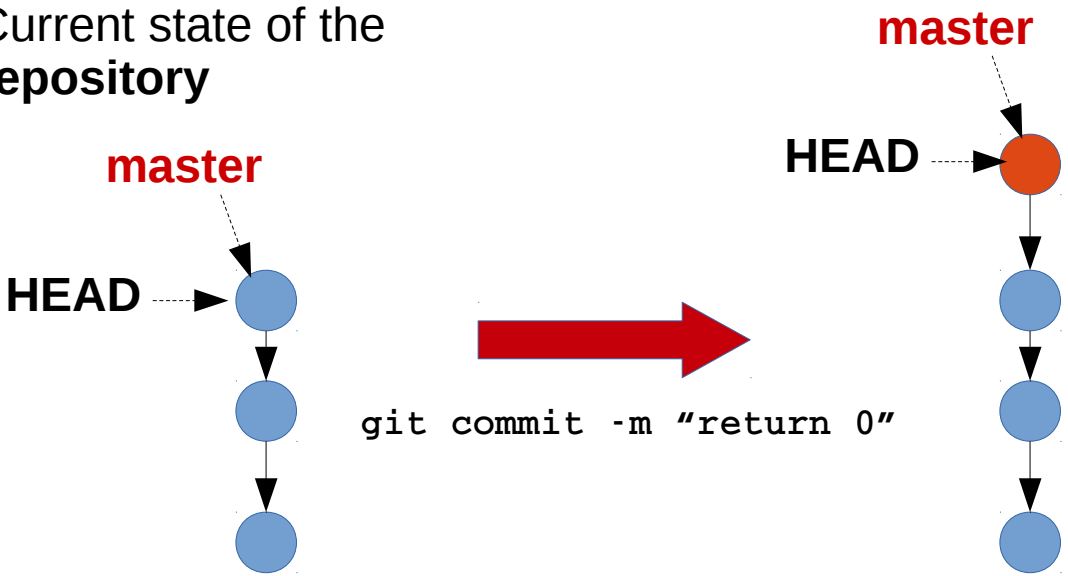
GIT Concepts

- **Committing** to the local repository
  - creating a new commit from all modifications registered in the staging area.
  - Adding a message explaining what has been modified.

Modifications registered in **staging area**

```
add other.cpp
in main.cpp
in main.cpp

cout<<"hello"<<endl;
+ return 0;
}
```

Current state of the **repository**

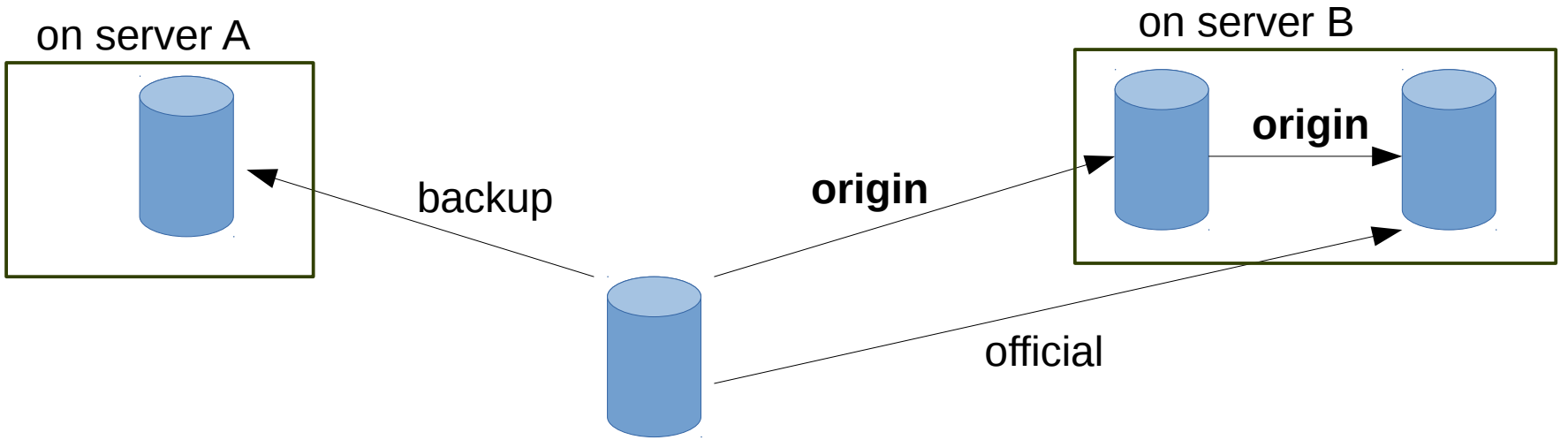


GIT Concepts

- Why two steps committing with **staging** ?
  - create “little” commits from a big bunch of modifications in the working folder.
  - delay the committing of some modifications: validated part of the code are committed while other are not.
  - isolate non committed code (e.g. debug traces).
- Once interesting modifications are committed, cleaning the staging area and the working folder is possible:
  - Using `git reset --hard`: permanent, code cannot be restored
  - Using `git stash save`: can be restored

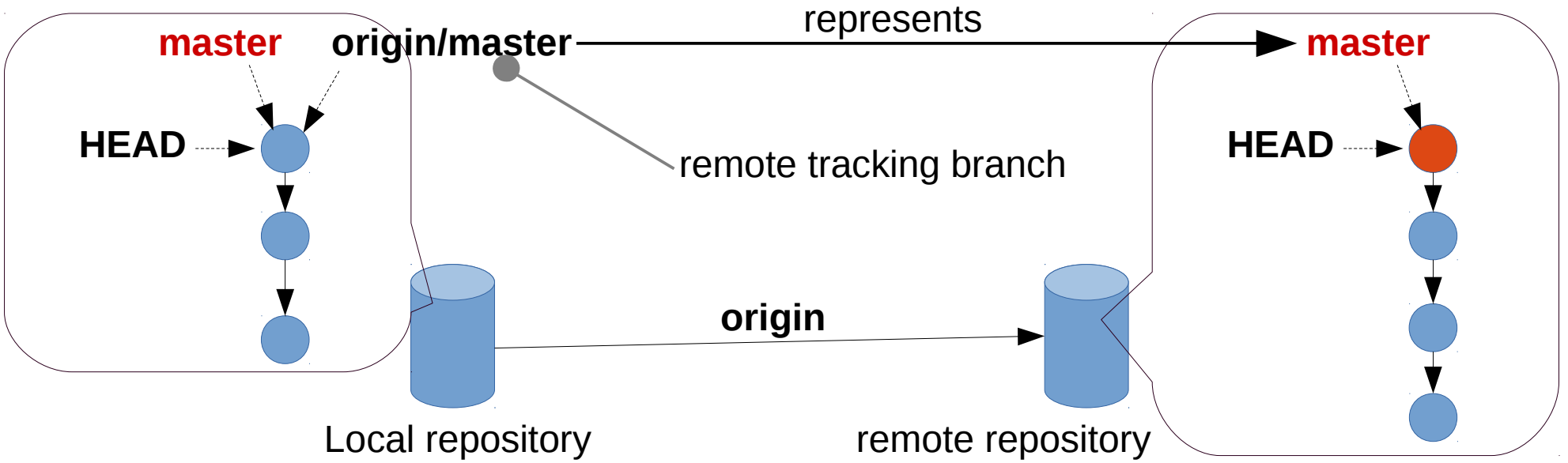
GIT Concepts

- Synchronization with **remote** repositories
  - Each repository knows a set of **remotes**.
  - The default remote is named **origin**.
  - Basic information on a remote is its **url** and its **name** (unique in the context of local repository).
  - Adding a remote : `git remote add backup <address>`
  - Removing a remote : `git remote rm backup <address>`



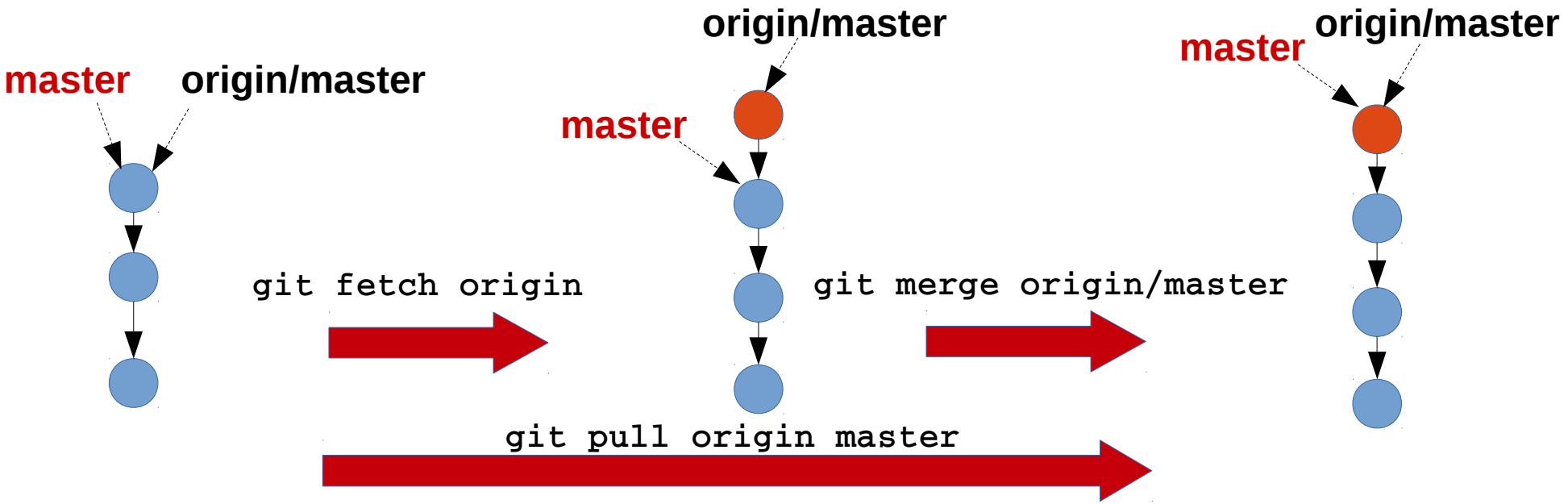
GIT Concepts

- Synchronizing local branches with **remote** branches
  - All branch of the remote are known in local repository but they are **read only** (impossible to directly commit to them). They are called **remote tracking branches**.
  - By default a local branch *tracks* the branch **with the same name** in its remotes (e.g. local master branch *tracks* origin master branch).



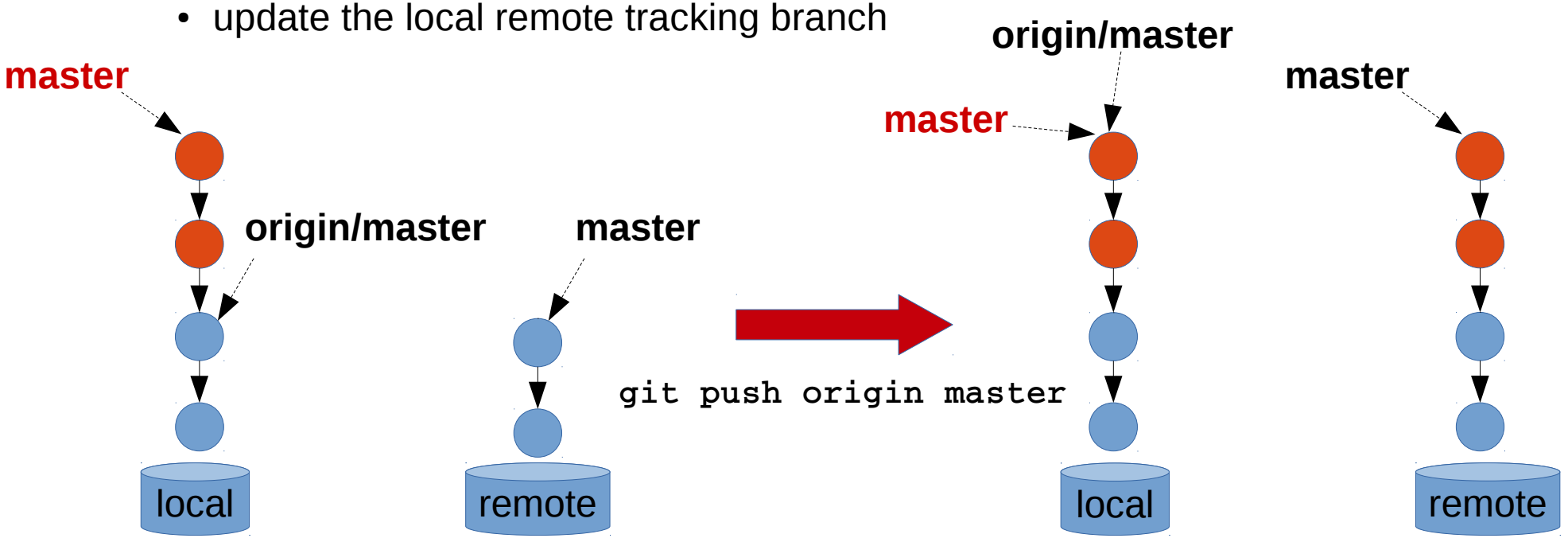
GIT Concepts

- Updating local repository from a remote
  - Achieved with the `fetch` command.
    - graph of commits is updated.
    - remote tracking branches are updated.
  - Then local branches can be **merged** with remote tracking branches
  - All in one command: `git pull`



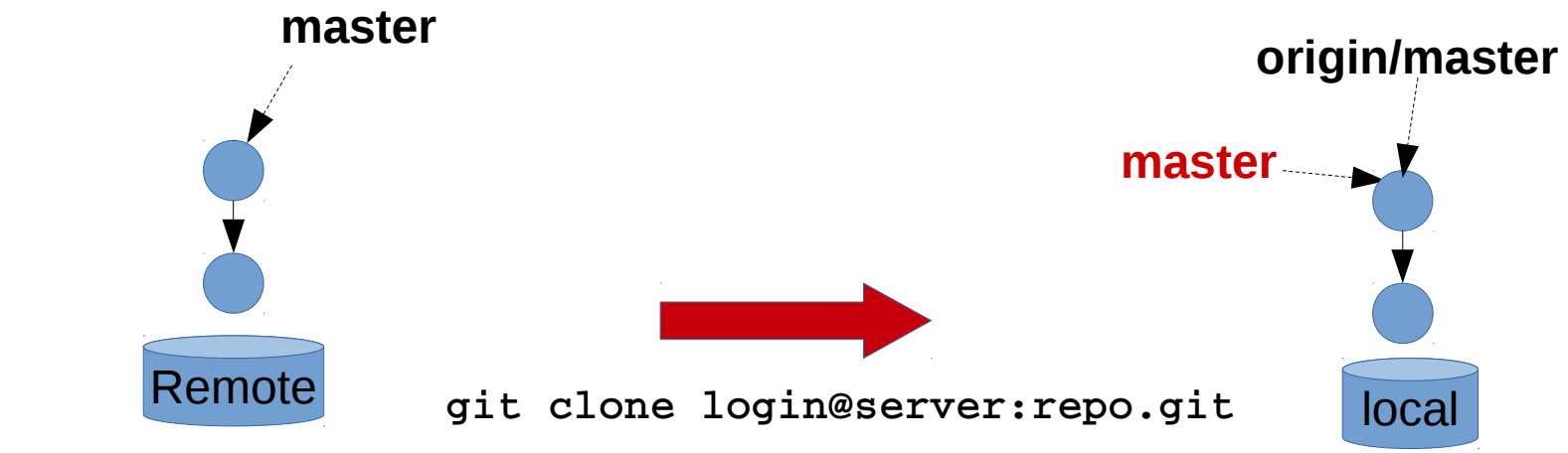
GIT Concepts

- Updating a remote branch from a local branch
  - Achieved with the `push` command (atomic operation).
    - check if local repository's remote tracking branch is up to date.
    - update the graph of commits of the remote and update the tracked branch of the remote.
    - update the local remote tracking branch



GIT Concepts

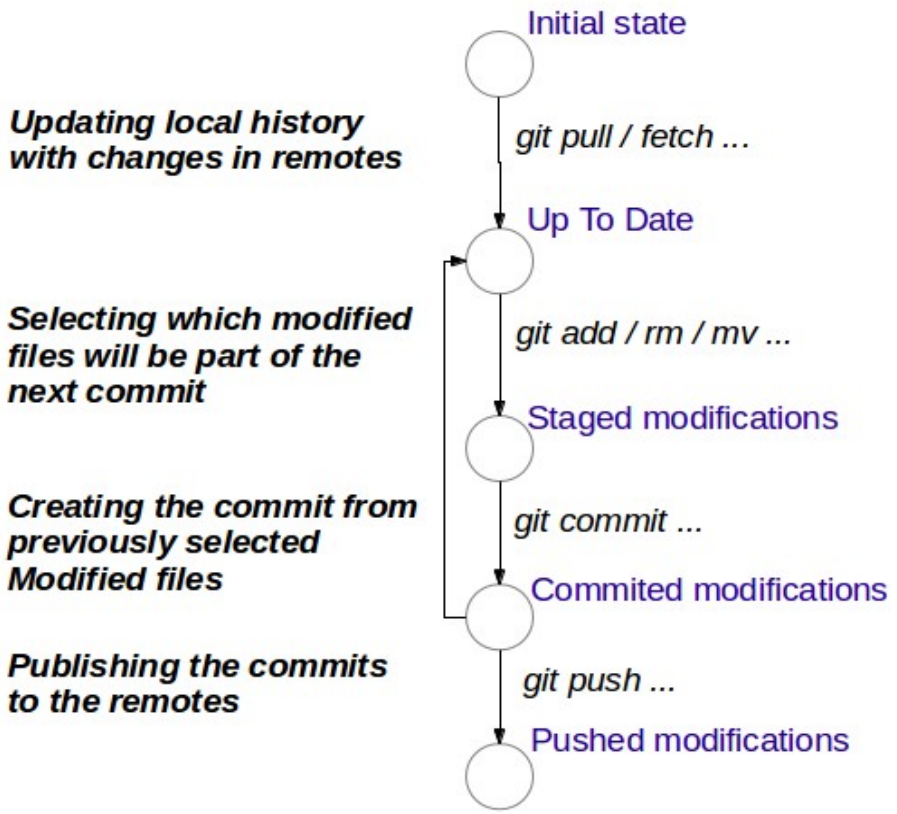
- Most known operation on remotes: **cloning**
  - Create a folder and initialize (`git init`) its repository.
  - Create a remote called origin (`git remote add origin <address>`)
  - Create a local master branch (`git checkout -b master`)
  - Update local master branch (`git pull origin master`).



