

Introduction à Git (et Gitlab)

Robin Passama
LIRMM – CNRS/UM

Qu'est-ce que Git ?

- Système de contrôle de version (***version control system***)
 - Un ensemble d'outils logiciels pour:
 - Mémoriser et retrouver différentes versions d'un projet.
 - Faciliter le travail collaboratif.
 - Initialement développé par Linus Torvalds pour faciliter le développement du noyau Linux.
 - Logiciel libre / open source.
 - Disponible sur toutes les plates-formes.

Qu'est-ce que Gitlab ?

- Application web basée sur git.
 - Permet de gérer:
 - Le cycle de vie de projets git.
 - Les participants aux projets (roles, groupes, etc.).
 - La communication entre ces participants.
 - Développée par une société privée
 - La “community edition” est libre.
 - Un service gratuit en ligne (comme github)

Pourquoi utiliser Git & Gitlab ?

- Postulat: besoin d'un système pour gérer le cycle de vie de projets logiciels.
 - Gérer les versions.
 - Gérer les développeurs (droits d'accès).
 - Gérer les “issues” (rapports de bug, suggestions d'évolution, etc.) .
 - Créer une documentation en ligne pour un projet.
 - Construction, test et déploiement automatiques.

Quand les utiliser ?

- Développement logiciel
 - Imaginés pour gérer le développement de logiciel...
- Ecrire des documents
 - Articles (e.g. sources latex).
 - Pages Web (e.g. HTML, markdown, etc.).
 - N'importe quel type de document texte (format ascii).
- Archiver des binaires (utiliser l'extension **git-lfs**)
 - executable et librairies “releases”
 - Fichiers générés par des éditeurs spécifiques (e.g. CAO, Word).

Objectifs de cette UE

- Comprendre:
 - Le contrôle de version de projets
 - Les concepts de base de *git*
 - Le travail collaboratif
 - La gestion de projet en utilisant *Gitlab*
- Pendant les TP, apprendre à utiliser:
 - Les commandes de base de git (côté station de travail)
 - Les principales fonctionnalités de Gitlab (côté server)
 - Les bonnes pratiques

Plan du cours

- **La brève histoire des systèmes de contrôle de version (VCS)**
- Les concepts de git
- Les fonctionnalités de Gitlab

Histoire des VCS

- Définitions simples :
 - **Version** = contenu du projet à un moment de son cycle de vie.
 - **Dépôt (*repository*)** = l'historique du projet, contenant toutes ses versions.
 - **Branche (*branch*)** = variante d'un projet.
- Pendant le développement d'un projet, il est nécessaire:
 - D'enregistrer les **versions** du projet dans un **dépôt**, de retrouver une **version** spécifique (e.g. dernière version sans BUG).
 - De travailler sur différentes variantes (**branches**) en parallèle, de fusionner ces variantes.
 - De partager les modifications faites entre développeurs (comprendre et fusionner les modifications faites par d'autres).

Histoire des VCS

- L'ancêtre des VCS: la méthodologie CPOLD
 - **dépôt** = dossier racine du projet.
 - A chaque dossier/fichier est associé un nom + une extension.
 - Nom = nom de l'élément contenu dans le projet
 - Extension = la **version** du fichier/dossier.
 - Basé sur la commande **cp** :
 - cp fichier1 fichier1.old //version "old" enregistrée
 - cp -R dossier dossier.1.4 //suppose que la version 1.3 existe !
 - Partage des modifications :
 - via un server de fichier classique (ou n'importe quel support)
 - la fusion des modifications se fait à la main.

Histoire des VCS

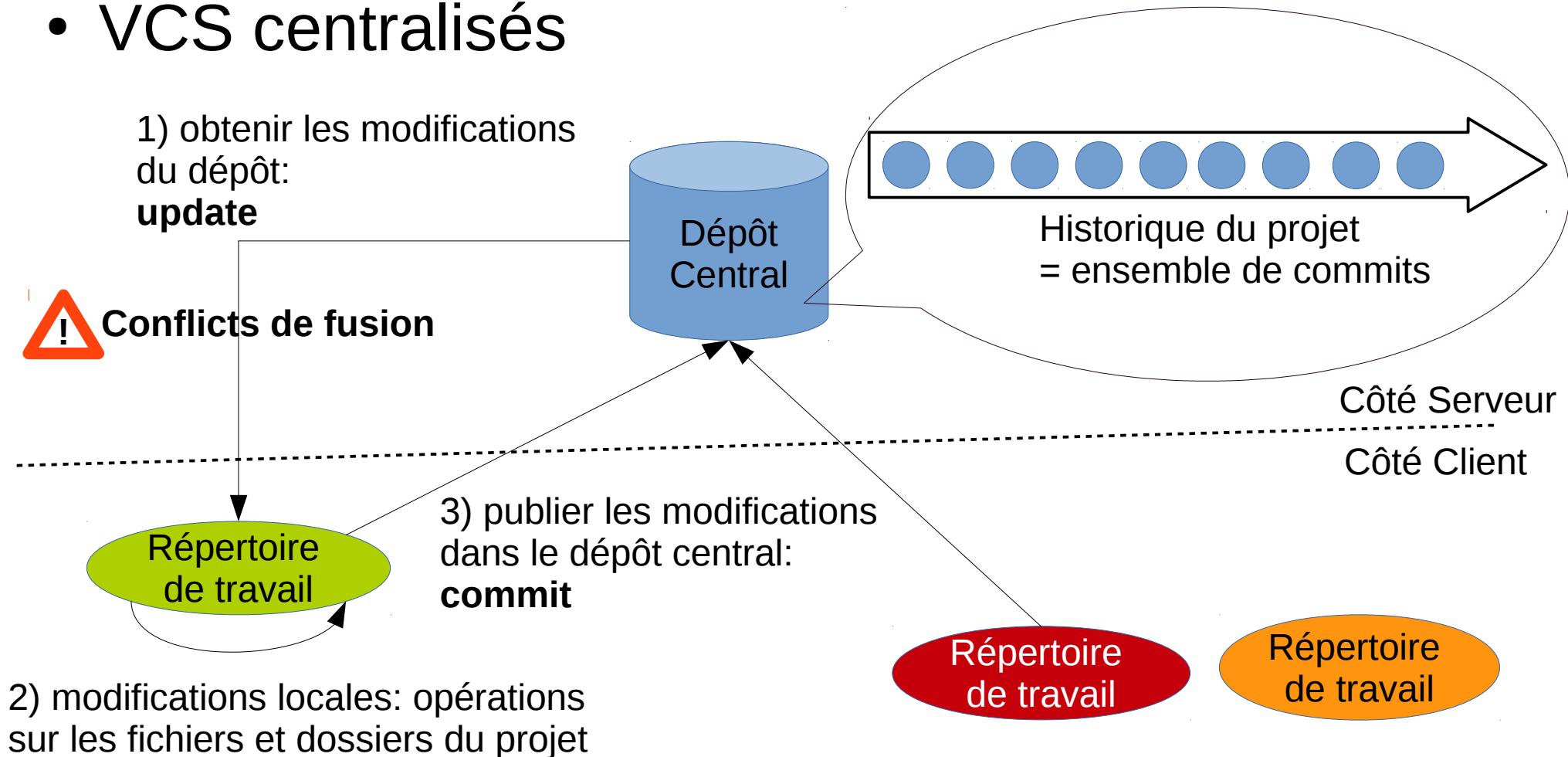
- CPOLD : limitations
 - Convention de nommage des extensions doit être suivie de manière stricte.
 - Sous-optimal en terme de place sur le disque
 - chaque version = copie complète code existant + ajouts.
 - Le partage des modifications et la fusion de variantes sont **très laborieux**.
-  Besoin d'outils pour automatiser, simplifier et robustifier la gestion de versions.

Histoire des VCS

- Les VCS centralisés : CVS, subversion, etc.
 - Outils logiciel basés sur une approche client/serveur.
 - Serveur = un serveur de fichiers qui **contient les dépôts** et gère les accès des utilisateurs.
 - Client = une station de travail qui contient **un répertoire de travail (*working folder*)** connecté au dépôt.
 - Principe : seules les **modifications** sont enregistrées.
 - **Commit** : un ensemble de modifications sur des fichiers / dossiers du projet, considéré comme atomique (indivisible).
 - **Dépôt** = historique des commits.