Address: login@server:repo.git

GIT Concepts

- Typical usage

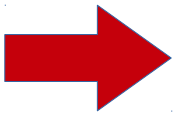


Plan

- Installation
- A brief history of version control systems
- GIT concepts
- **GITLAB Server**
- Step by step tutorial

GITLAB server

- A web application to ease the management of git projects
  - Easy creation of server repositories.
  - Easy registration of users.
  - Easy management of access right to projects for users.
  - Deep integration with git: graphical tools to visualize server side commits, branches, tags, user activities, files, etc.
  - Additional features to manage project documentation, to get statistics about projects, to easy team communication (issues).
  - Implement *Github-like workflow* based on **fork** and **merge requests**.



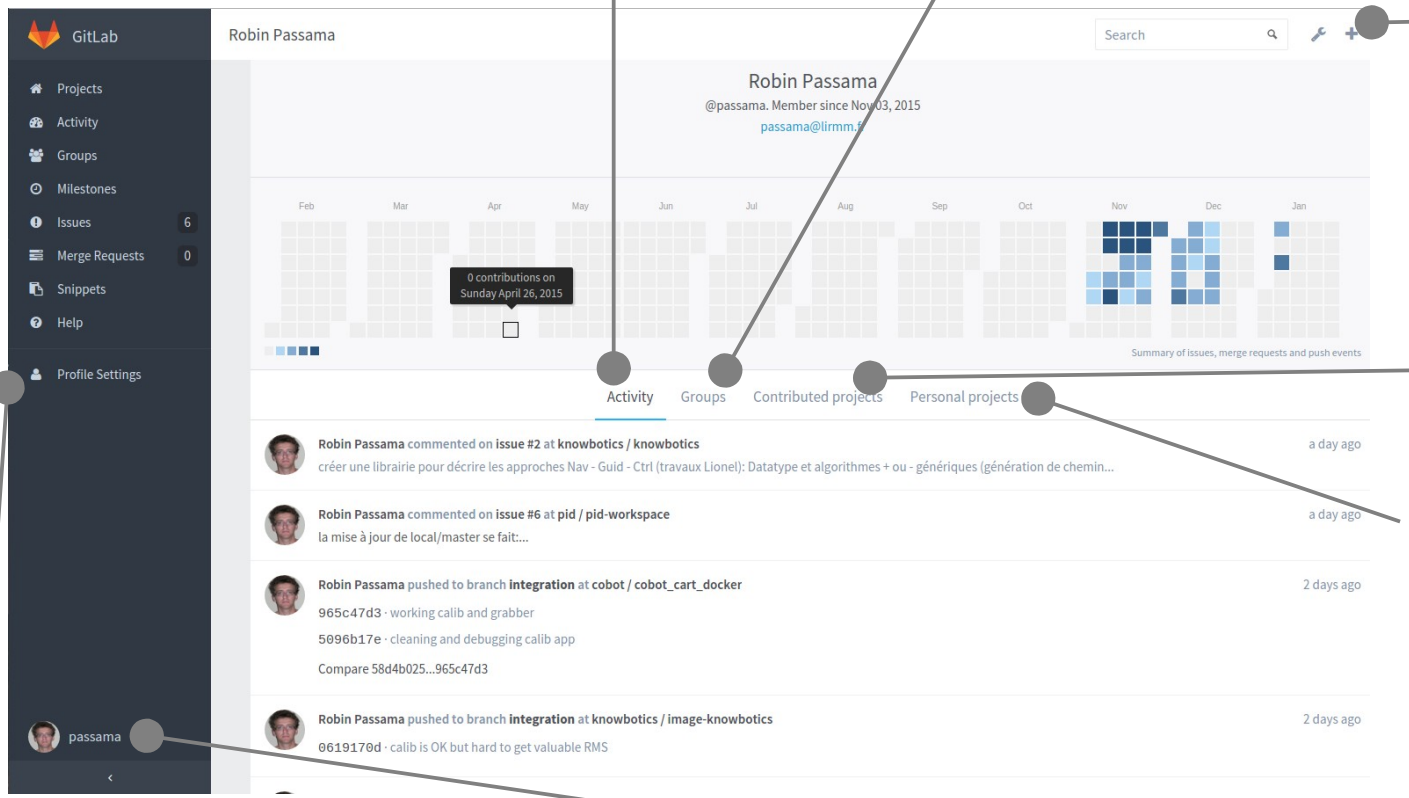
LIRMM Gitlab server: gite.lirmm.fr

GITLAB server

- Some definitions:
  - *Project*: a git repository + additional information managed by gitlab (access rights, wiki, issues, etc.).
  - *Workspace*: a place (folder on server) where to put projects (repositories).
  - *User*: a person registered in the server. Each user has a personal workspace.
  - *Group*: a community of users working on many related projects, with a specific workspace. Helps grouping related projects together.

GITLAB server

- User main page



Recent activities

Groups the user belongs to

Create new project

All projects user works on

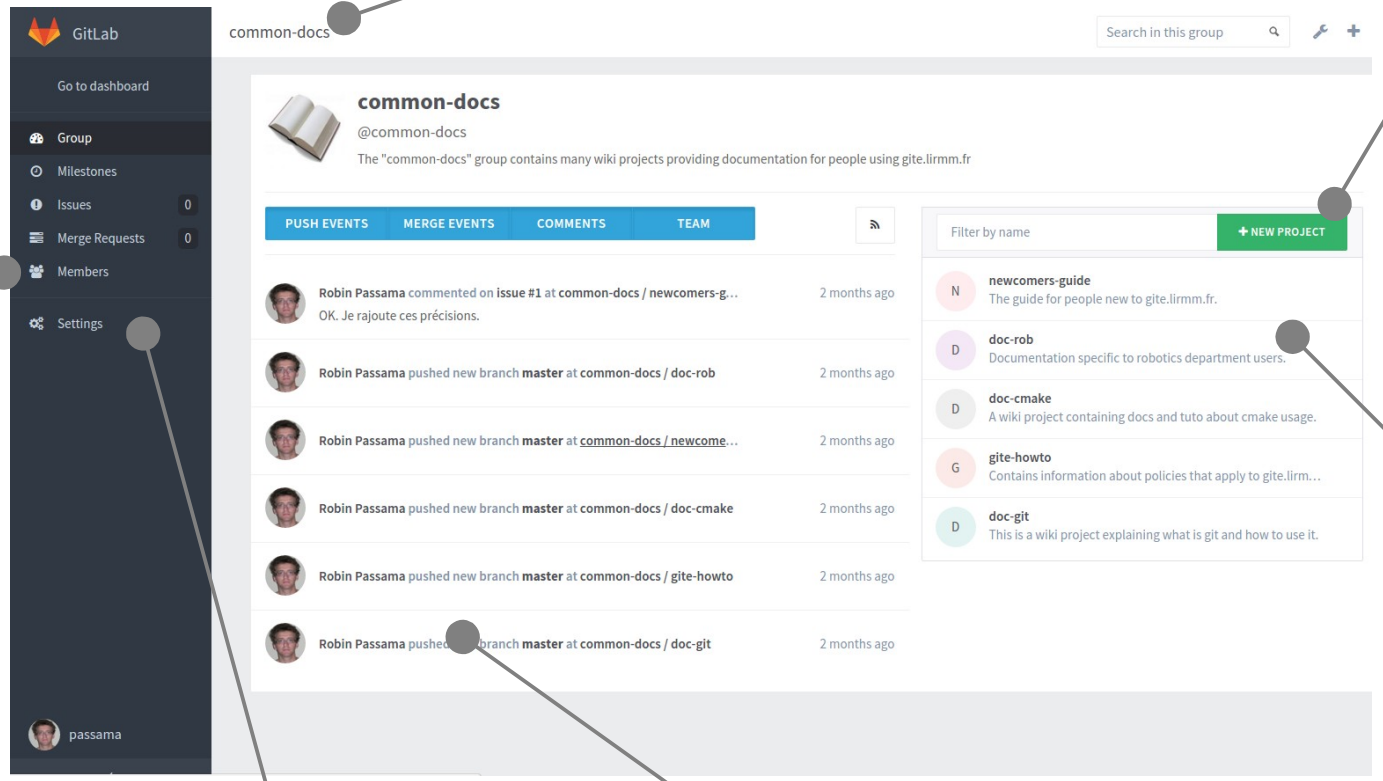
Projects in user's personal workspace

Configuration of user settings (SSH keys)

Direct access to this page (from anywhere)

GITLAB server

- Group main page



Workspace defined by the group

Create a new project in the group workspace

Projects in group workspace

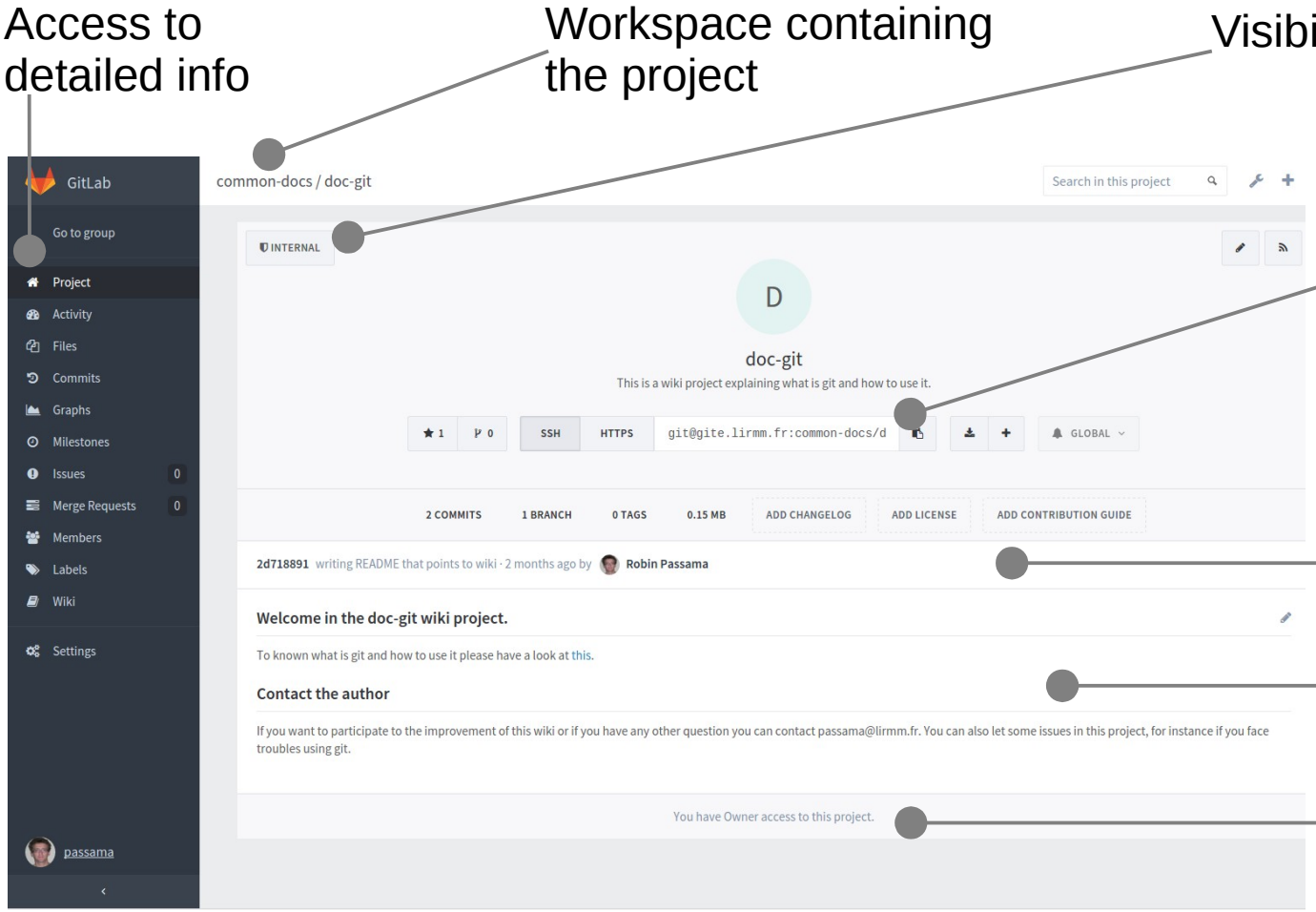
Group members management

General configuration of the group

Recent activities in group (include other users of the group)

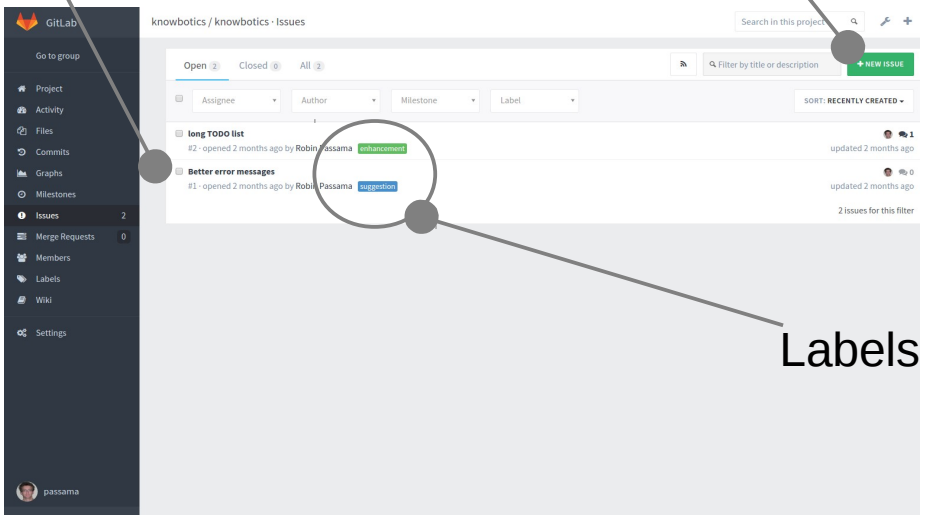
GITLAB server

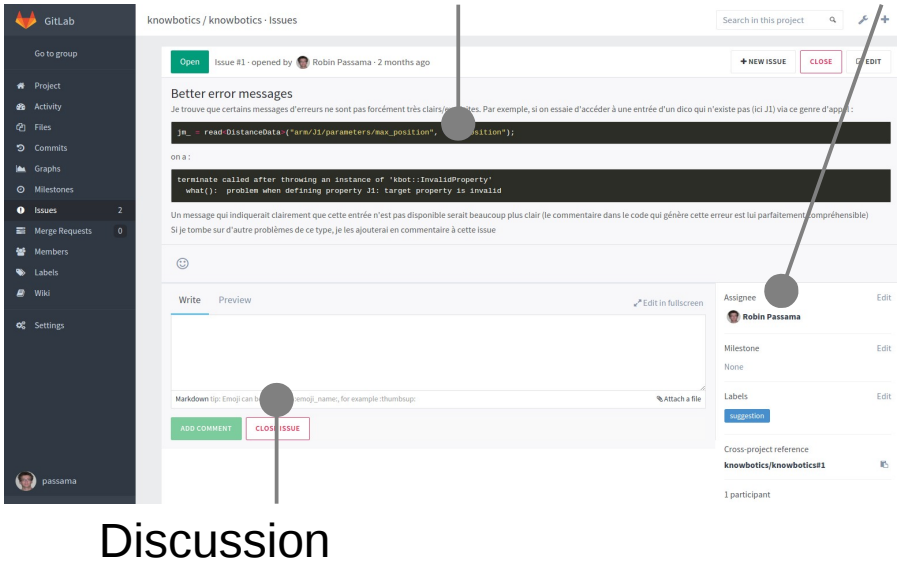
- Projects main page



GITLAB server

- Using *issues* to communicate
  - Declare BUGS, suggest some improvements, etc.
  - Open a discussion between users
  - Possibility to label issues to clearly categorize them (bug, suggestion, improvement, documentation, etc.)



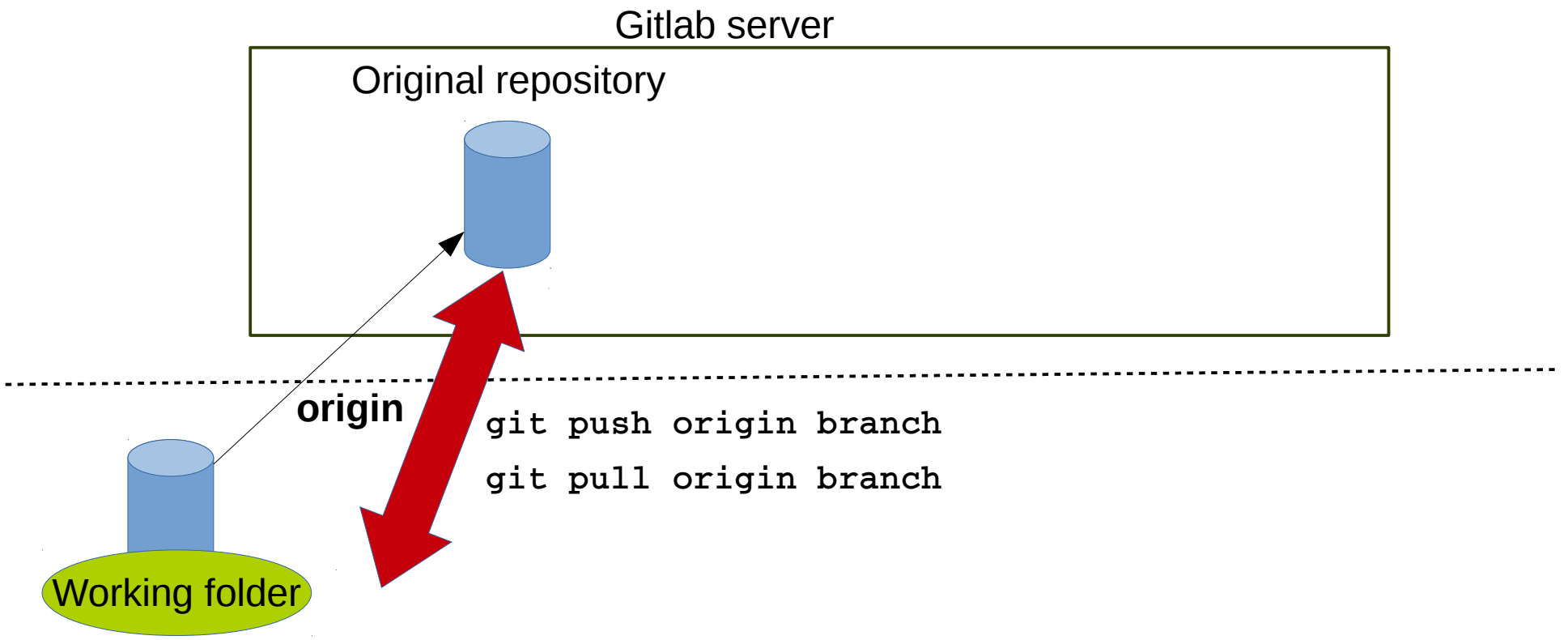


GITLAB server

- Github like work-flow
  - Based on ***Fork*** and ***merge request***.
  - Fork = cloning a repository directly in gitlab (server side repository)
    - Maintain relationship between original and clone repository
  - Merge request = proposing to **merge a branch of the clone repository** with a branch of the original repository.
    - Only developers of the original repository can do it.
    - Possibility to review proposed changes

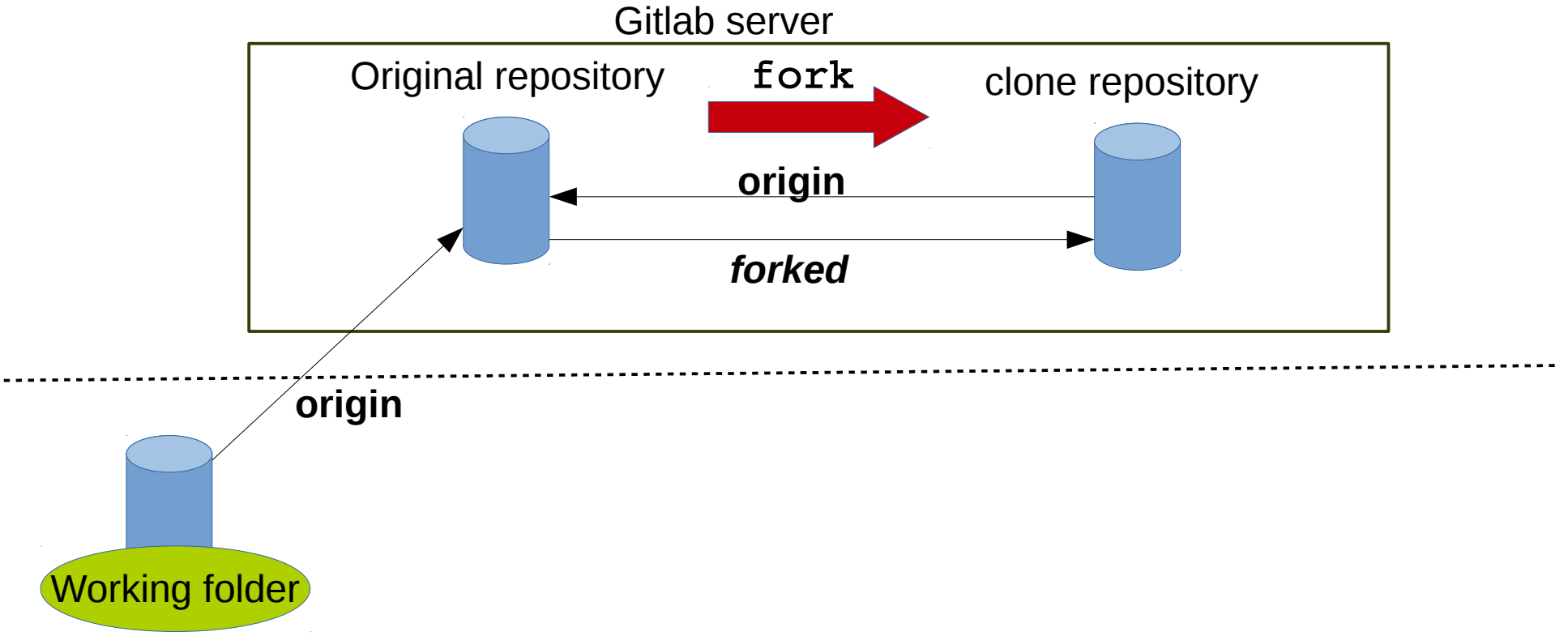
GITLAB server

- Github like work-flow



GITLAB server

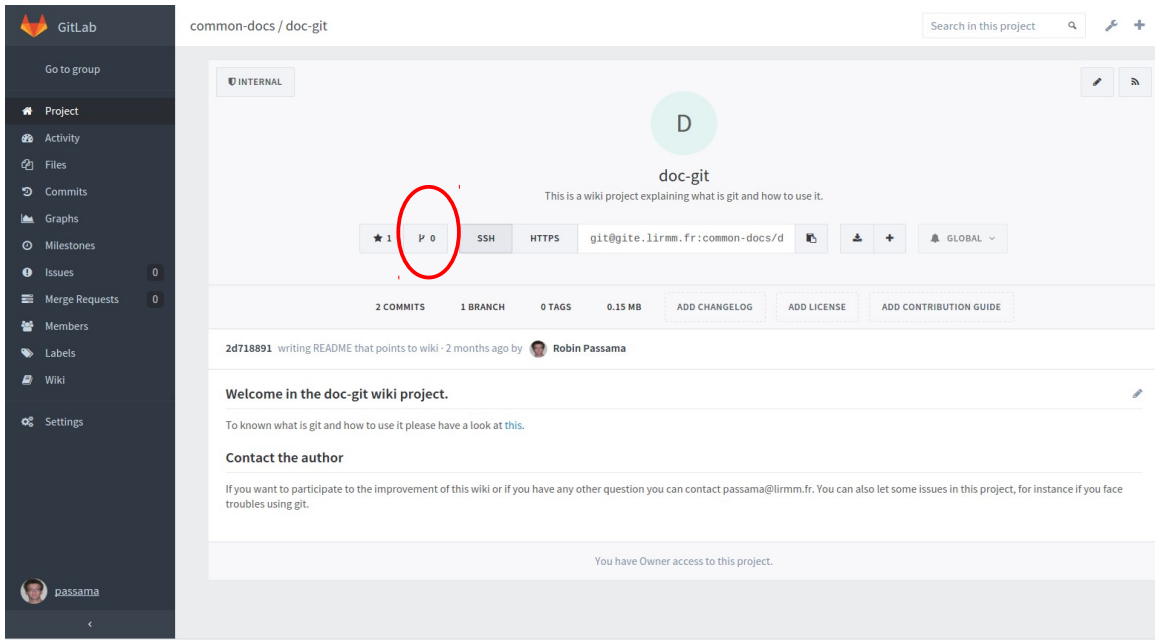
- Github like work-flow: forking
  - Clone repository is in another workspace



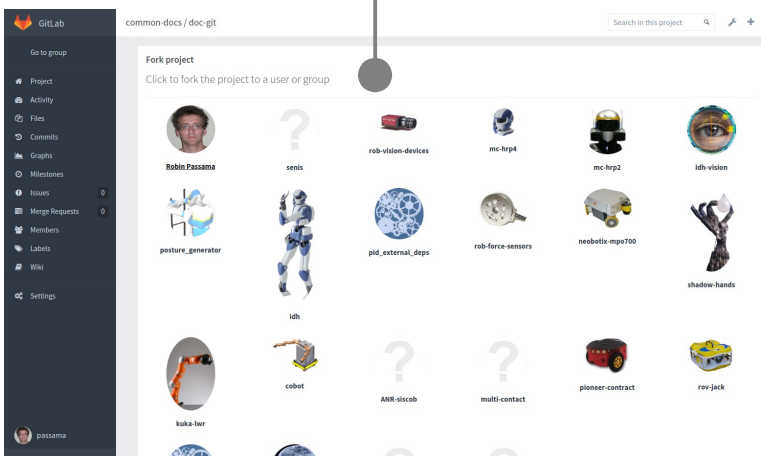
“Official” developers

GITLAB server

- Github like work-flow: forking
  - Clone repository is in another workspace

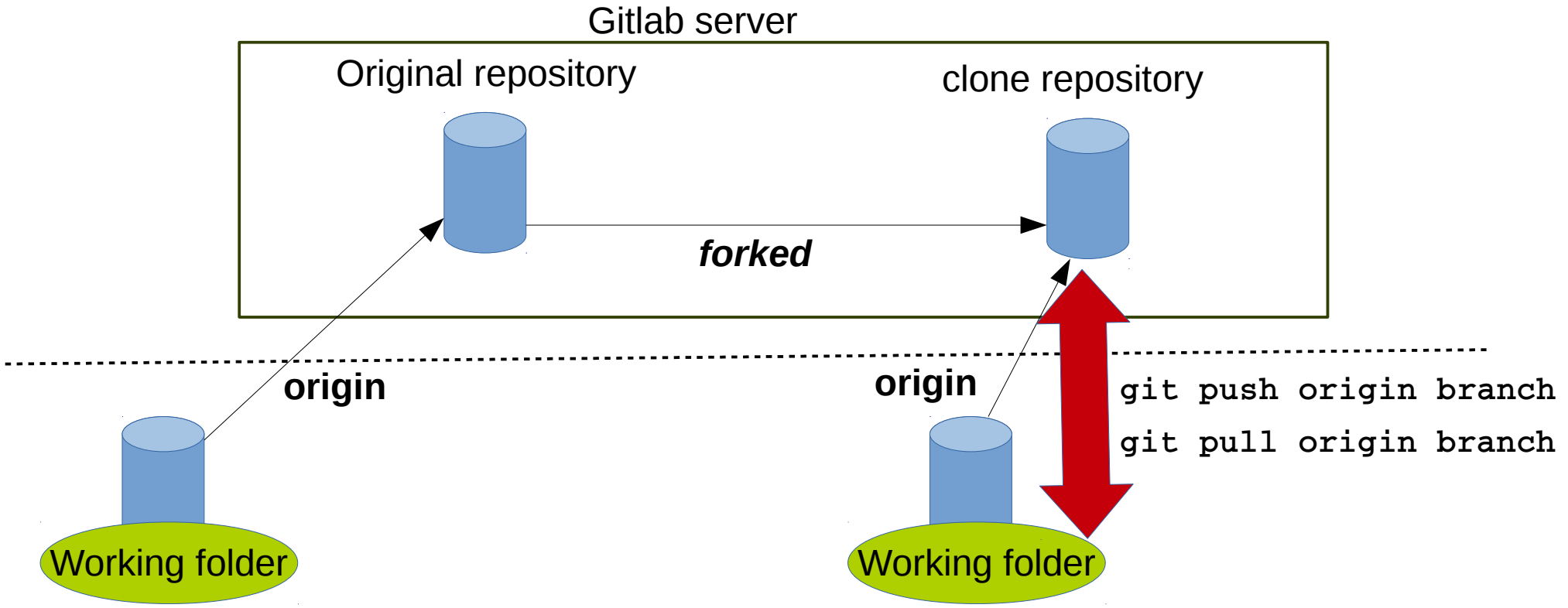


Select in which workspace (personal or group) the project will be cloned.