Histoire des VCS

- VCS centralisés



Histoire des VCS

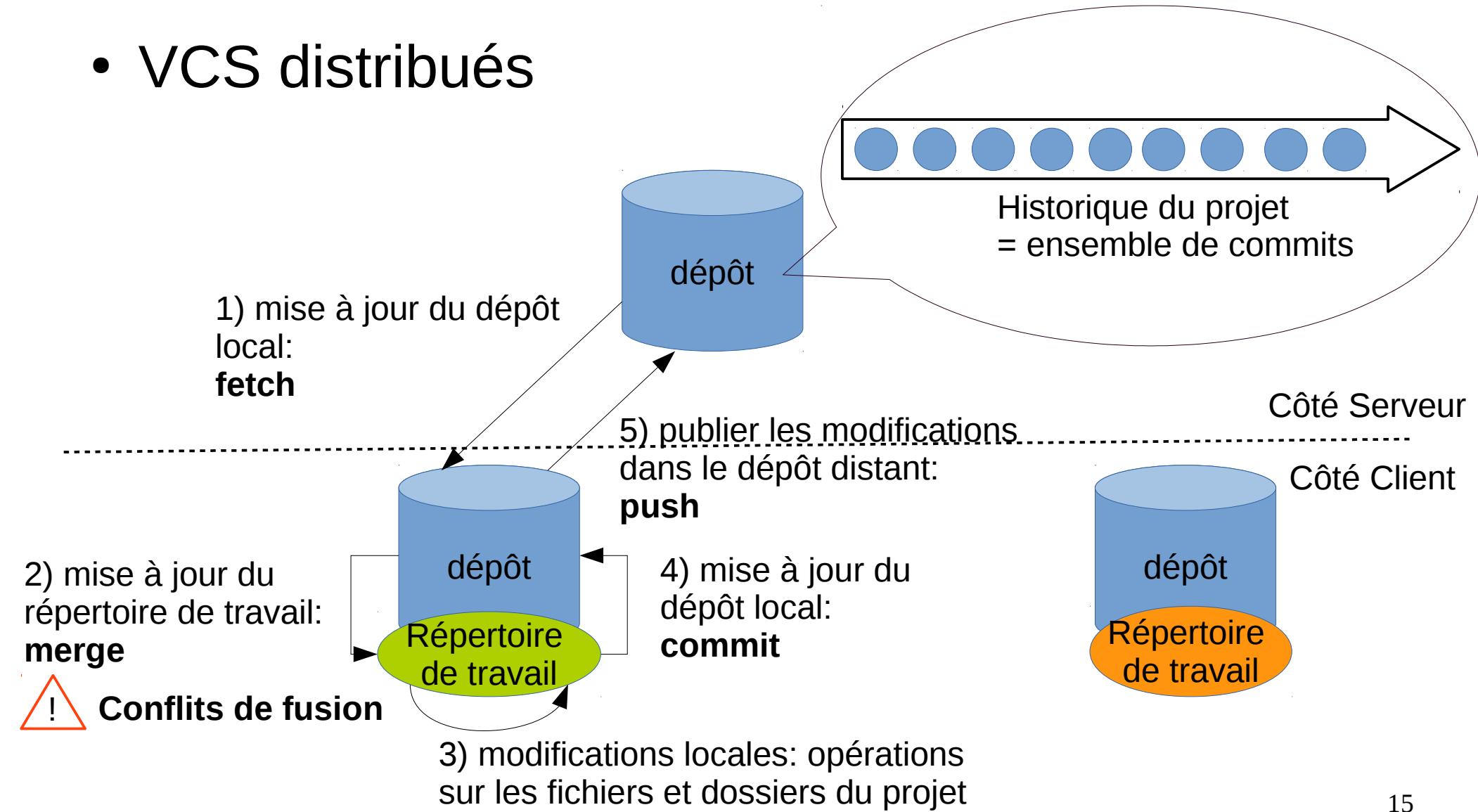
- VCS centralisés: avantages
 - Optimal en terme de taille sur le disque
 - Version = modifications apportées (commit) depuis version précédente.
 - Pas besoin de conventions.
 - Partage des versions est automatisé.
- VCS centralisés: limitations
 - Travail hors connexion difficile : pour créer des commits/fusionner des modifications, il faut se synchroniser avec le serveur.
 - Manque de robustesse : Perdre le dépôt central (e.g. corruption des données) = perdre tout l'historique.

Histoire des VCS

- VCS distribués: git, Bazaar, mercurial, etc.
 - Comme les centralisés mais ...
 - Basés sur une approche pair-à-pair pour la synchronisation des opérations.
 - Les dépôts existent **côté serveur et côté client**.
 - Sur les stations de travail: un répertoire de travail (*working directory*) est connecté à chaque dépôt.

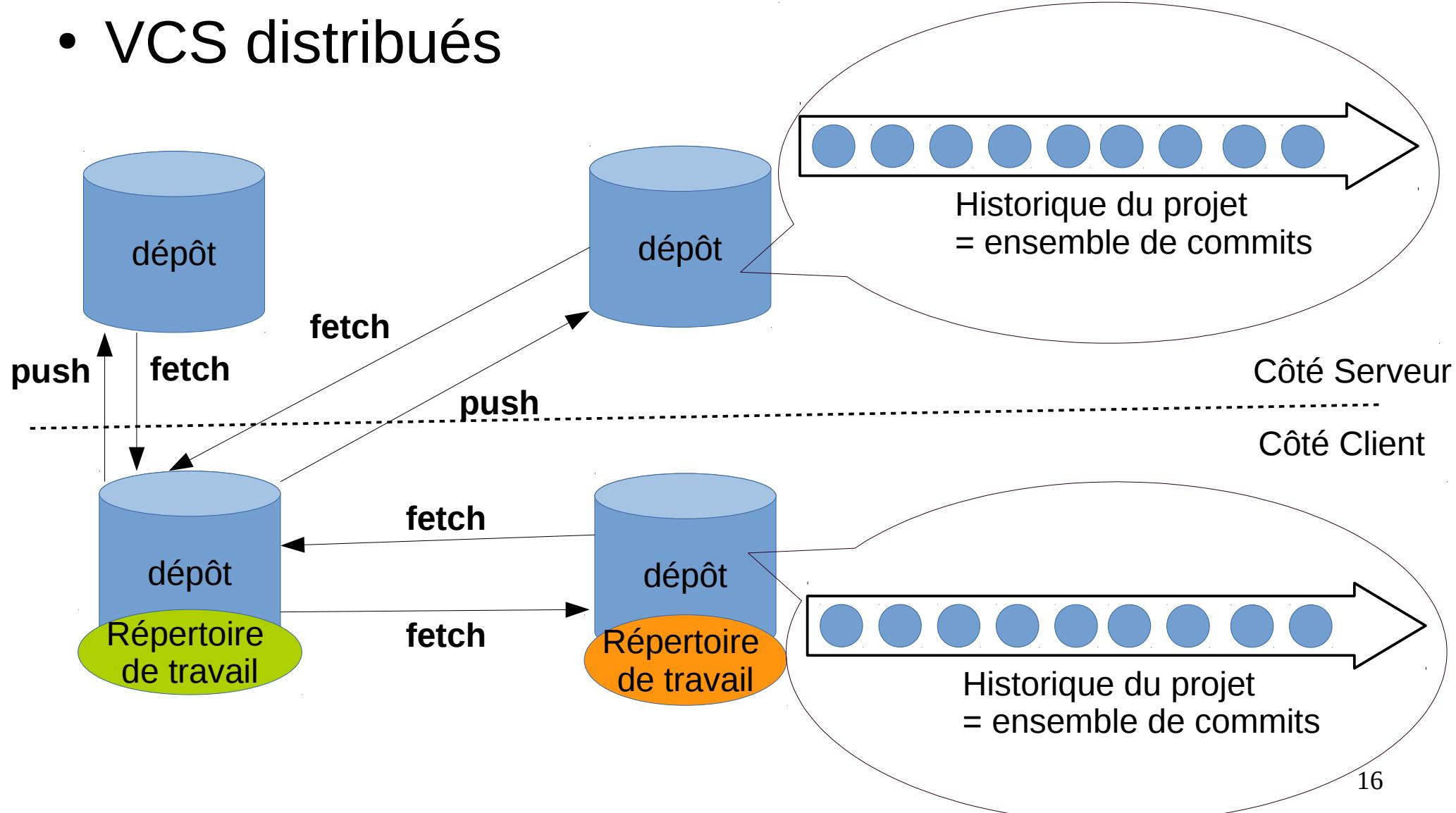
Histoire des VCS

- VCS distribués



Histoire des VCS

- VCS distribués



Histoire des VCS

- VCS distribués: Avantages
 - Robustesse : multiples copies du même dépôt.
 - Travail hors connexion facile : créer des commits et fusionner des modifications se fait hors connexion.
- VCS distribués: limitations
 - Plus de commandes à comprendre...
 - **Garantir une politique d'accès aux dépôts est très difficile avec les DVCS**
 - Chaque dépôt a une copie complète de l'historique (depuis dernier **fetch**).
 - Chaque dépôt défini ses propres droits d'accès pour ses clients.