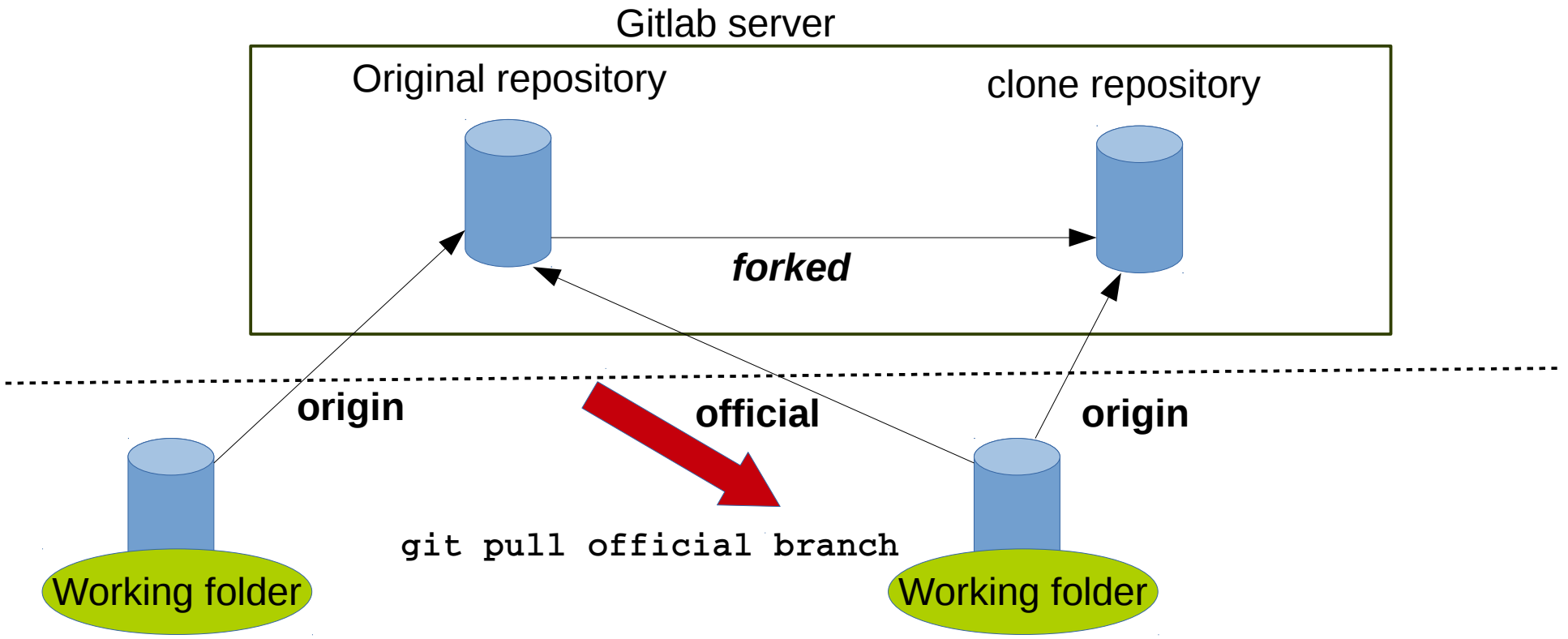
GITLAB server

- Github like work-flow : cloning
  - Isolated work for external developers



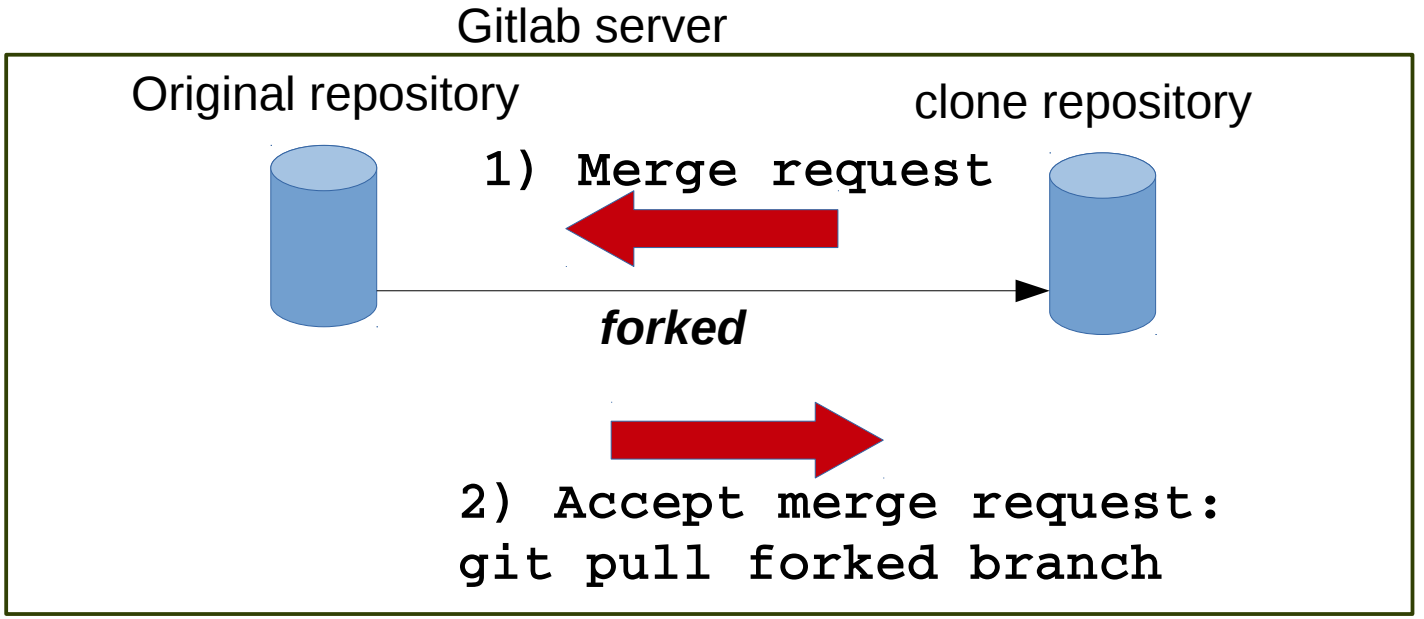
GITLAB server

- Github like work-flow: updating external repositories



GITLAB server

- Github like work-flow: merge request
  - Developers of *clone* decide to propose a merging of a branch.
  - Developers of *original* decide if merge is performed.

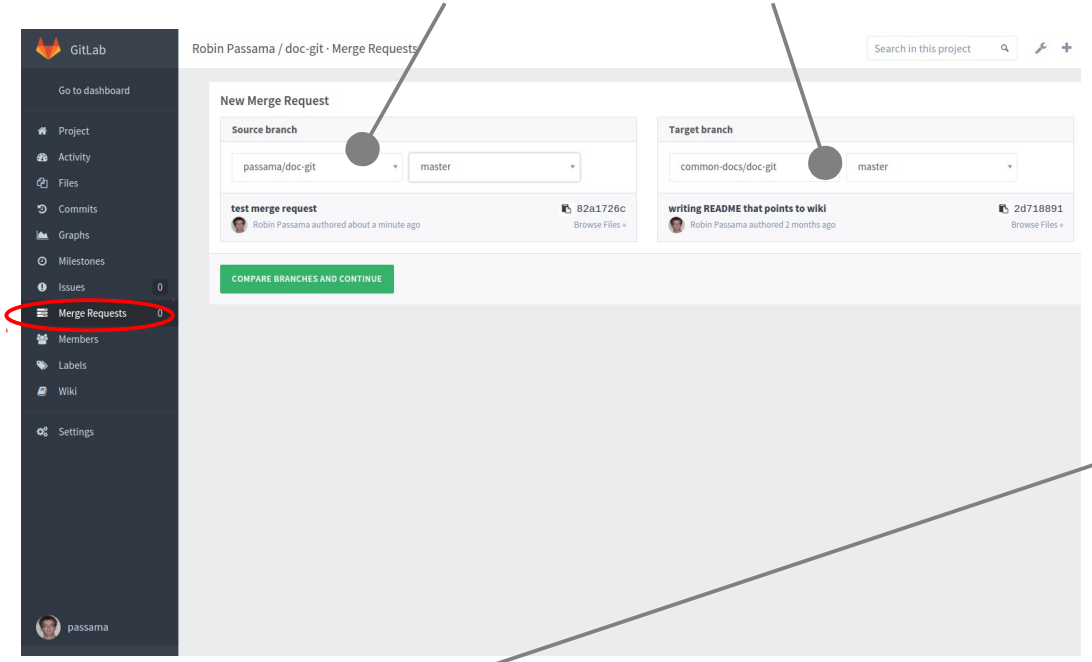


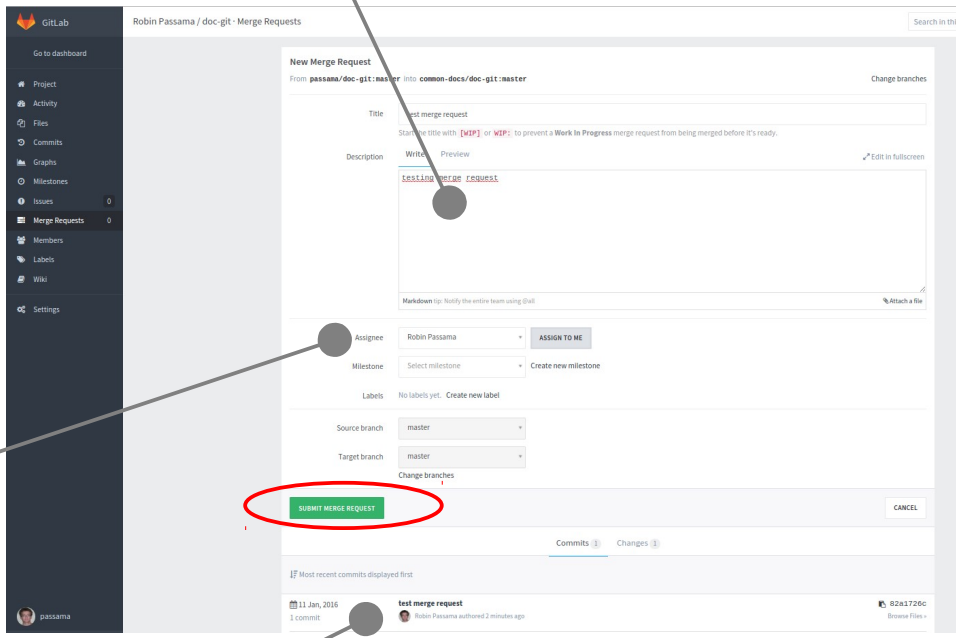
GITLAB server

- Github like work-flow: proposing a merge

Selecting source (from clone)
and target (from original) branches

Explaining the merge request



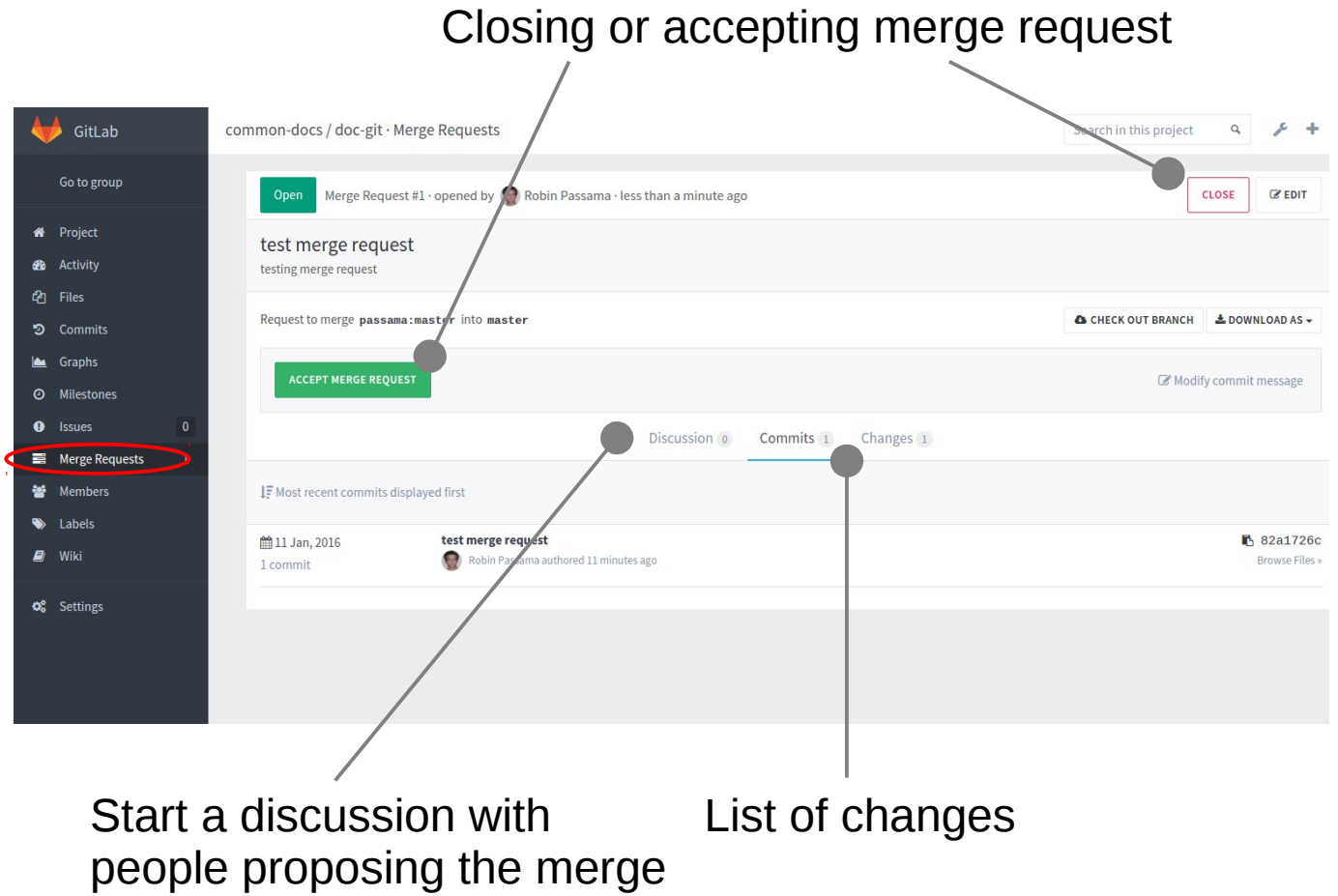


Developer from original repository in charge of managing the merge request

Commits proposed by the merge request

GITLAB server

- Github like work-flow: accepting a merge



Plan

- Installation
- A brief history of version control systems
- GIT concepts
- GITLAB Server
- **Step by step tutorial**
  - **A first project**
  - Collaborative work
  - Useful tips and advices

Tutorial: first project

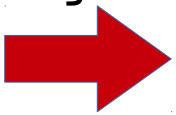
- In Gitlab:
  - Search for ***git-first-example*** in search bar.
  - Go to the project and get the address of the repository.

- In your workstation, open a terminal:
  - `cd <somewhere>`
  - `git clone git@gite.lirmm.fr:common-docs/git-first-example.git`
  - `cd git-first-example && ls -la`

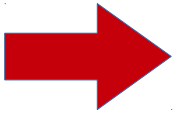
- Open README.md and look at its content

Tutorial: first project

- Listing branches:
  - **`git branch`**

    Only `master` branch appear (default local branch)
  - **`git branch -a`**

    `origin/master` and `origin/dev` branches also appear (you can find them on Gitlab)
  - The current branch is shown by an asterisk
- To automate the visualization of current branch
  - see last section of https://gite.lirmm.fr/common-docs/doc-git/wikis/tips:

```
parse_git_branch() ...
```

Tutorial: first project

- Working on **dev** branch
  - **`git checkout dev #change current branch`**
  - **`ls -la //there is one more file now`**
- Open README.md and look at its changed content
- Role of `.gitignore` file:
  - Exclude from version control files that match the pattern (here all file with a terminal '~').
  - Applies to sub-folders...
  - … But sub-folders may contain their own .gitignore.
- Git manage version of `.gitignore` file itself!

Tutorial: first project

- To visualize your local repository
  - `git log #text version`
  - `gitk #graphical tool`
- To visualize server repository
  - Click on “commit” menu of the project (left side panel), then:
  - Click on “Network” tab (~= gitk), or
  - Click on “Commits” tab (~= git log) after selecting the branch.

Tutorial: first project

- Check the status of your working folder
  - **`git status`**

    ```
    Sur la branche dev
    Votre branche est à jour avec 'origin/dev'.
    ...
    ```

  - Nothing to do for now...

Tutorial: first project

- Modify the content of README.md
  - Add text where you want ...
- Check again the status of your working folder:
  - **git status**

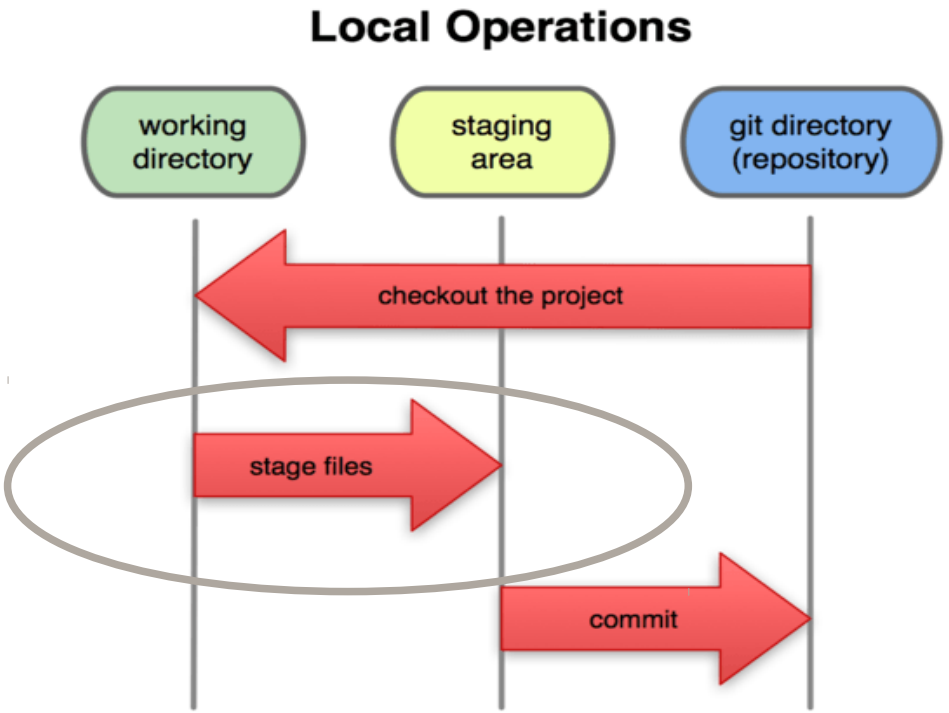
```
Sur la branche dev
Votre branche est à jour avec 'origin/dev'.


Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git checkout -- <fichier>..." pour annuler les modifications dans la copie de travail)

modifié:         README.md
aucune modification n'a été ajoutée à la validation (utilisez "git add" ou "git commit -a")
```

Tutorial: first project

- To see difference between last commit and your modifications
  - `git diff`
  - '+' indicates that a file has been added or content has been added into file.
  - '-' indicates that a file has been removed or content has been removed into file.

Tutorial: first project

- Now you need to select modifications
  - `git add -A #all modifications are staged`



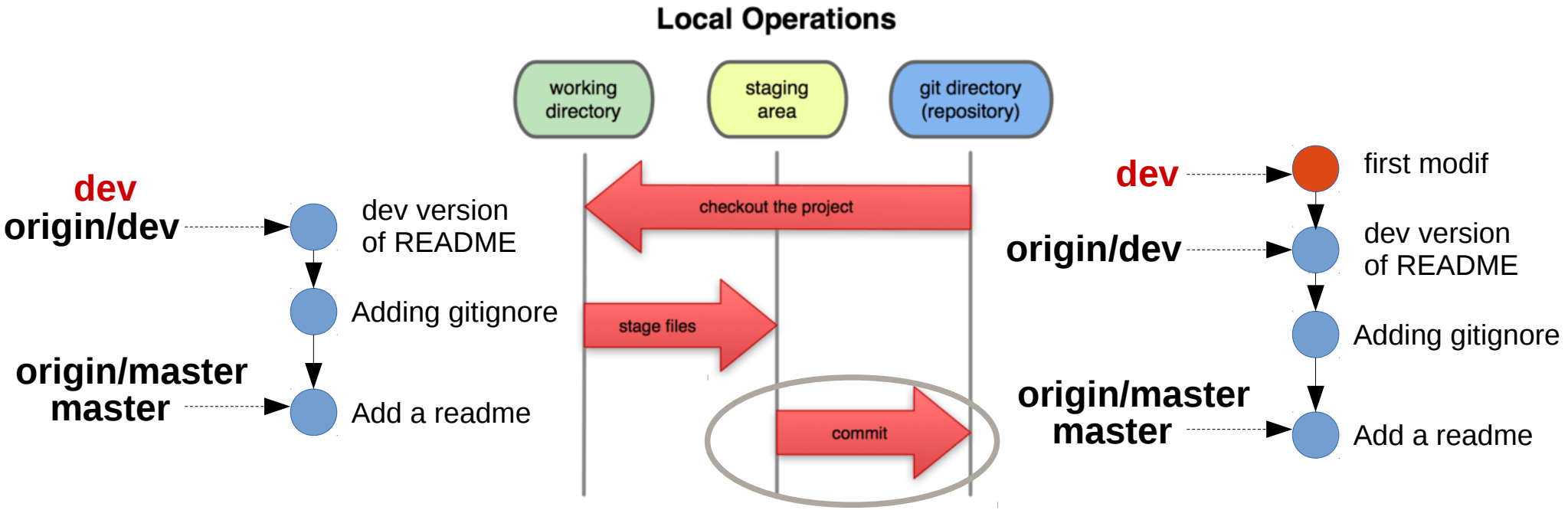
Tutorial: first project

- You need to stage your modifications
  - **`git add -A #all modifications are staged`**
- Check again the status of your working folder:
  - **`git status`**

```
Sur la branche dev
Votre branche est à jour avec 'origin/dev'.

Modifications qui seront validées :
  (utilisez "git reset HEAD <fichier>..." pour
désindexer)

modifié:         README.md
```

Tutorial: first project

- Now you need to commit your changes to your local repository



Tutorial: first project

- Committing your modifications
  - **`git commit -m “first modif” #all staged modifications are committed`**
- Check again the status of your working folder:
  - **`git status`**

```
Sur la branche dev
Votre branche est en avance sur 'origin/dev' de 1 commit.
  (utilisez "git push" pour publier vos commits locaux)

rien à valider, la copie de travail est propre
```

- Use `gitk` to see the new status of your repository

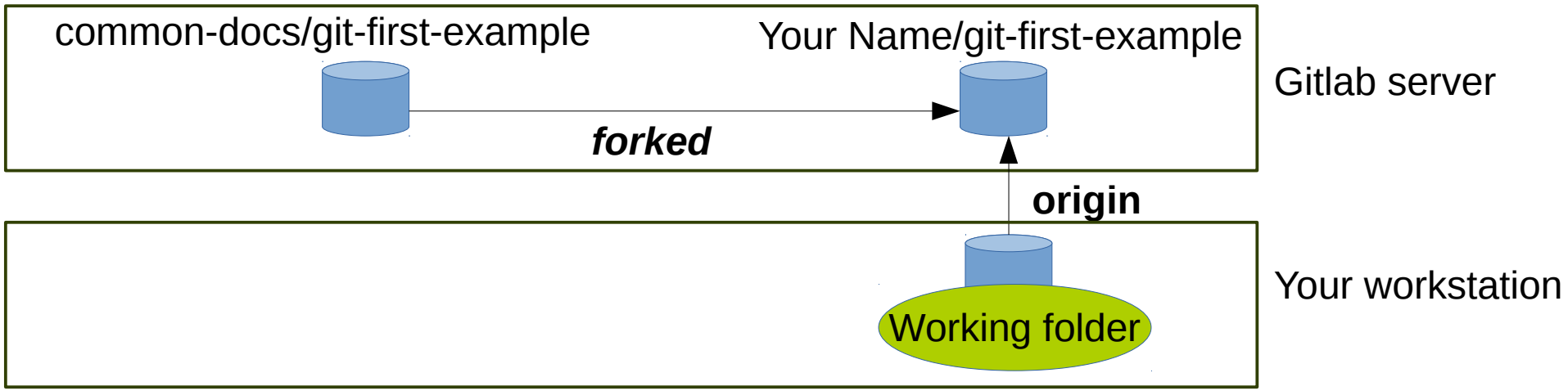
Tutorial: first project

- Publishing modifications to the server repository
  - `git push origin dev`

    FAILED !!!
  - **Normal situation: you simply do not have right to push !**
- Remedy: ***Fork*** the server repository into your personal workspace
  - You are owner of this new repository, **you can push to any branch**.
  - Copy the address of the clone server repository.

Tutorial: first project

- Change the origin of your local repository
  - `git remote set-url origin <address of the clone repository>`
- Verify the change
  - `git remote -v`
- New architecture



Tutorial: first project

- Again, publishing modifications to the server repository
  - **`git push origin dev`**

    Now it works !

```
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 334 bytes | 0
bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To git@gite.lirmm.fr:passama/git-first-example.git
   0ca3e2d..321b394  dev -> dev
```

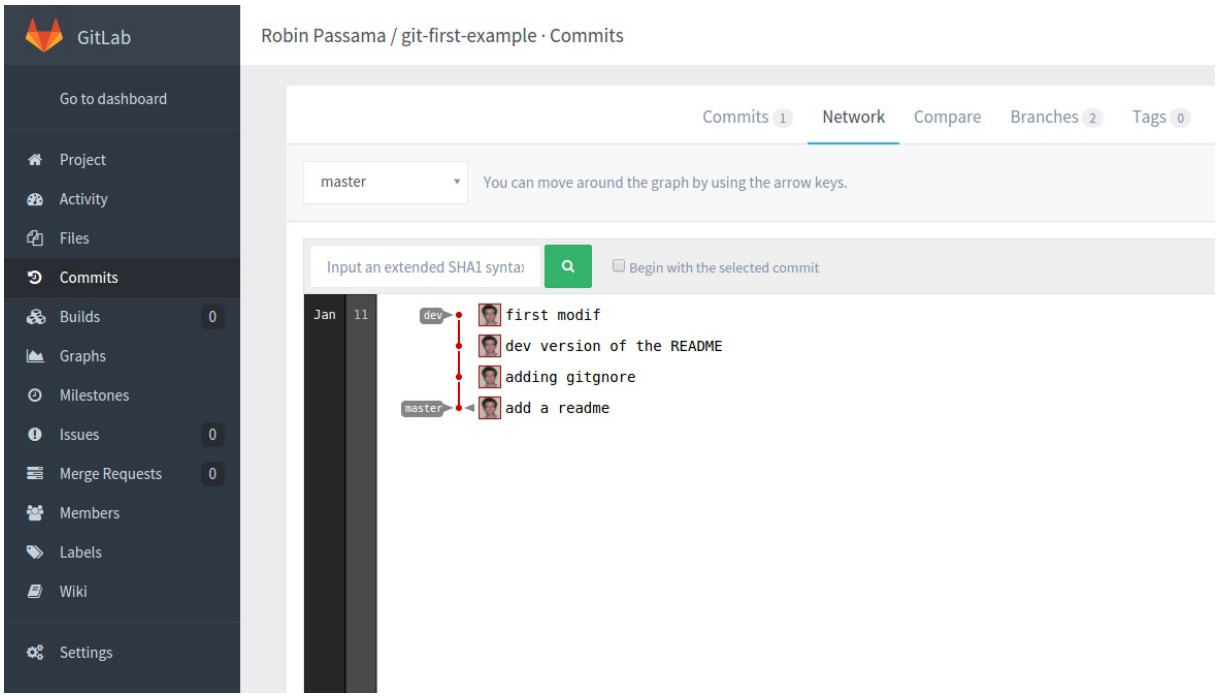
Target remote repository

Local (source) branch

remote (updated) branch

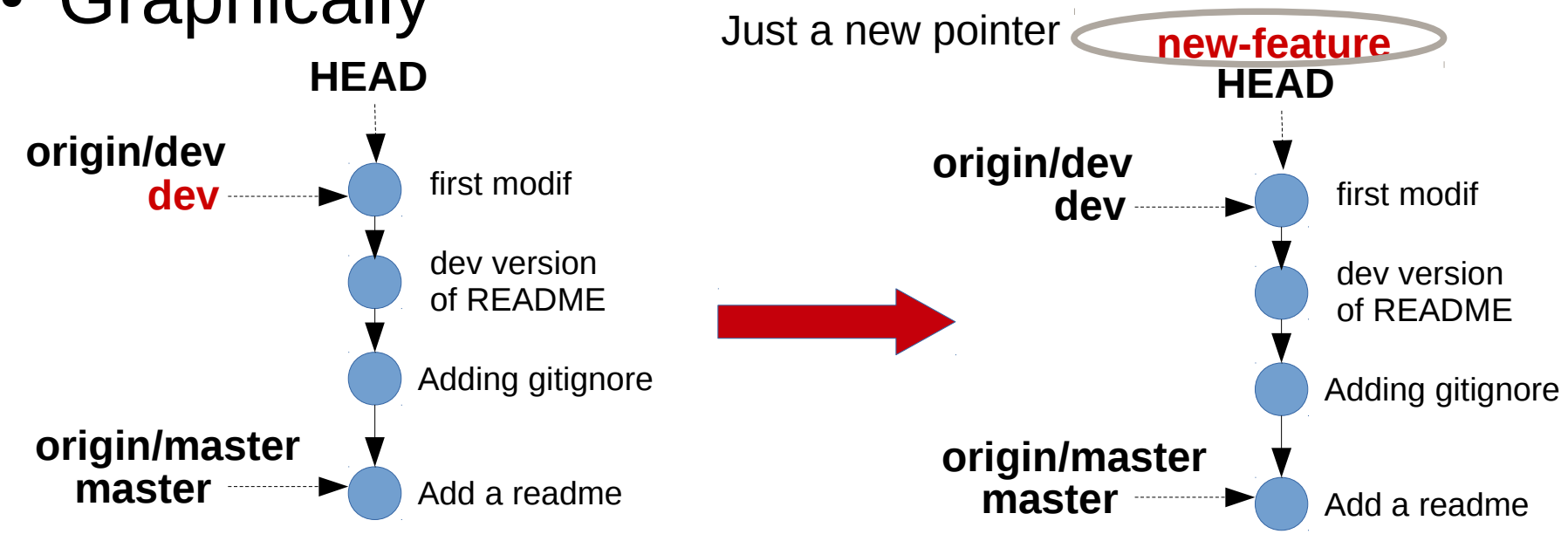
Tutorial: first project

- Check modifications in gitlab
  - Go to “commits” menu > “Network” tab
- You should see something like:



Tutorial: first project

- Now create a new branch
  - `git checkout -b new-feature`
  - `git branch #current branch has changed`
- Graphically



Tutorial: first project

- Create new content
  - **`mkdir dir && nano dir/newfile`**
    - Input some text and write **`newfile`** (Ctrl+O, Enter, Ctrl + X)
  - **`nano dir/otherfile`**
    - Input some text and write **`otherfile`** (Ctrl+O, Enter, Ctrl + X)
- Check the status of your working folder:
  - **`git status`**

```
Sur la branche new-feature
Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)

dir/

aucune modification ajoutée à la validation mais des fichiers non suivis sont présents (utilisez "git add" pour les suivre)
```

Tutorial: first project

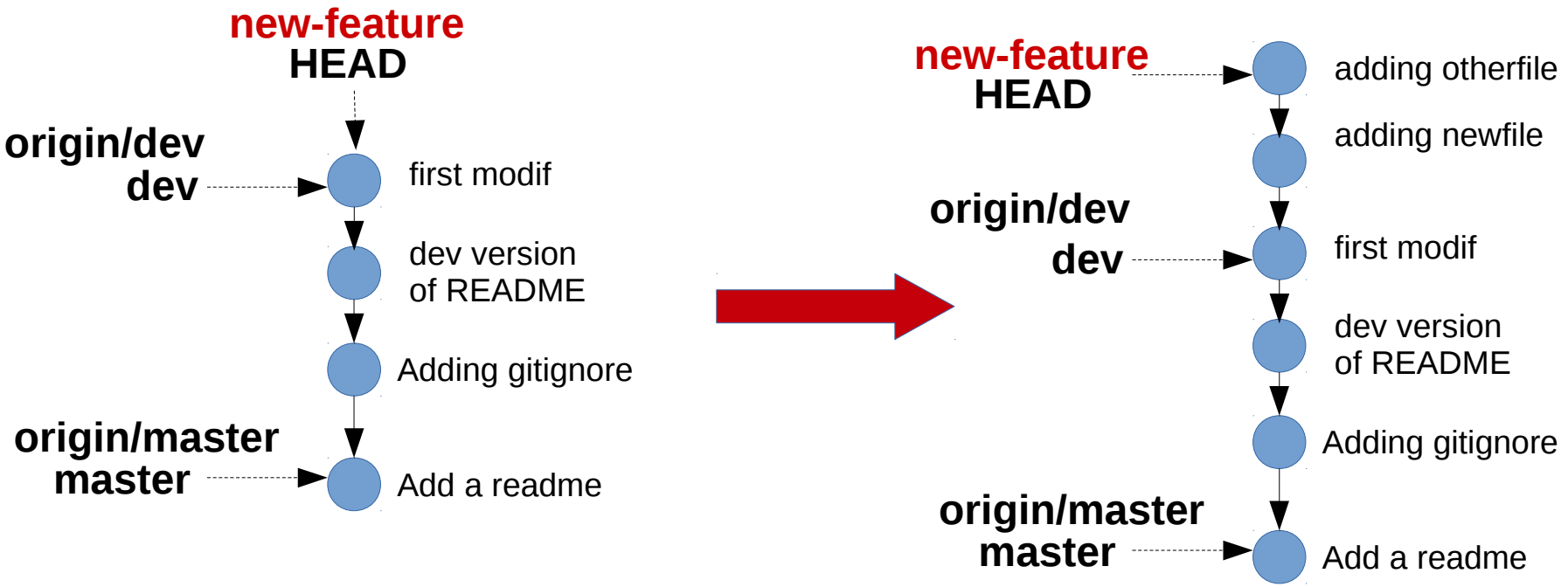
- Let's suppose you want to separate your modifications **into 2 commits**
  - **git add dir/newfile**
  - **git status**

```
Sur la branche new-feature
Modifications qui seront validées :
  (utilisez "git reset HEAD <fichier>..." pour désindexer)

nouveau fichier: dir/newfile

Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)

dir/otherfile
```

  - **git commit -m “adding newfile”**

Tutorial: first project

- Let's suppose you want to separate your modifications **into 2 commits**
  - `git add dir/otherfile`
  - `git commit -m “adding otherfile”`



Tutorial: first project

- Creating 2 commits from two modifications
  - Edit `newfile` and input 2 new lines one at the beginning, one at the end of the file
  - **`git add -p`**