Plan du cours

- La brève histoire des systèmes de contrôle de version (VCS)
- **Les concepts de git**
- Les fonctionnalités de Gitlab

Concepts de GIT

- Le dépôt (*repository*)

- Un répertoire dont la structure décrit le graphe de commit, les branches, etc.
- Transformer un dossier en dépôt: `git init`



Introduction à Git (et Gitlab)

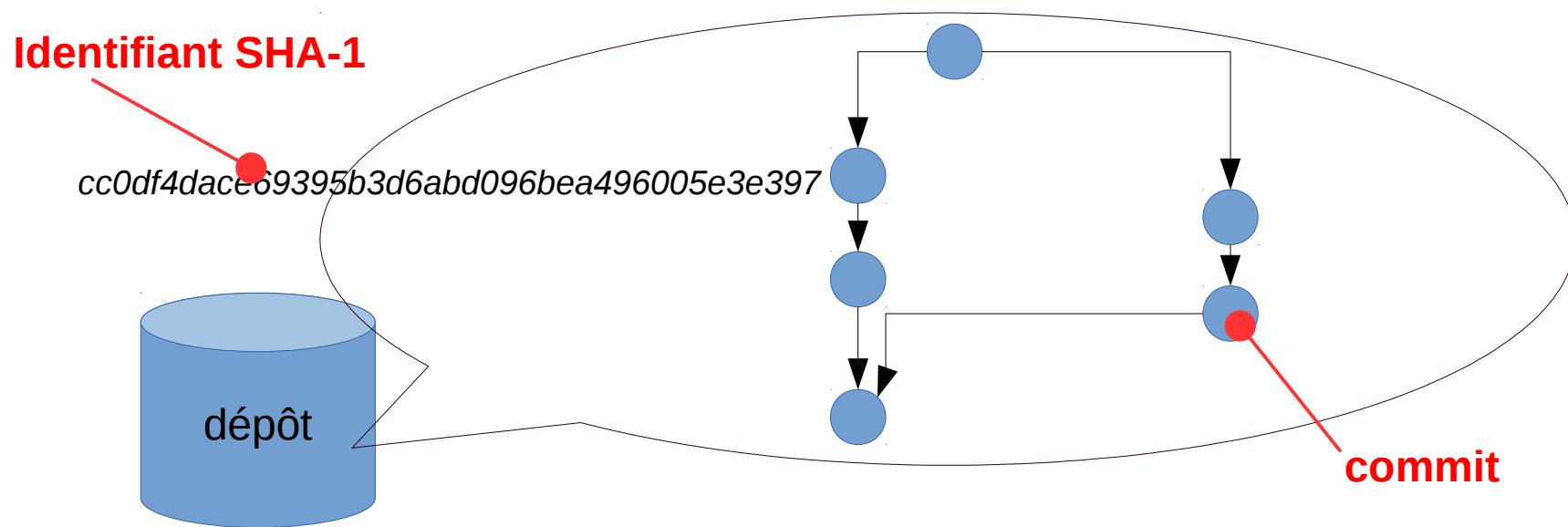
Robin Passama
LIRMM – CNRS/UM

Concepts de GIT

- **Le commit**

- Un ensemble de modifications à des fichiers et répertoires.
 - Représenté par une opération de *patch*.
- Identifié de manière unique (relativement à un projet) par un hash code SHA-1

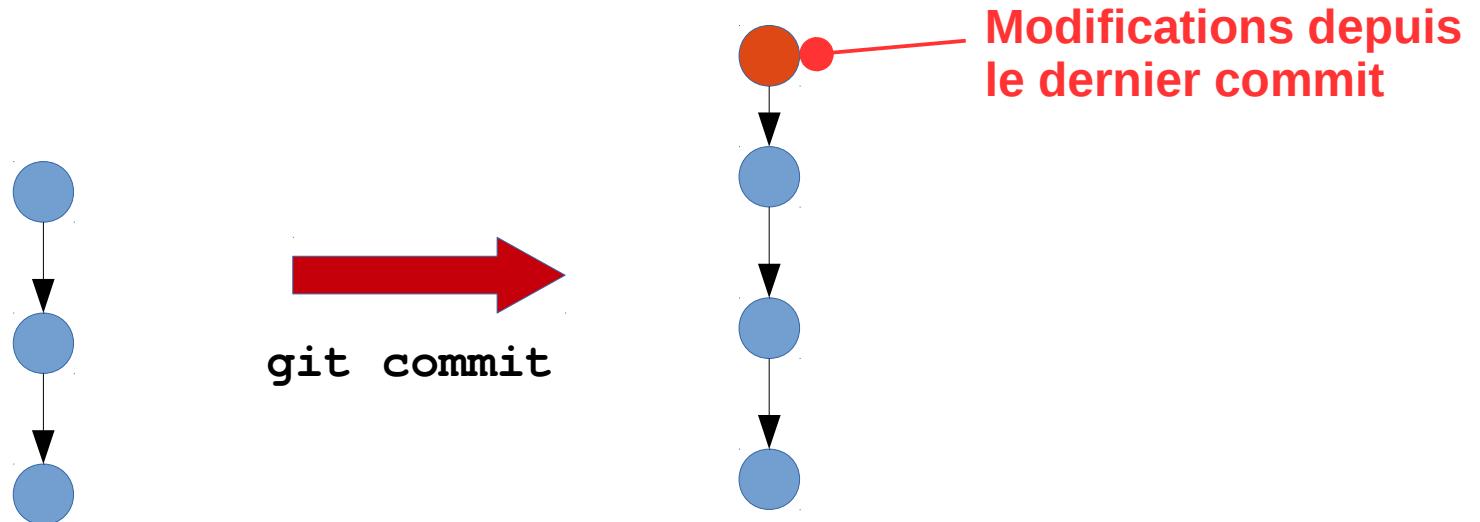
- Les commit sont structurés suivant un **graphe acyclique orienté**



Concepts de GIT

- **Le commit**

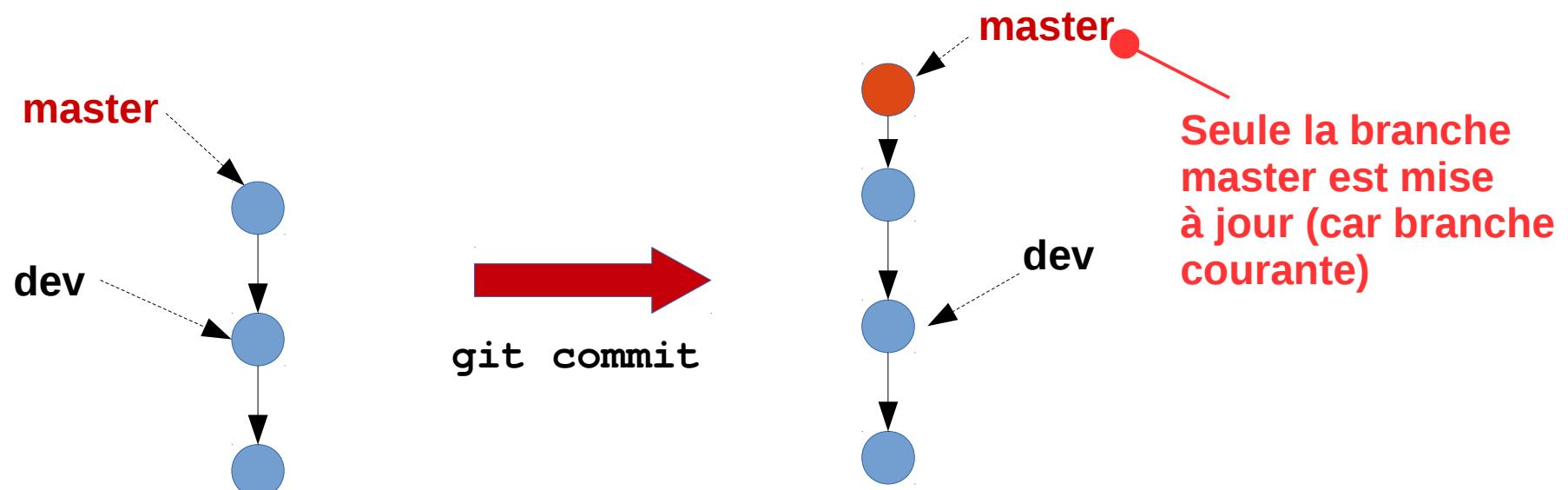
- Le graphe est mis à jour chaque fois qu'un commit est créé (e.g. action de « *committer* »)
 - Le nouveau commit définit un pointeur sur le commit précédent et contient les modifications apportées au contenu du précédent.