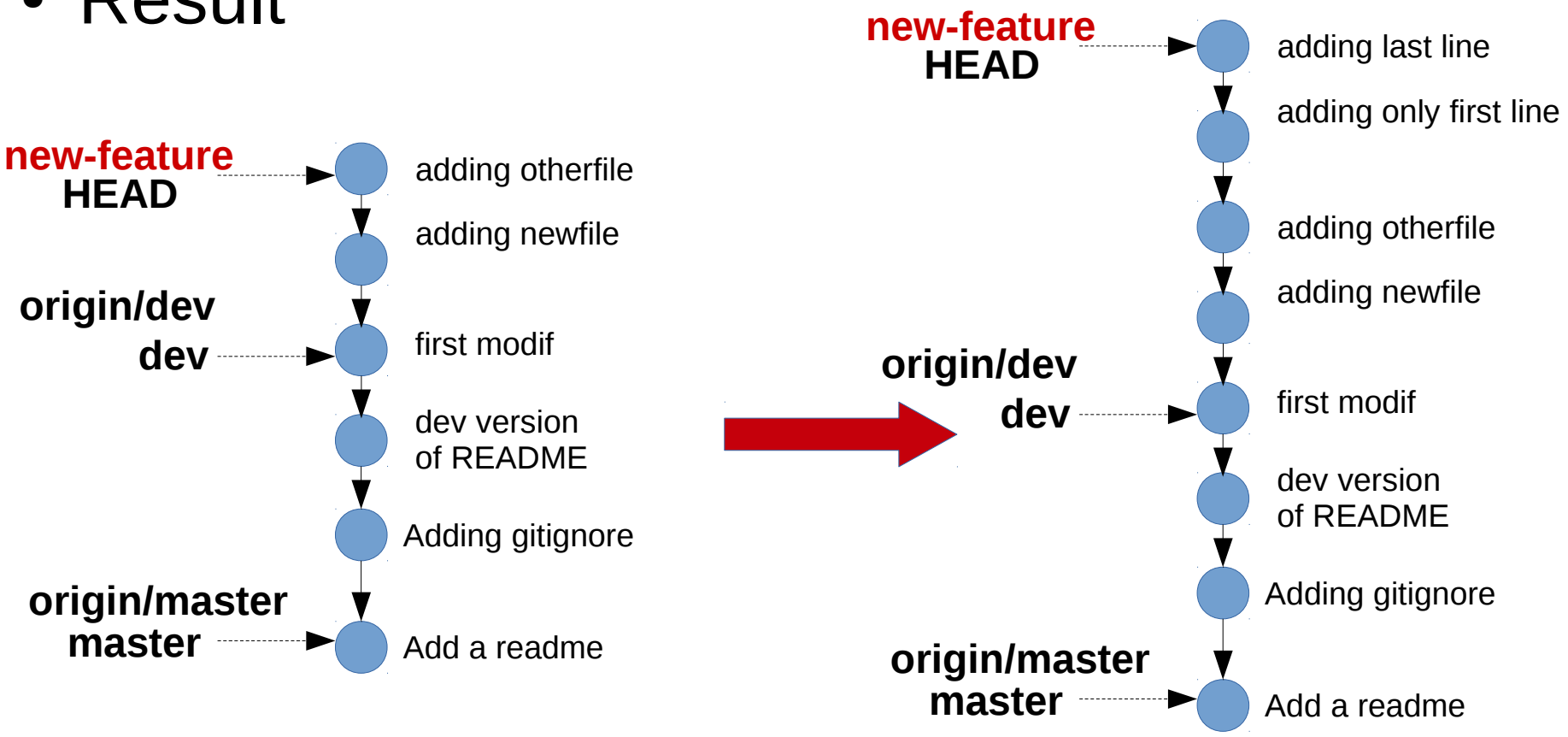
    Select **`'y'`** for first line modification

    Then select **`'n'`** for last line modification
  - **`git commit -m “adding only first line”`**
  - **`git add -p`**

    Select **`'y'`** for last line modification
  - **`git commit -m “adding last line”`**

Tutorial: first project

- Result



Good practice: by default use `git add -p` to check modifications you will commit

Tutorial: first project

- Undo a sequence of commit
  - Finally, we want to remove the two last commits...
  - **`git log`**
    - Copying the SHA1 identifier of the commit preceding these two commits.
  - **`git reset <SHA-1-ID>`**

```
Modifications non indexées après reset :
M  dir/newfile
```

  - **`git status`**

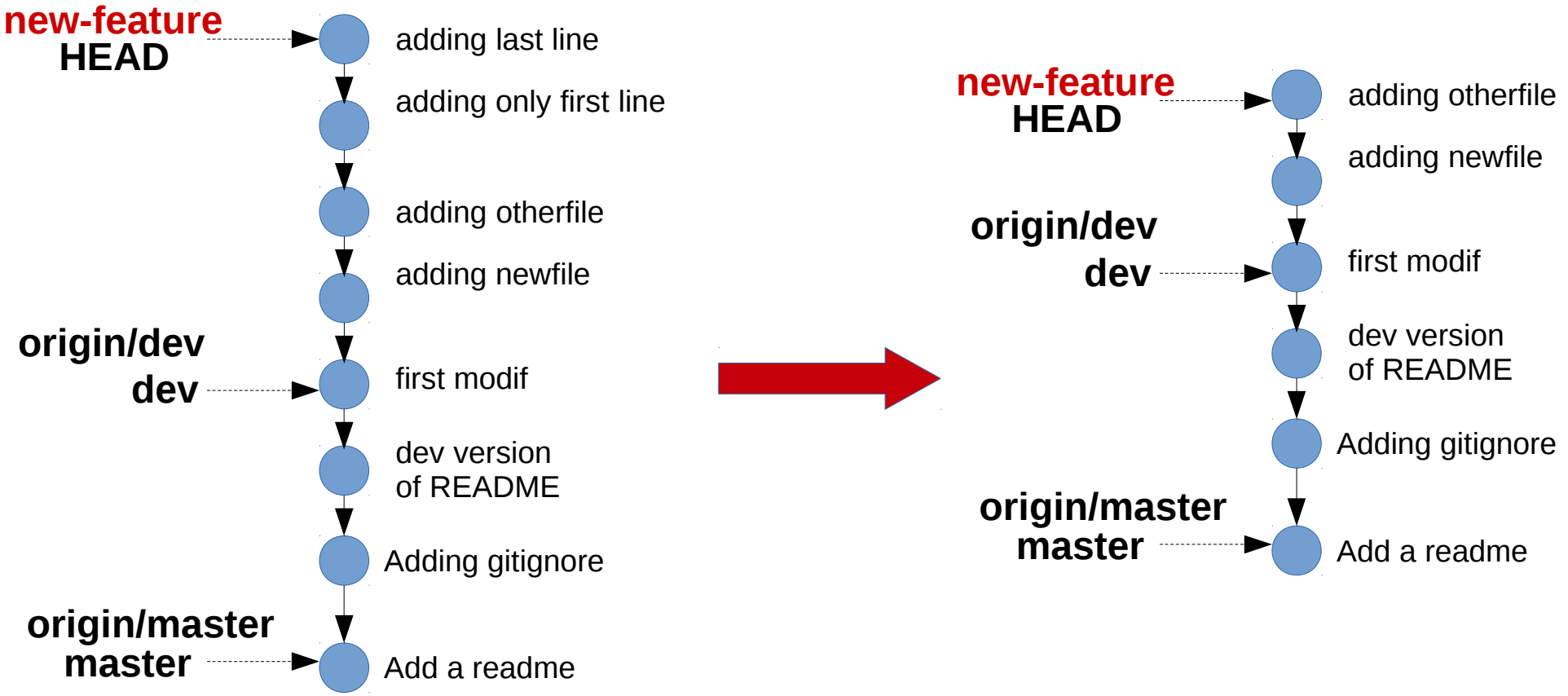
```
Modifications qui ne seront pas validées :
modifié:          dir/newfile
```

    - Modifications contained in the removed commits **are now again in working folder**

WARNING: Never use git reset on a published content (only local commits not pushed)

Tutorial: first project

- Result



Tutorial: first project

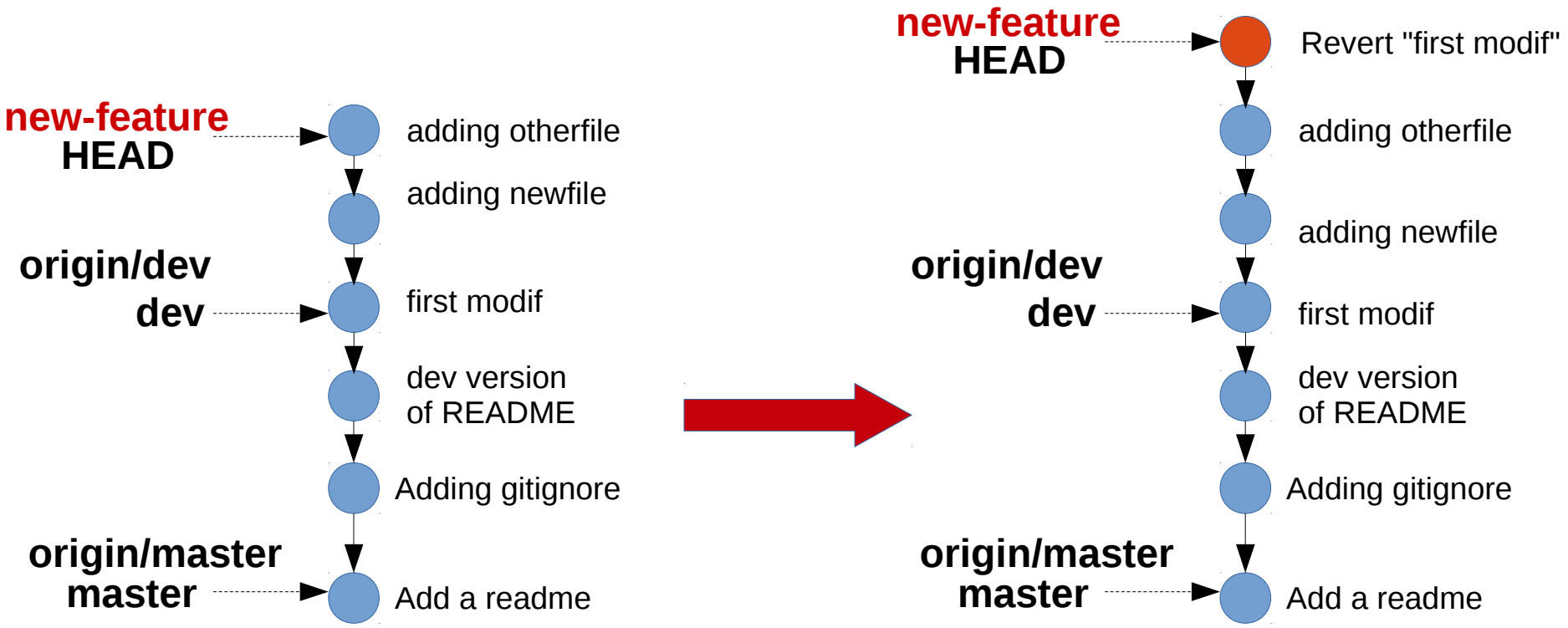
- Get rid of these modifications
  - How can I remove these modifications from working folder ? 2 solutions:
  - **`git reset --hard <SHA-1-ID> #use --hard in previous command`**
    - Definitive cleaning 
  - **`git stash`** (or **`git stash save`**)
    - Modifications contained in working folder **are put in a temporary commit object and cleaned from working folder**. Then you can do either**:**
    - Reapply saved changes to working folder and delete the stash:
      - **`git stash pop`**
    - Forget all stashed changes (definitive)
      - **`git stash clear`**

Tutorial: first project

- Undo a commit
  - Finally, we want to undo modification made in commit **“first modif”**.
  - **`git log`**
    - Copying the SHA1 identifier of the commit to undo.
  - **`git revert <SHA-1-ID>`**
    - A new commit object is generated !

Tutorial: first project

- Result



Good practice: by default use git revert since it is not dangerous (can be done on published commits)

Tutorial: first project

- Let's suppose the new feature is finished, we want to update `dev` branch with it.
  - **`git checkout dev #go to dev branch`**
  - **`git merge new-feature`**

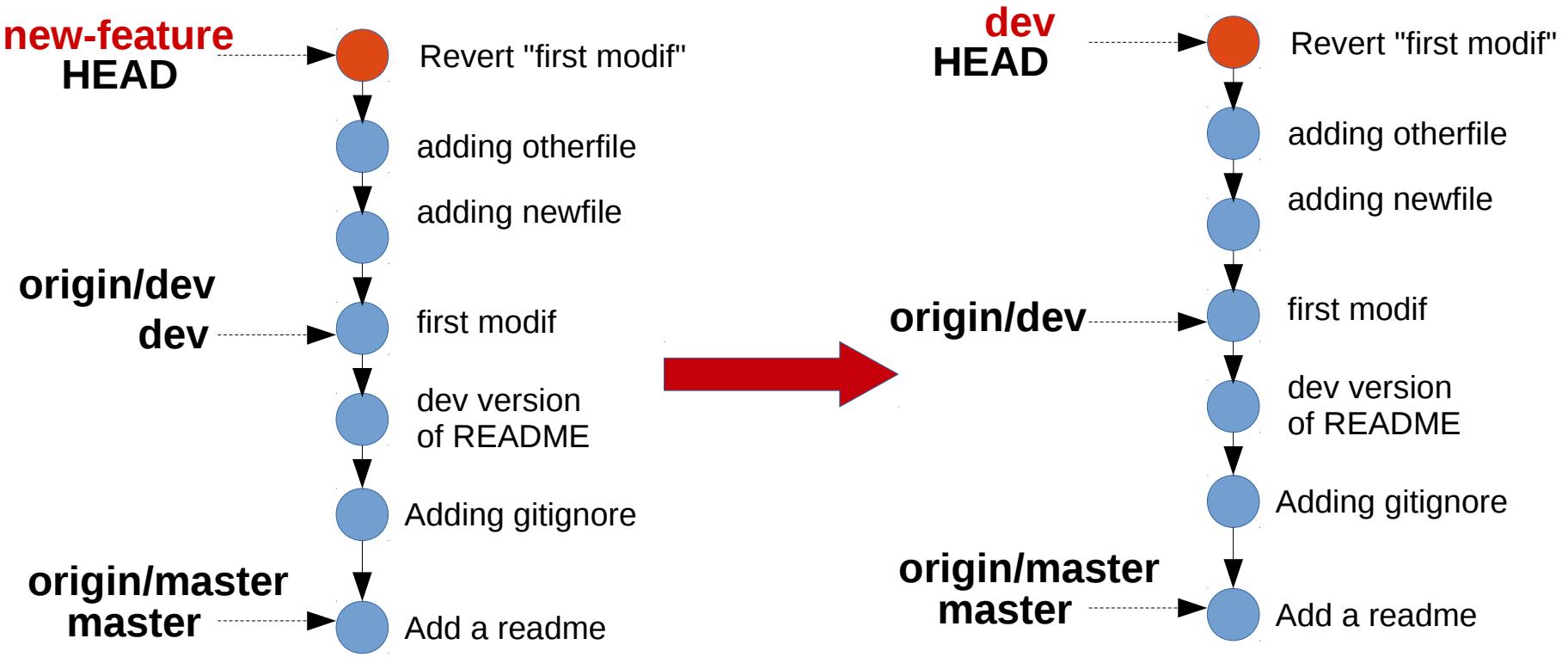
```
Mise à jour 321b394..2a77b12
Fast-forward
 README.md     | 1 -
 dir/newfile   | 7 +++++++
 dir/otherfile | 6 ++++++
 3 files changed, 13 insertions(+), 1 deletion(-)
 create mode 100644 dir/newfile
 create mode 100644 dir/otherfile
```

Sum up of all modifications since last commit previously pointed by `dev`

- Then we want to delete `new-feature` branch (no more useful)
  - **`git branch -D new-feature`**

Tutorial: first project

- Result



Tutorial: first project

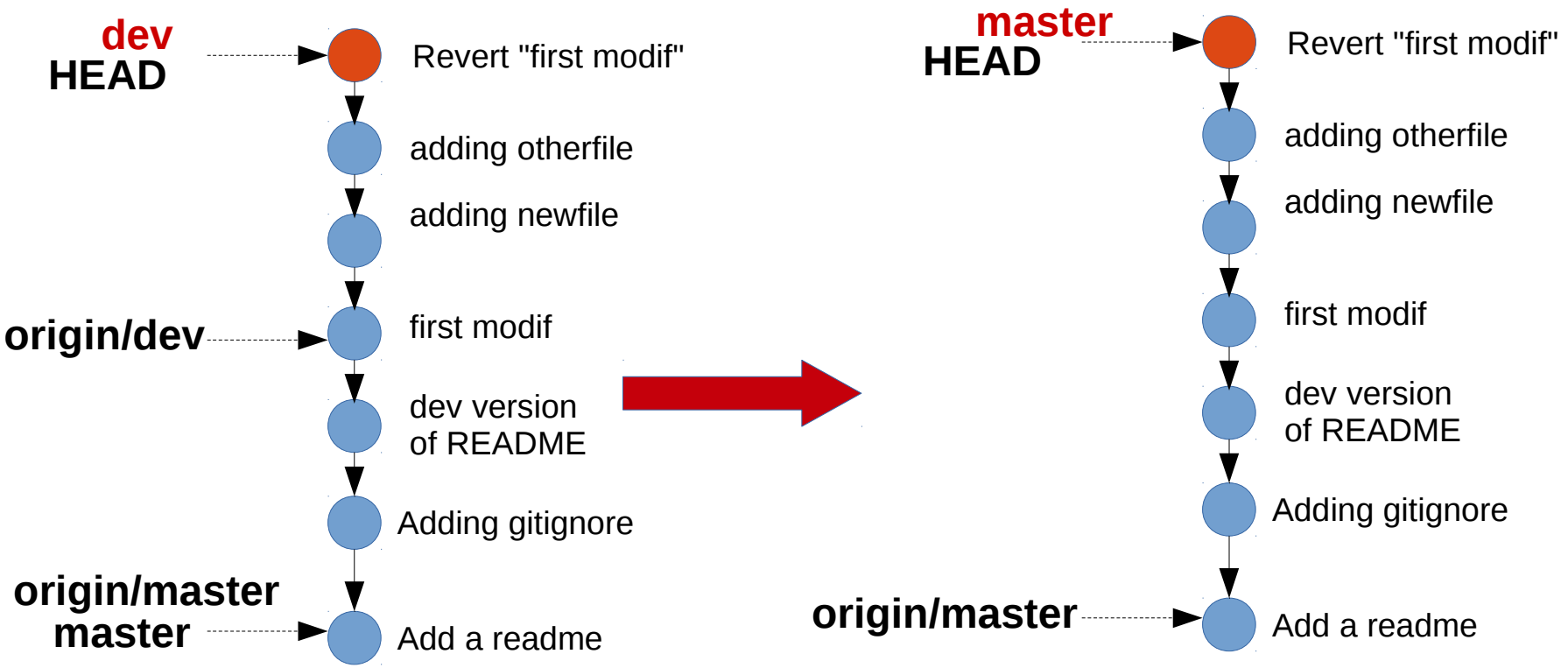
- Let's suppose development is finished, we do not need `dev` anymore.
  - **`git checkout master #go to master branch`**
  - **`git merge dev`**