Concepts de GIT

- La branche (branch)

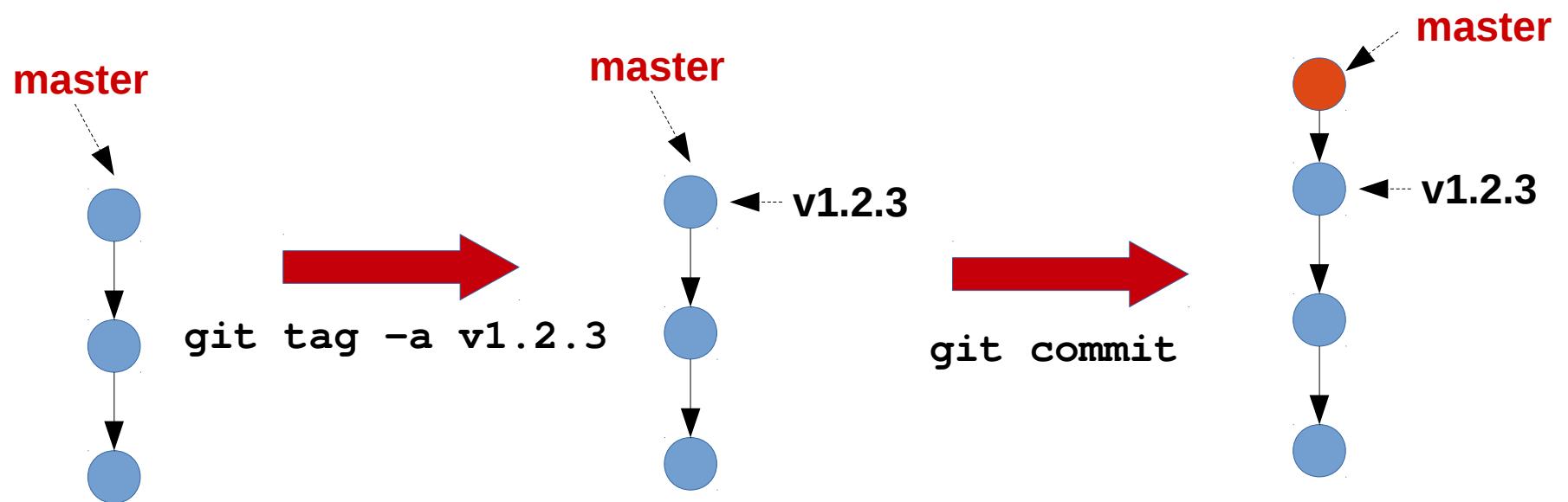
- Un **pointeur** sur (i.e. une référence à) un commit.
- Mis à jour chaque fois qu'un commit est créé, pointe alors vers ce nouveau commit.
- branche par défaut : **master**.
- Git définit une **branche courante**.



Concepts de GIT

- Le **tag**

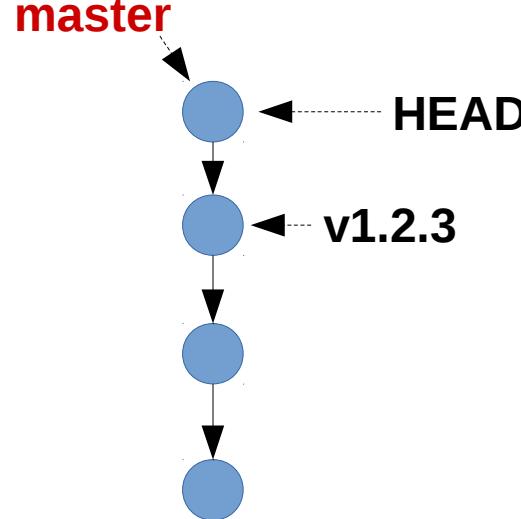
- Un **pointeur** sur (i.e. une référence à) un commit.
- Mis à jour uniquement sur commande explicite de l'utilisateur.
- Permet de mémoriser les versions intéressantes.



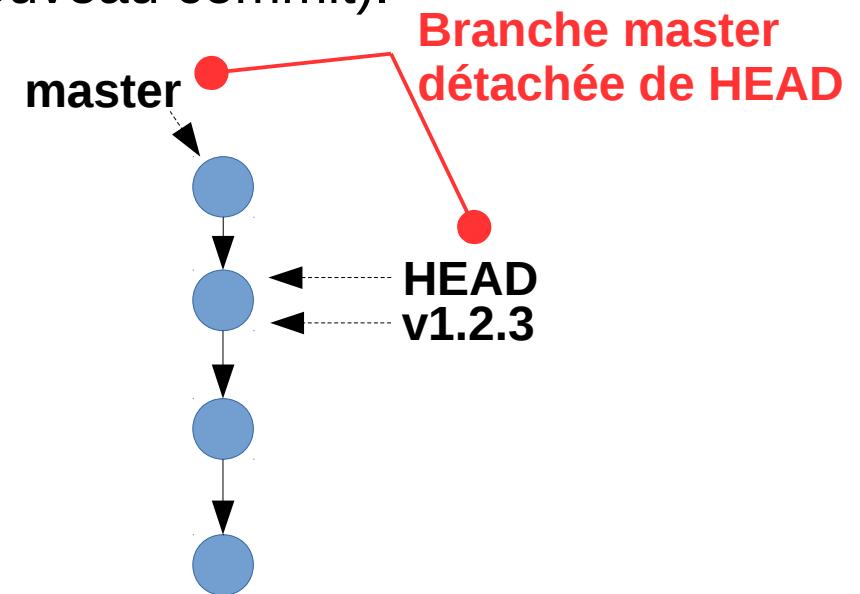
Concepts de GIT

- **Le pointeur HEAD**

- Un **pointeur** sur (i.e. une référence à) un commit.
- Représente l'**état courant du répertoire de travail** : modifications appliquées au projet jusqu'au commit ciblé.
- Permet de naviguer dans le graphe pour retrouver une version donnée.
- Mis à jour sur commande ou automatiquement chaque fois qu'un commit est créé (pointe alors vers ce nouveau commit).