- Then we want to delete `dev` branch on local workstation and on server
  - Delete local branch
    - **`git branch -D dev`**
  - Delete remote branch
    - **`git push origin :heads/dev`**

```
To git@gite.lirmm.fr:passama/git-first-example.git
 - [deleted]         dev
```

Tutorial: first project

- Result

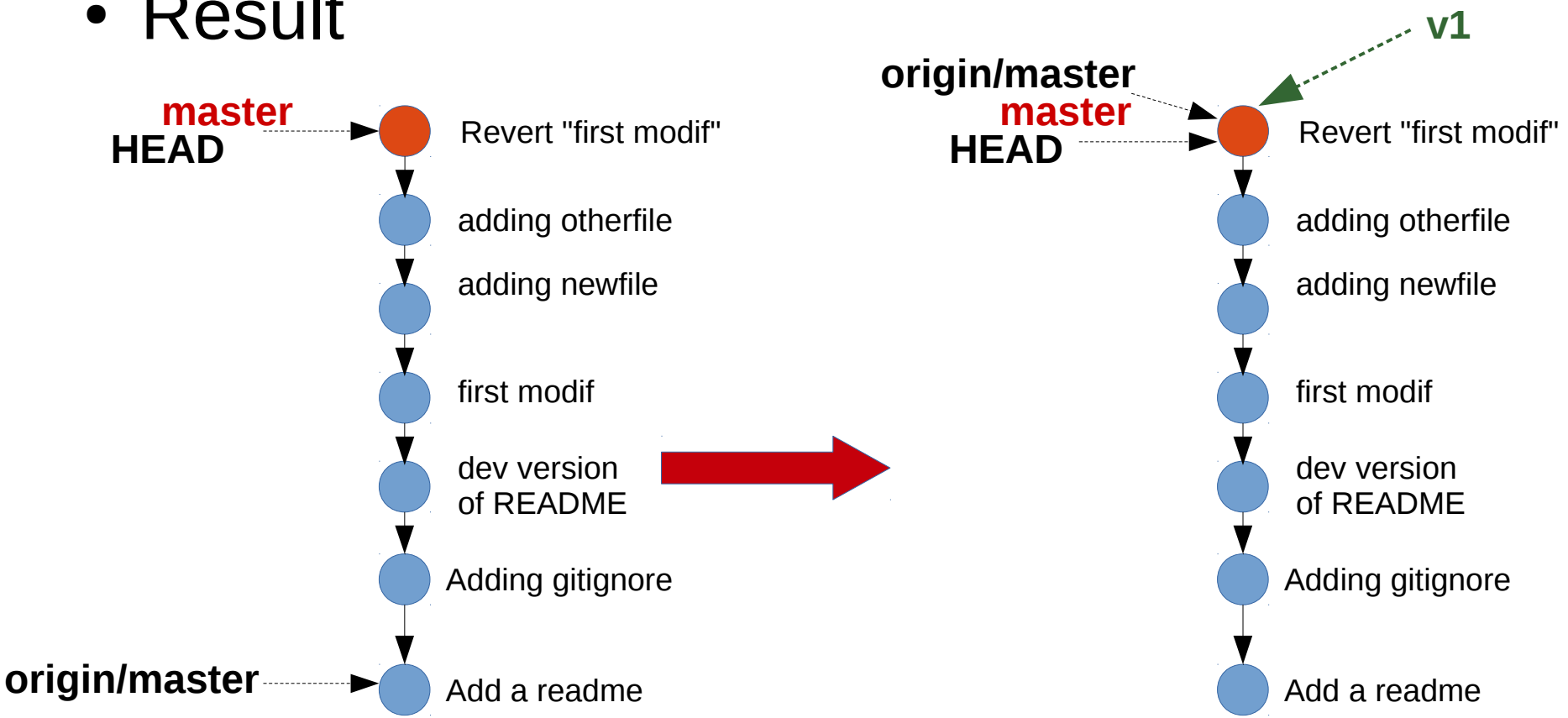


Tutorial: first project

- Memorizing the state of the repository
  - `git tag -a v1 "version1"`
- Then update server repository
  - `git push origin master`
  - `git push origin v1`
- Any time you want to go back to this state
  - `git checkout v1`

Tutorial: first project

- Result



Plan

- Installation
- A brief history of version control systems
- GIT concepts
- GITLAB Server
- **Step by step tutorial**
  - A first project
  - **Collaborative work**
  - Useful tips and advices

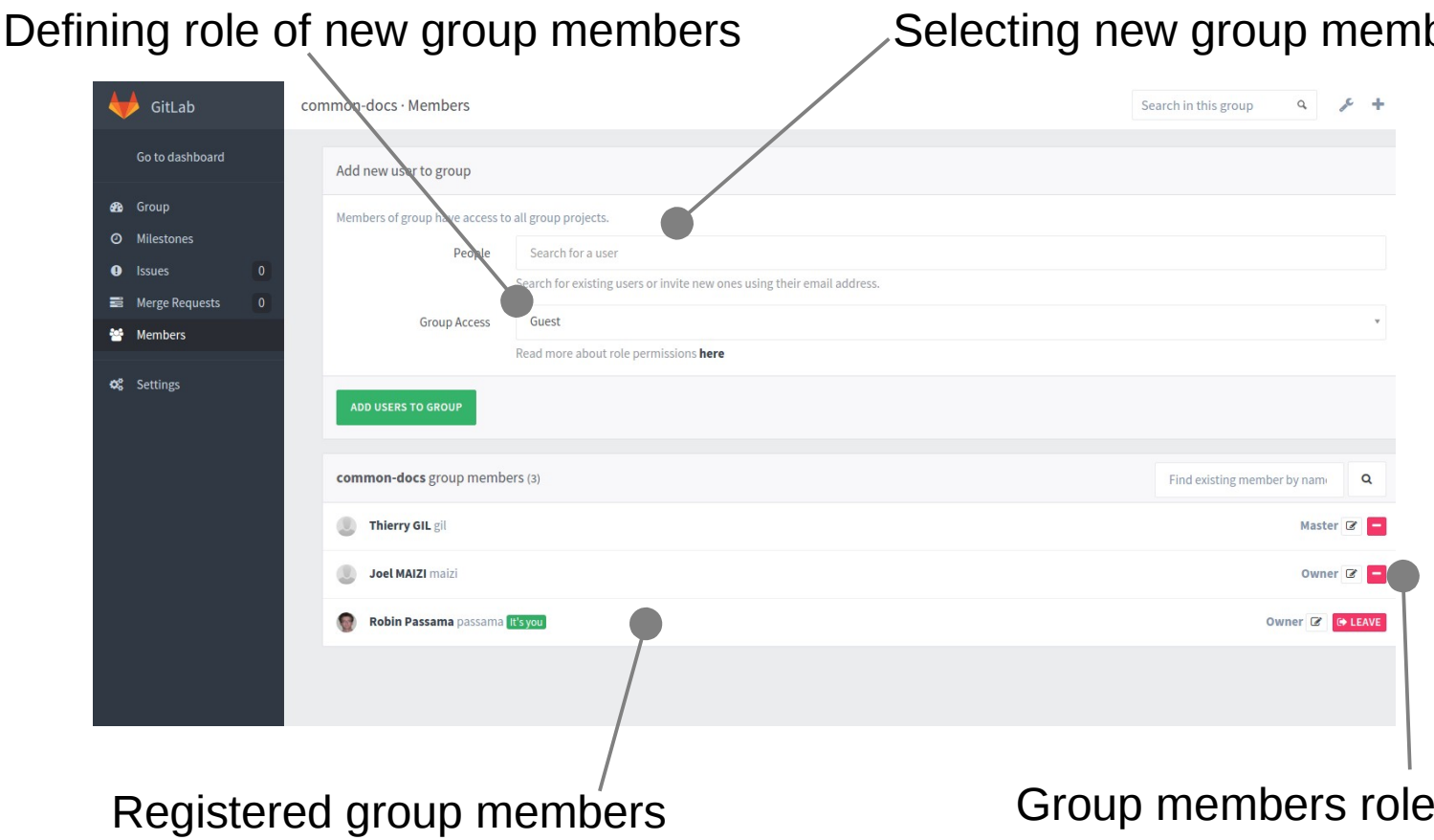
Tutorial: collaborative work

- In Gitlab create a group for 3-5 persons
  - A group is a set of related projects.
    - A group defines a workspace for projects.
  - A group defines a set of developers working on these projects.

Group names are unique in the server

Tutorial: collaborative work

- Now you have to add members to this group

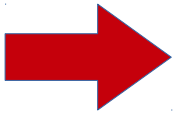


Tutorial: collaborative work

- Understanding permissions attached to roles
  - Available roles:
    - Guest: cannot pull/clone repository, can only create issues.
    - Reporter: Guest + can pull/clone repository
    - Developer: Reporter + can contribute (push to non protected branches, create and manage merge request, write wiki, etc.)
    - Master: Developer + manage team, manage branch protection, push to protected branch, can create projects in group.
    - Owner: Master + manage project configuration (create, rename, remove, switch visibility, etc.), manage group membership.
  - **A role in a group implies at least same role in all projects of this group.**

Tutorial: collaborative work

- Now create a project `test-git` in the group (Master or owner of the group)
  - All members of the project must be at least **Developer** (need push rights).
  - In a project you can invite people not belonging to the group.
  - You can add greater permissions to people of the group.

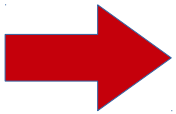


Click on “new project” in the group page.

Tutorial: collaborative work

- The **Owner** of `test-git` has to initialize the project:
  - Locally, open a terminal:
  - **`cd <somewhere>`**
  - **`mkdir test-git`**
  - **`cd test-git`**
  - then edit **`README.md`** and edit a **`.gitignore`** file (ignore temporary files)
  - **`git init #transform an existing folder into a git repository`**
  - **`git add --all`**
  - **`git commit -m “first commit”`**
  - **`git remote add origin <address of the project created in Gitlab>`**
  - **`git push origin master`**

- In Gitlab your project is initialized



Good practice: create a README.md file when creating your project (use markdown syntax) to generate simple welcome page.

Tutorial: collaborative work

- For other members of the group
  - Locally, open a terminal:
  - `cd <somewhere>`
  - `git clone <address of the project created in Gitlab>`

- **Now you are ready for collaborative work**

Tutorial: collaborative work

- **Owner** or a **Master** creates a file `file1.cpp`, then writes some content like:

```
#include <iostream>
int main(){
  cout<<”Hello world”<<endl;
  return 0;
}
```

- Then commit and publish this modification as usual:
  - `git add file1.cpp`
  - `git commit -m “adding file1”`
  - `git push origin master`

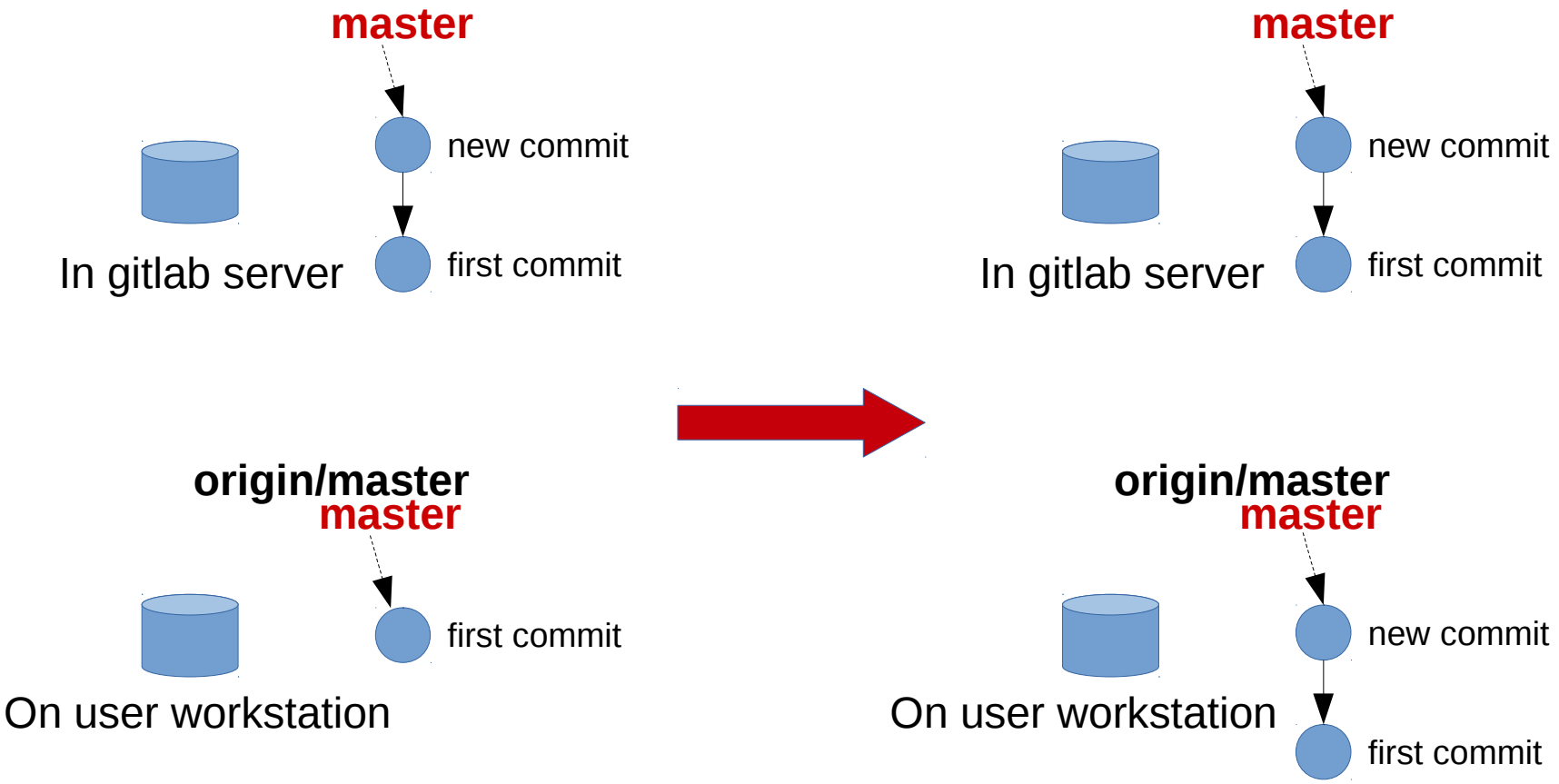
Tutorial: collaborative work

- Now other members have to update their local repository:
  - `git pull origin master`

- This pull command does:
  - A `fetch` of the repository (getting all new modifications from server repository).
  - A merge of the `origin/master` branch into local `master` branch.