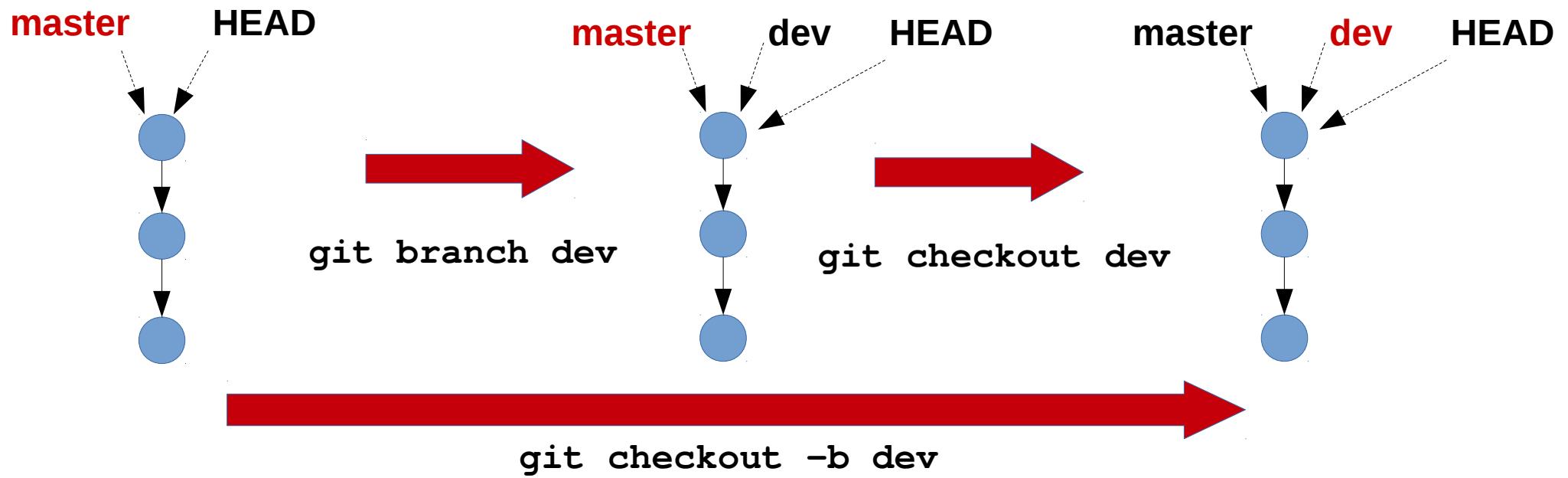


`git checkout v1.2.3`



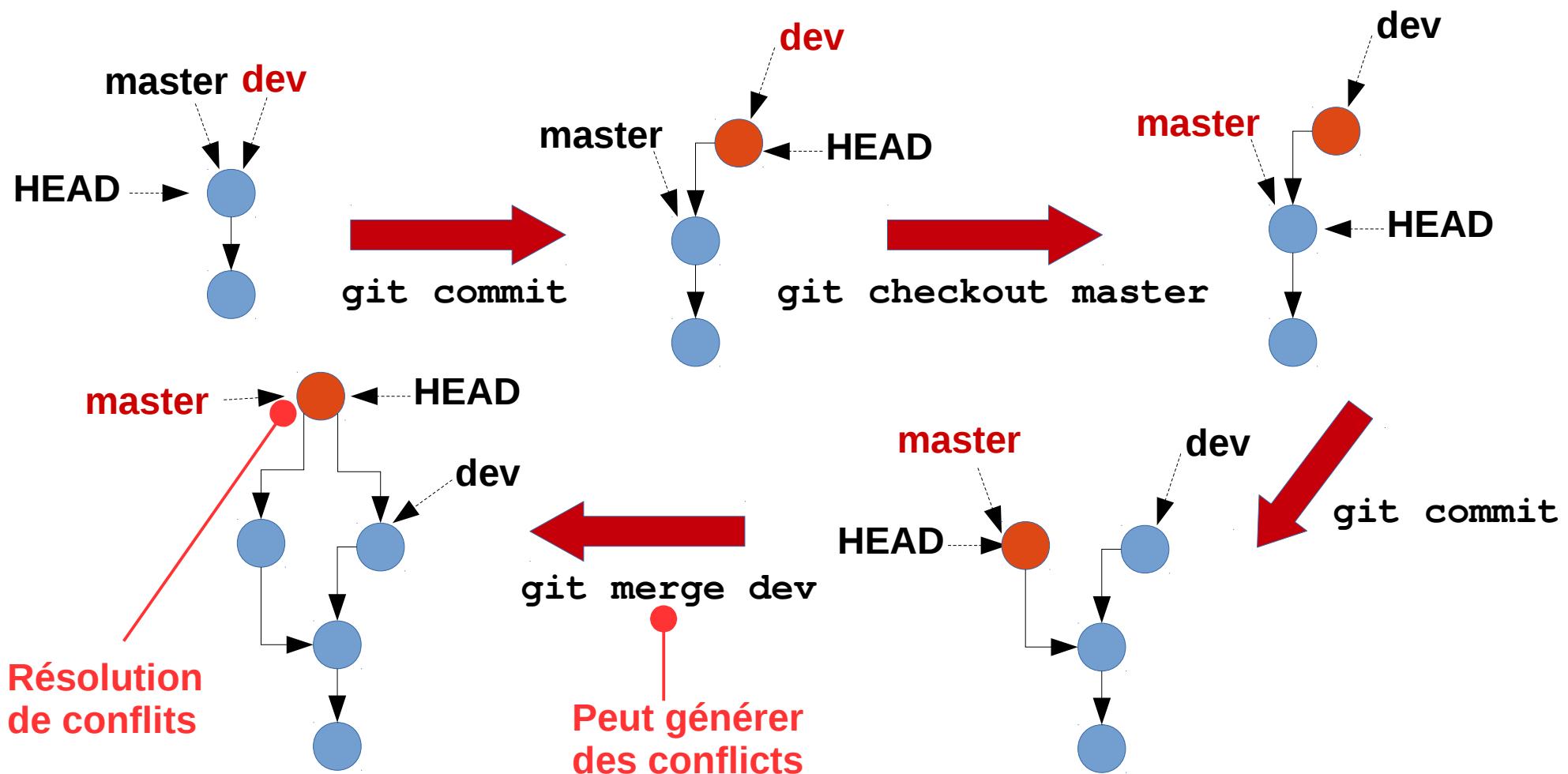
Concepts de GIT

- Créer des **branches**
 - Plusieurs branches dans le même dépôt.
 - Créer une branche = créer un pointeur sur le commit ciblé par **HEAD**.



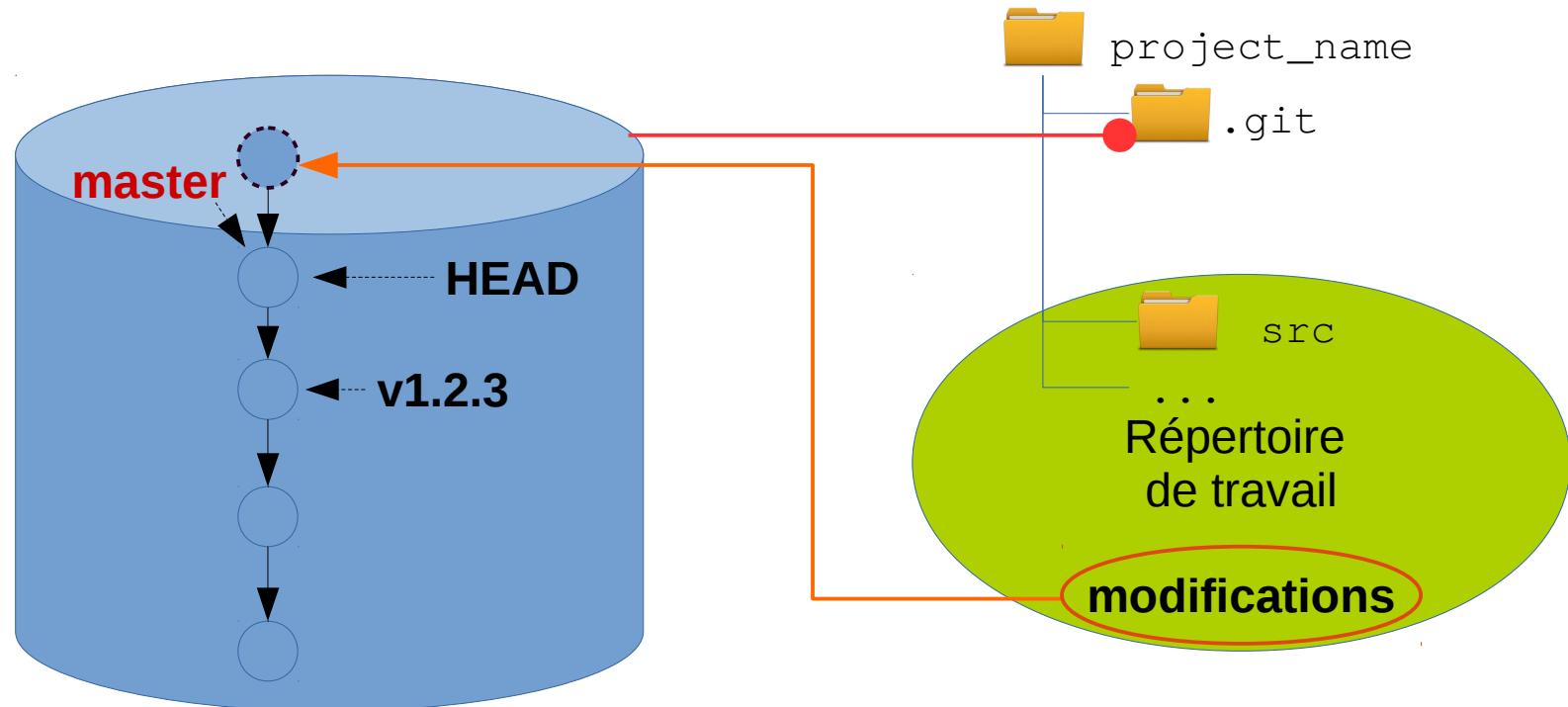
Concepts de GIT

- Fusion (*Merge*) de branches



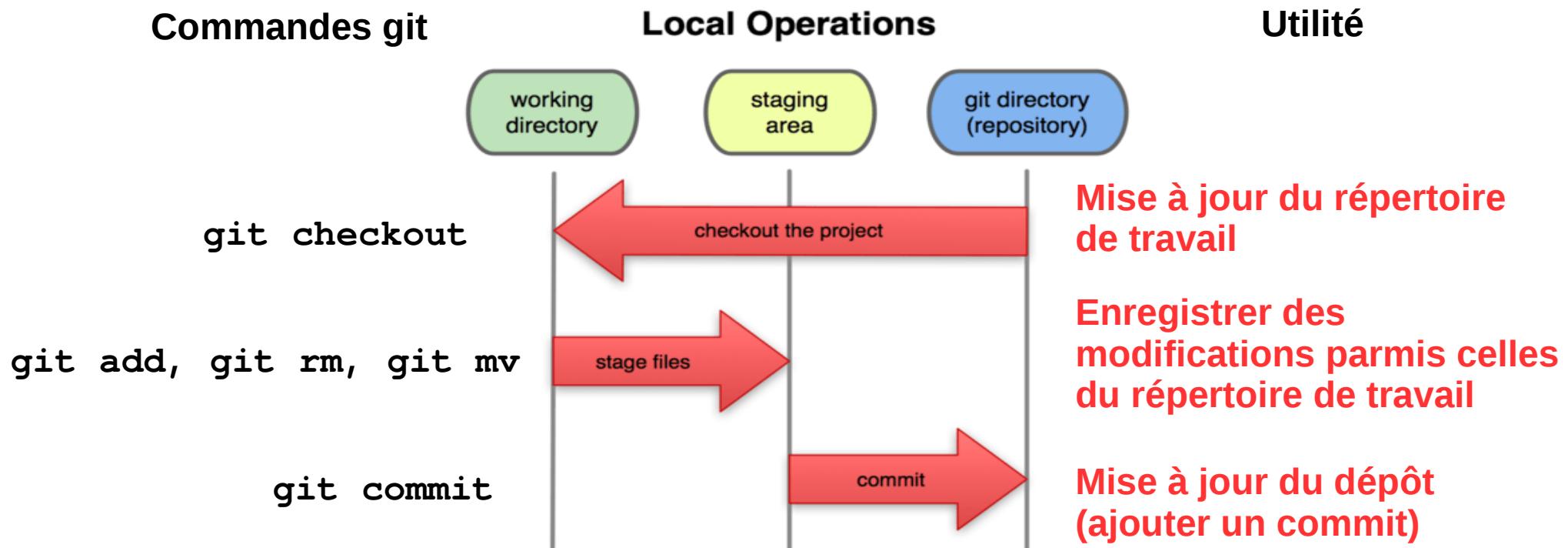
Concepts de GIT

- Créer un commit à partir de modifications
 - But : enregistrer des modifications entre le contenu du répertoire de travail (dossiers et fichiers) et le commit pointé par HEAD (dernier état enregistré).



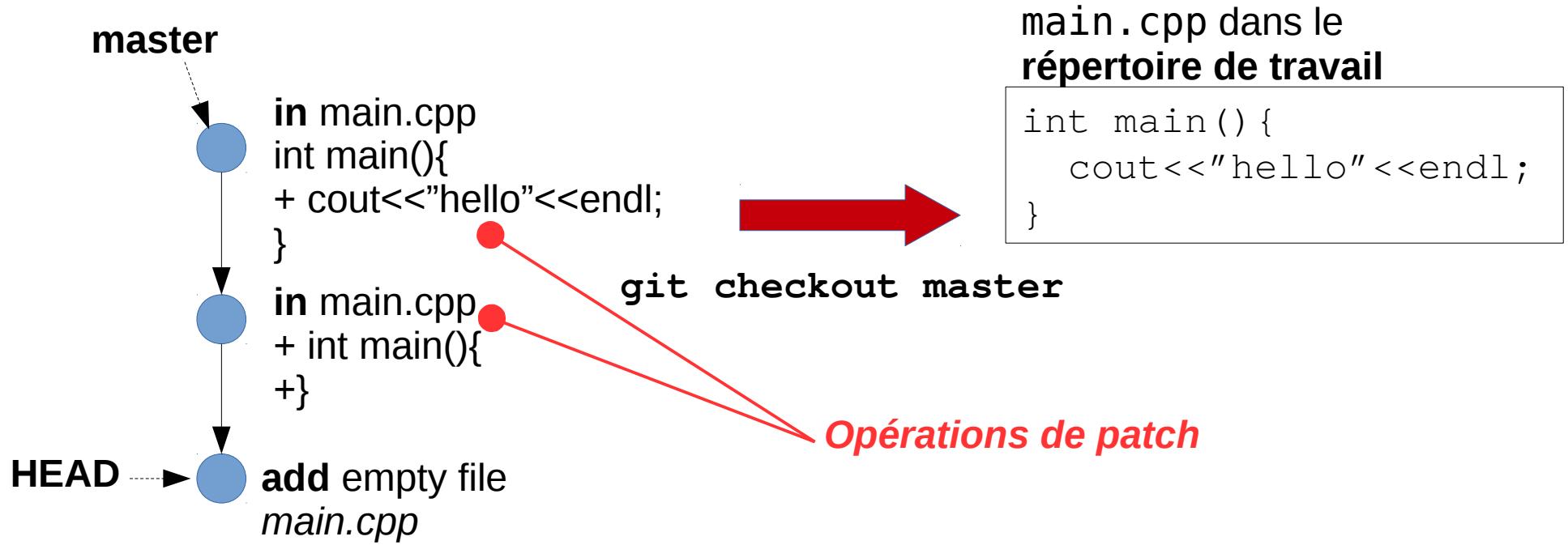
Concepts de GIT

- Créer un commit à partir de modifications
 - Originalité de git: 2 phases



Concepts de GIT

- Mise à jour du répertoire de travail
 - À chaque checkout ou merge.
 - Applique la séquence de modification (patches) depuis le premier jusqu'au commit courant (pointé par **HEAD**).



Concepts de GIT

- Enregistrement de modifications dans le répertoire de travail
 - Sélectionner les modifications qui seront contenu dans le prochain commit.
 - internes à des fichiers (utiliser `git add -p`) ; toutes les modifications d'un fichier ou nouveau fichier (utiliser `git add filename`)
 - Toutes les modifications (utiliser `git add --all`)
 - Désélectionner des modifications est possible (utiliser `git reset`)

main.cpp en **HEAD** dans le dépôt

```
int main() {  
    cout<<"hello"<<endl;  
}
```

`git add main.cpp`

main.cpp dans **répertoire de travail**

```
int main() {  
    cout<<"hello"<<endl;  
    return 0;  
}
```

Modification enregistrée
dans la **staging area**

`in main.cpp`

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

Opération diff pour déduire le patch.

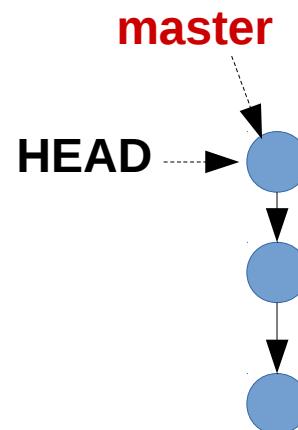
Concepts de GIT

- Mise à jour du dépôt local (« committer »)
 - Créer un nouveau commit à partir des modifications enregistrées dans la « *staging area* ».
 - Ajout d'un message expliquant ce qui a été modifié.