Tutorial: collaborative work

- Pull effect



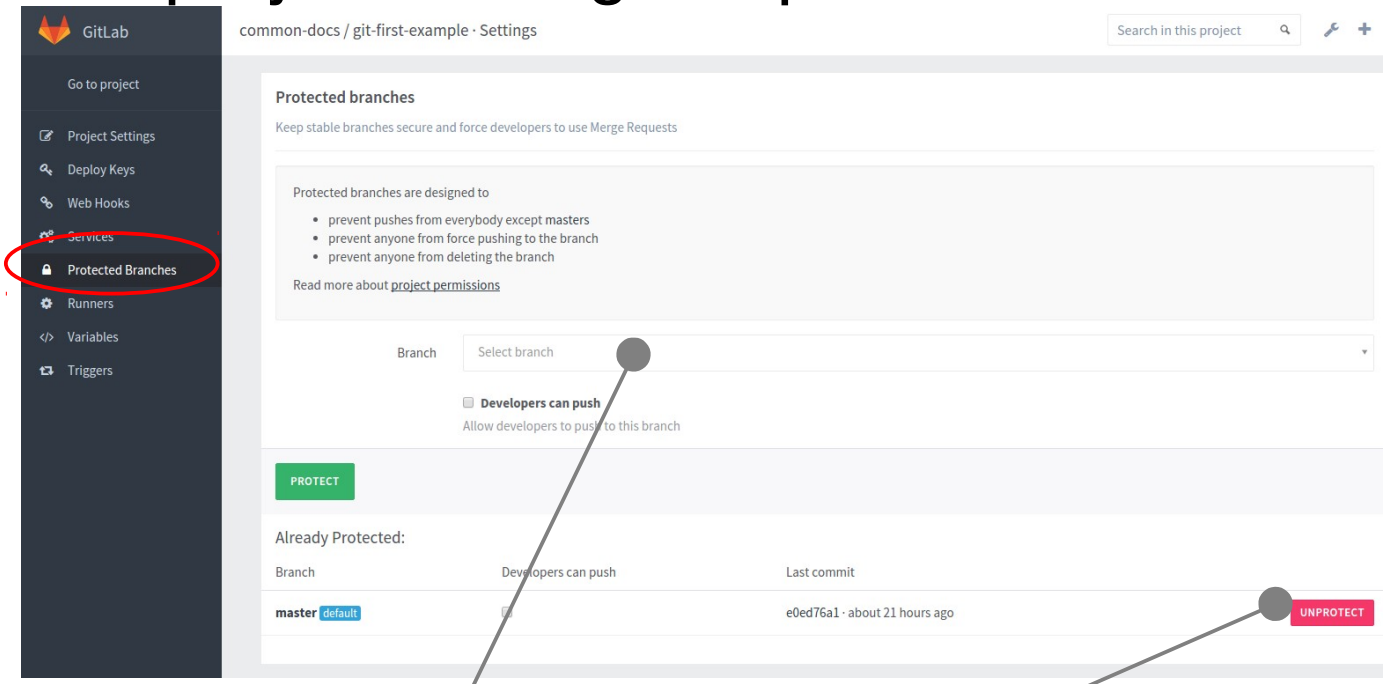
Tutorial: collaborative work

- Now all users of the group except **Master** and **Owner** modify the file `file1.cpp` in their local repository.
- Then, commit and push the changes to master
  - `git push origin master`

    FAILED
- Normal situation since `master` branch is **protected** by default
  - Only **Masters** and **Owners** can push to protected branches by default.
  - Why: prevent branch deletion and forced push by developers

Tutorial: collaborative work

- Manage branch protection in Gitlab
  - Go to project settings > “protected branches”



Select branch to protect

Unprotect the branch

Good practice: keep `master` branch protected and do not allow developers to push

Tutorial: collaborative work

- Solution: developers create another branch and propose a merge request
  - Create a new branch on server
    - **`git checkout -b <my-branch-name>`**
    - **`git push origin <my-branch-name>`**
  - Propose a merge request
    - In Gitlab create a new merge request with
      - **`<my-branch-name>`** as source
      - **`master`** as target

Tutorial: collaborative work

- Now Owner and Master can manage the merge request directly in gitlab
  - In “merge requests” menu of the project, check for modifications,
    - if OK, accept the merge request.
  - If conflicts, they must be resolved “by hand” (i.e. in your local repository):
    - `git checkout master #in case of`
    - `git pull origin master #update master`
    - `git pull origin:<branch name> master`
      - Should complain about a conflict

Tutorial: collaborative work

- Resolving a conflict
  - To get information about the conflict
    - **`git status`**

```
# On branch master
# You have unmerged paths.
#   (fix conflicts and run "git commit")
#
# Unmerged paths:
#   (use "git add ..." to mark resolution)
#
# both modified:      file1.cpp
```

Files where to find conflicts

Tutorial: collaborative work

- Resolving a conflict
  - Look into these files you should see something like

```
the number of planets are
<<<<<<< HEAD
nine
=======
eight
>>>>>>> <the branch name>
```

What you current branch contains

What merged branch contains

Tutorial: collaborative work

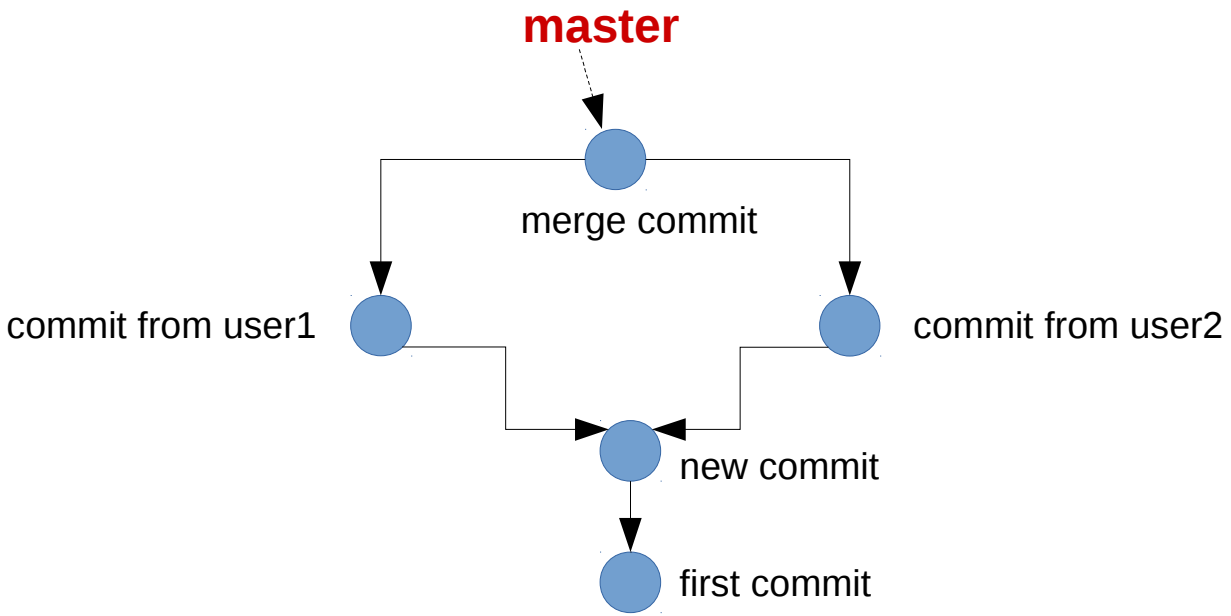
- Resolving a conflict
  - Resolution = choosing alternative (or rewriting everything) + deleting specific comments

```
the number of planets are
eight
```

  - Then doing a specific commit:
    - `git commit -am “conflict on planets resolved” #add --all and commit in one step is possible`
  - Then updating server master branch
    - `git push origin master`

Tutorial: collaborative work

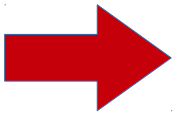
- Repository, after a merge (conflicting or not, coming from local or remote branches)



- When no conflict, merge commit is generated automatically

Tutorial: collaborative work

- Most of time, **Developers** should resolve merge by themselves
  - Create a kind of “sandbox” branch shared by all contributors. Example names: `dev`, `integration`.
    - `git checkout -b dev`
    - `git push origin dev`
  - Master branch is updated (by merge) only when `dev` branch state is considered as “stable”.



Good practice: make `dev` branch protected but allow developers to push (ensure branch will not be deleted)

Tutorial: collaborative work

- All users
  - Get the `dev` branch
    - `git fetch origin #update repository`
    - `git checkout dev #local dev branch is automatically created`
  - Write some code and commit it to `dev` branch
  - Update repository by pulling `origin dev` branch
  - Eventually resolve conflicts on local workstation
  - Once done push to `origin dev` branch
  - Etc.

Plan

- Installation
- A brief history of version control systems
- GIT concepts
- GITLAB Server
- **Step by step tutorial**
  - A first project
  - Collaborative work
  - **Useful tips and advices**

Tutorial: Useful tips and advices

- Control Visibility of your project with Gitlab
  - To keep your project private use “private” visibility.
    - Only members of the project (or group) can clone/fork it if they have adequate rights.
  - To share your project with the world set it “public”.
    - Not recommended, instead use popular services like github.com, gitlab.com or SourceSup, **for better visibility** !
  - To share with any people from LIRMM, set it “internal”.
    - Anyone connected can find and clone the project.
    - Anyone connected **can fork the project to contribute via merge requests.**

Tutorial: Useful tips and advices

- Typical organization of “big” software projects
  - Create a group for a big project
    - **Owners** of the group are project managers
    - others are **Developers**.
  - Create one Gitlab project for each “independent” element of your software,
    - Each manager of individual project is a **Master** (or **Owner**).
    - Other are **Developers**.

Tutorial: Useful tips and advices

- With Gitlab, use **issues** and **code snippets** to communicate on bugs, improvements, suggestions
  - Issues are the best way to keep traces of important things to do, improvements, etc.
  - Use **labels** on issues to clearly identify the subjects of your issues (bugs, documentation, etc.)
  - Use **code snippet** to write examples of code, to report long error messages, etc. then reference them in issues.

Tutorial: Useful tips and advices

- Use `git-svn` to port your projects into git world
  - Import the entire SVN repository into a git repository

```
git svn clone <address> -s
```

This operation may be quite long for repositories with a lot of commits

  - Create the corresponding project in Gitlab, then

```
git remote rename origin svn-server
git remote add origin <gitlab project address>
```

  - Push all branches and tags to this new repository ... finished !

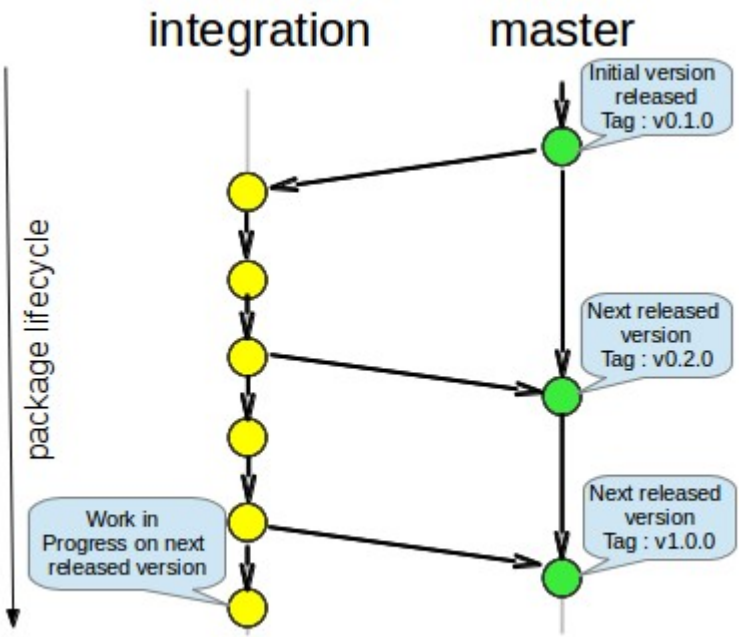
```
git push origin --all #pushing all branches
git push origin --tags #pushing all tags
```

Tutorial: Useful tips and advices

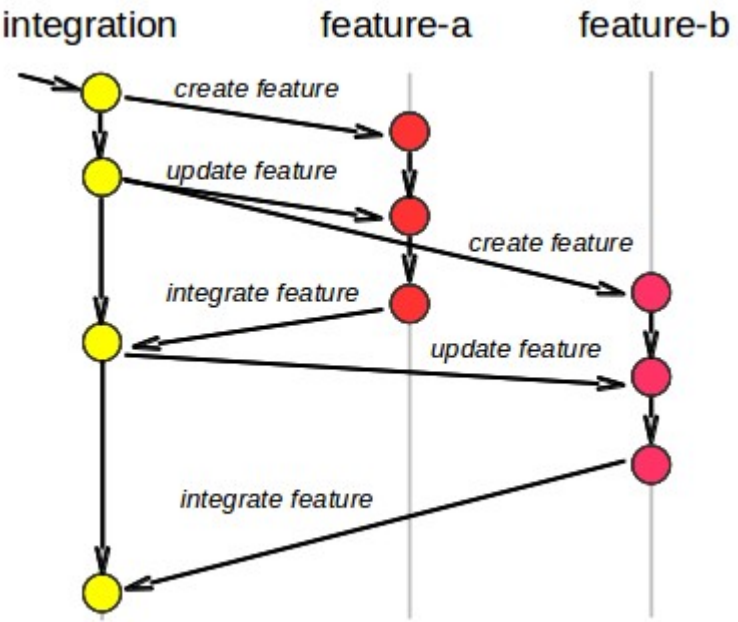
- Ignoring files with .gitignores
  - Always create a .gitignore file at the root of your project.
    - Removes temporary files and folders generated by development tools you use.
  - To enforce an organization for projects' file system
    - add a .gitignore for each **empty directory you want** (typically `build`, `bin` and `lib` folders and the like).
    - Make it remove all the content of the folder by using a unique `*` rule.
    - These folders exist in the repository but not their content (except .gitignore) !

Tutorial: Useful tips and advices

- A simple and efficient branching model (see doc-git wiki)
  - **Integration:** protected and “Developers can push”
  - **Master**: protected and **NOT** “Developers can push”



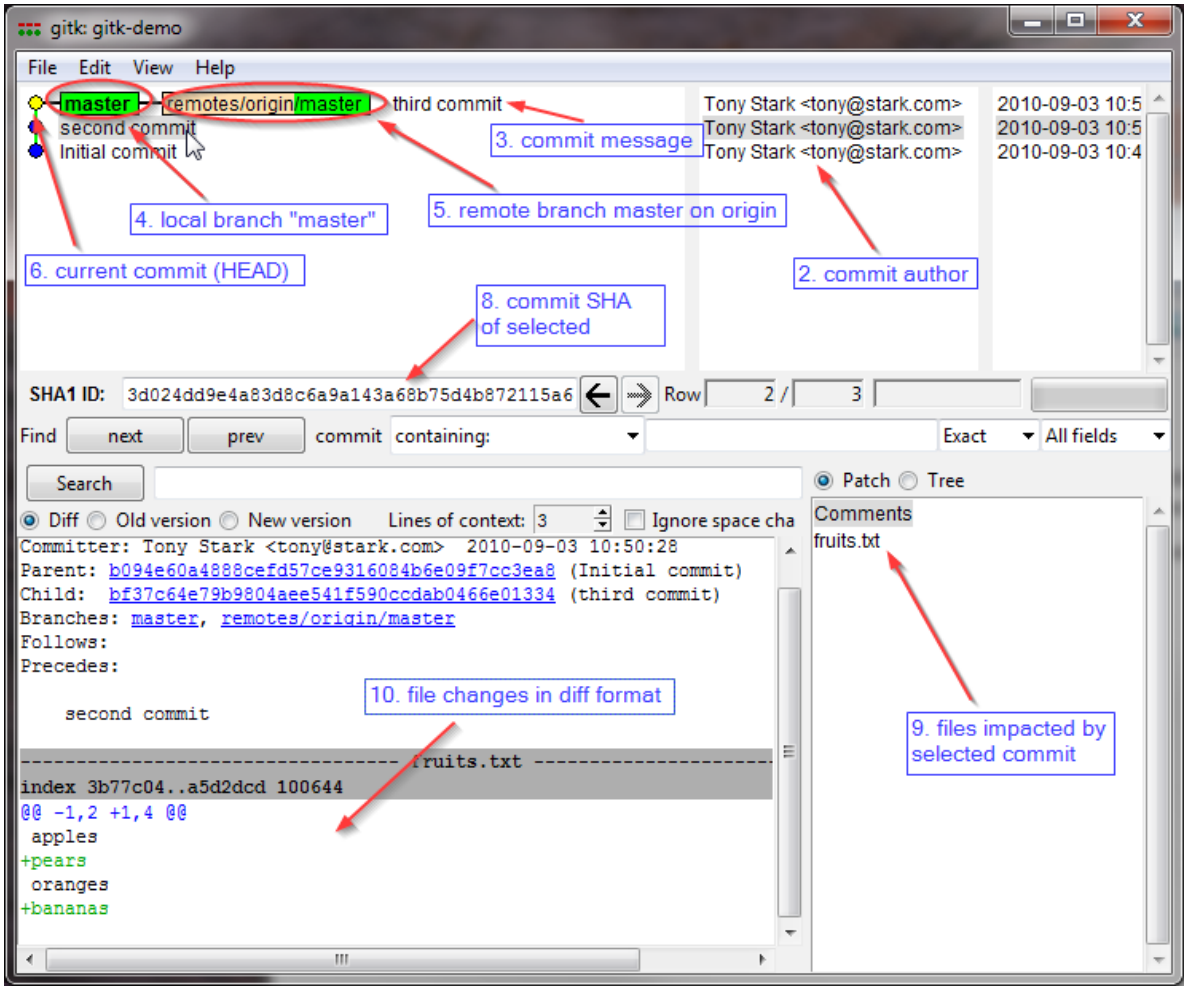
Permanent branches (protected)



Temporary branches for features development

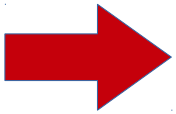
Tutorial: Useful tips and advices

- Use `gitk` tool to understand the state of local repository



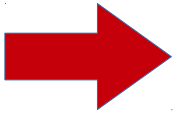
Conclusion

- All basic concepts and commands of git have been studied … but they have hundreds of subtle refinements !!



Look at the wiki of project doc-git on gite.lirmm.fr (your contributions are welcome).

- Basic functionalities of Gitlab have been studied … but there is still a lot to learn



Continuous integration will be part of a new tutorial in the future.

Thanks for your attention