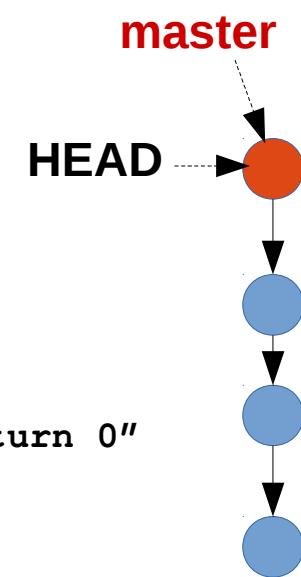
Modifications enregistrées
dans la **staging area**

```
add other.cpp  
in main.cpp  
in main.cpp  
  
cout<<"hello"<<endl;  
+ return 0;  
}
```

État courant du dépôt



git commit -m "return 0"

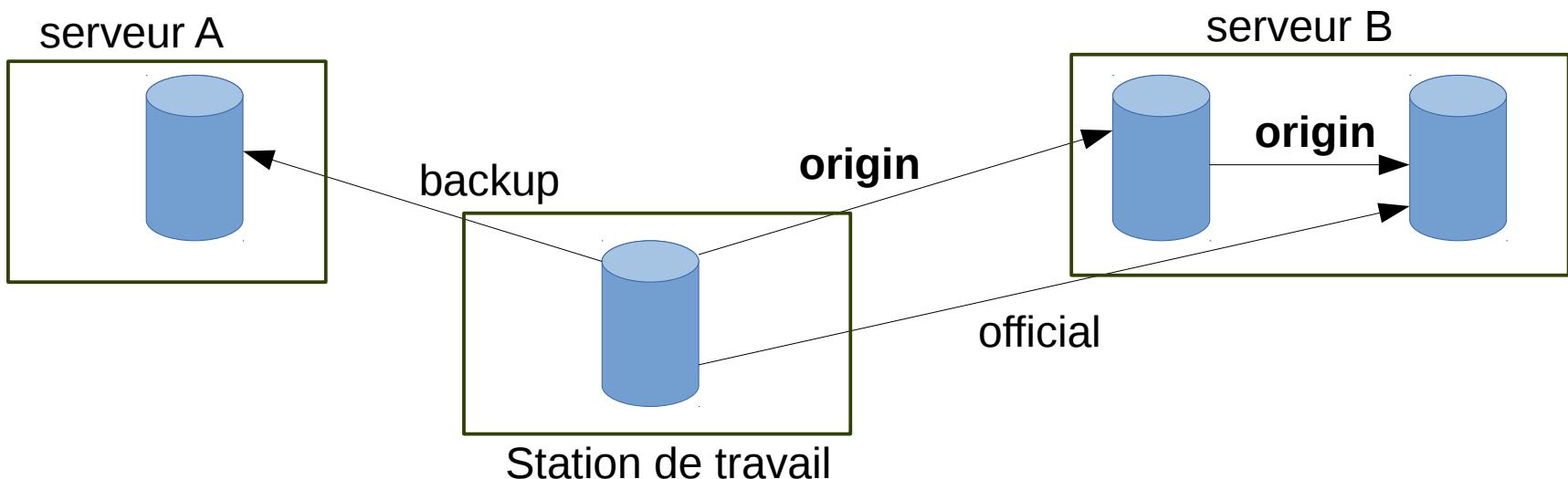


Concepts de GIT

- Pourquoi 2 phases pour créer des commits ?
 - créer de « petit commits » à partir d'un grand nombre de modifications.
 - retarder le commit de certains modifications: les fonctionnalités validées peuvent être « commitées » dessuite.
 - isoler le code qui ne sera jamais commité (e.g. les traces de debug).
- Une fois que les modifications intéressantes sont « commitées », nettoyer la staging area et répertoire de travail est possible :
 - Utiliser `git reset --hard`: suppression définitives des modifications non commitées.
 - Utiliser `git stash save`: modifications non commitées peuvent être restaurées.

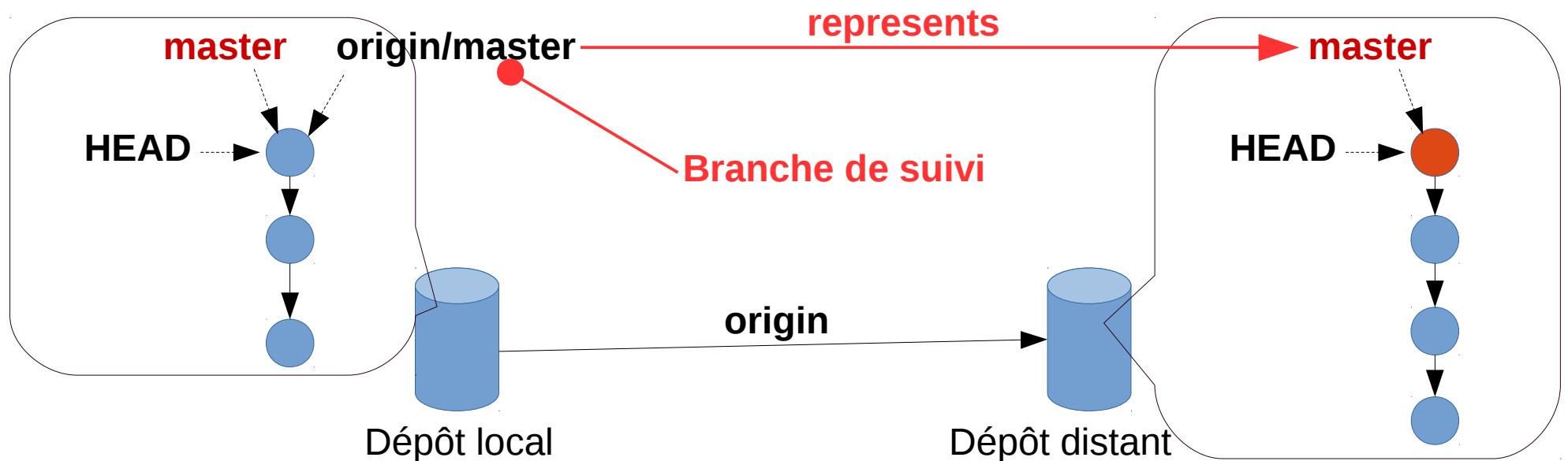
Concepts de GIT

- Synchronisation entre dépôt distants (*remote repositories*)
 - Chaque dépôt connaît un ensemble de dépôt appelés **remotes**.
 - Le *remote* par défaut est appelé **origin**.
 - Chaque *remote* est associé à un **nom** (unique dans le dépôt local) et définit une **url** (adresse d'un dépôt distant).
 - Ajouter un *remote* : `git remote add backup <address>`
 - Supprimer un *remote* : `git remote rm backup <address>`



Concepts de GIT

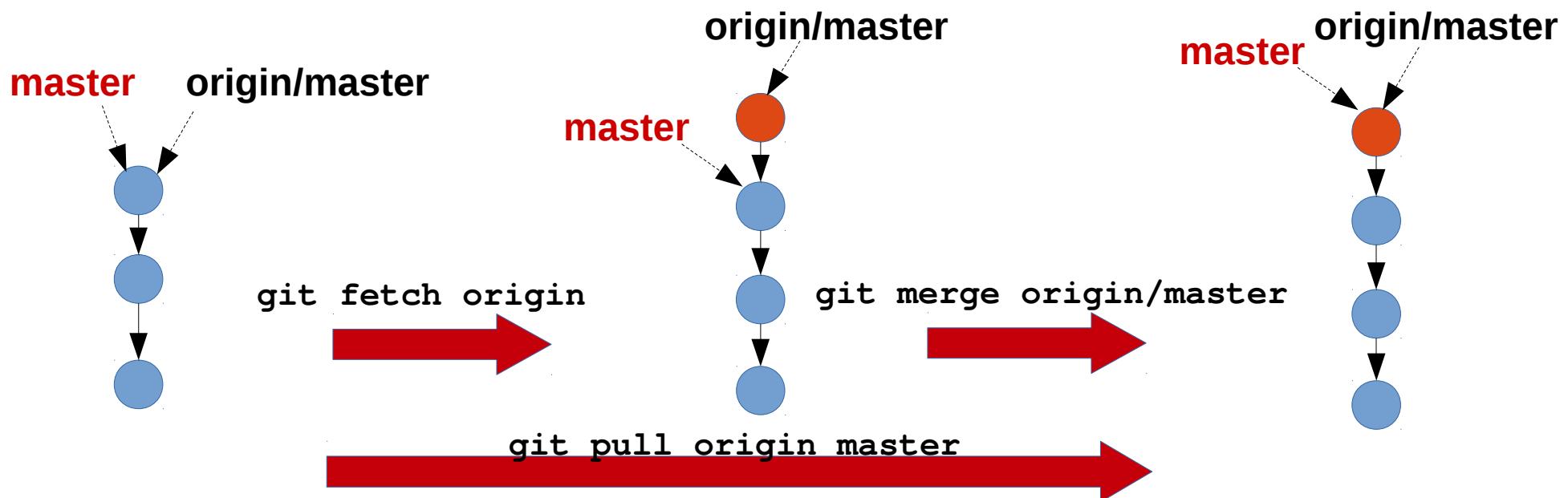
- Synchroniser les branches **locales** avec les branches **distantes**
 - Les branches des *remotes* sont connues dans le dépôt local mais elles sont en **lecture seule** (impossible de créer des commit sur ces branches). Elles sont appelées **branches de suivi (remote tracking branches)**.



35

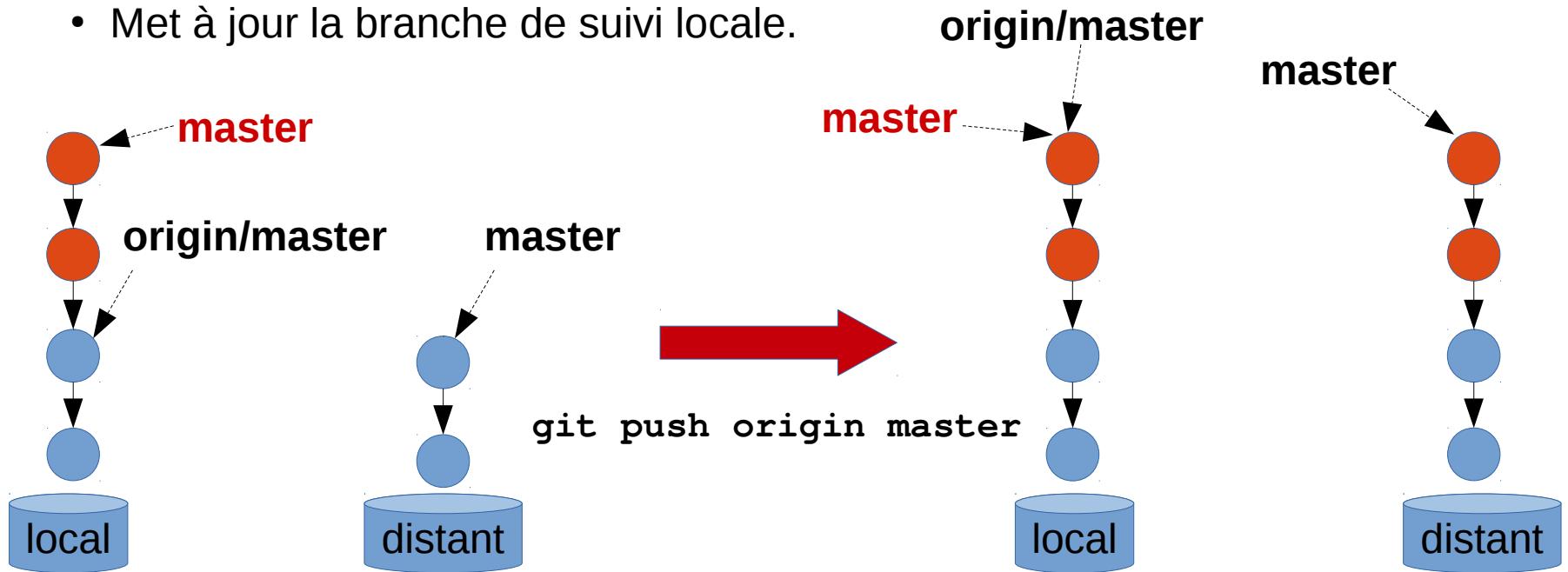
Concepts de GIT

- Mettre à jour un dépôt à partir d'un de ses *remotes*.
 - Mettre à jour (si besoin) le graphe des commits **local** et les **branches de suivi** (commande `fetch`).
 - Les branches locales sont **fusionnées** avec les branches de suivi (commande `merge`).
 - Commande tout-en-un: `pull`



Concepts de GIT

- Mettre à jour une branche distante à partir d'une branche locale
 - Utiliser la commande `push` (opération atomique).
 - Vérifie que la branche de suivi locale correspondante est à jour.
 - Met à jour le graphe des commits du dépôt distant ainsi que sa branche correspondante (de même nom).
 - Met à jour la branche de suivi locale.



Concepts de GIT

- **Cloner**, opération git la plus connue correspond à :
 - Créer un répertoire local et l'initialiser comme dépôt (`git init`).
 - Créer un *remote* appelé ***origin*** (`git remote add origin <address>`)
 - Créer une branche ***master*** locale (`git checkout -b master`)
 - Mise à jour de la branche ***master*** locale (`git pull origin master`).



Concepts de GIT

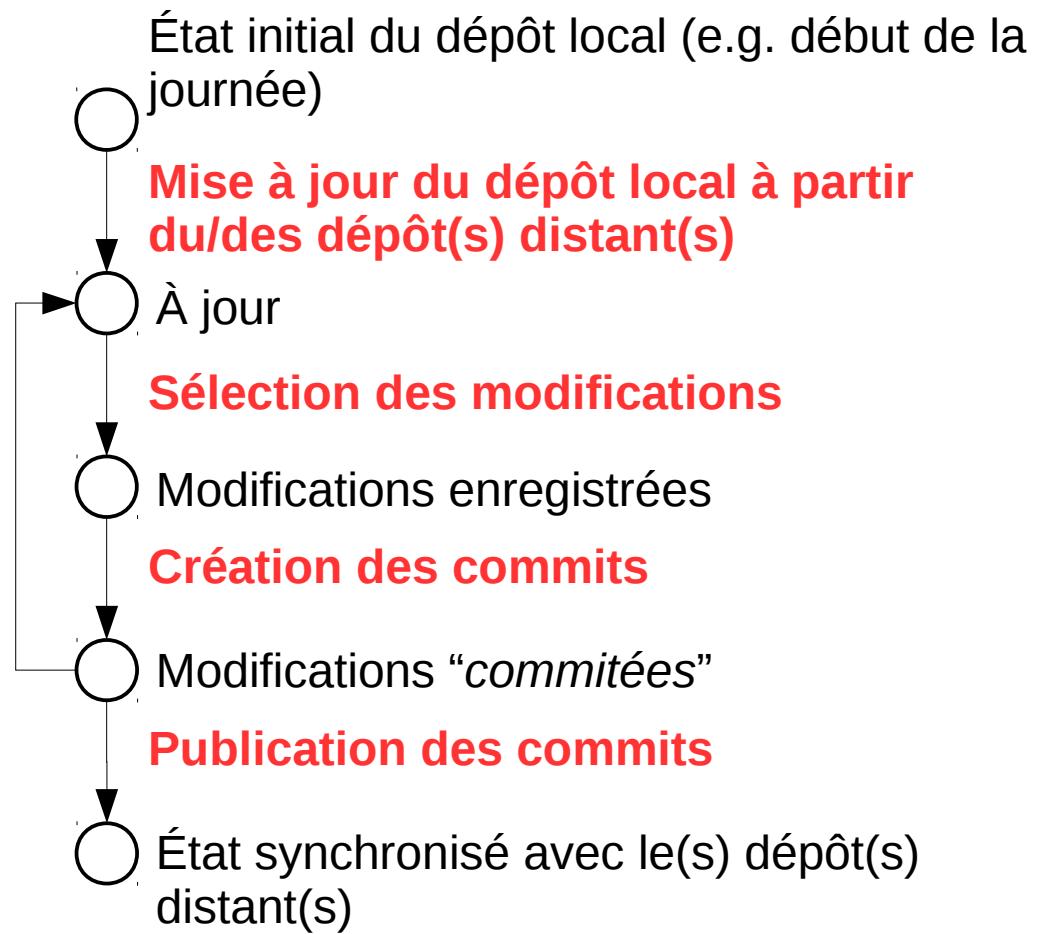
- Usage typique

git pull/fetch/merge ...

git add/rm/mv/reset ...

git commit ...

git push ...



Plan du cours

- La brève histoire des systèmes de contrôle de version (VCS)
- Les concepts de git
- **Les fonctionnalités de Gitlab**

Fonctionnalités de Gitlab

- Application web pour gérer des dépôts **côté serveur**
 - Création / suppression de dépôts.
 - Enregistrement et gestion des **droits d'accès** aux dépôts .
 - Outils graphiques pour visualiser et éditer : graphe des commits, branches, tags, fichiers, etc.
 - Documentation des projets (fichiers README)
 - Outils de communication de projets (**issues**)
 - Mécanisme de **fork** et **merge requests** (comme sur github).



Serveur Gitlab de l'IUT:

<https://gitlabinfo.iutmontp.univ-montp2.fr/>

Définitions

- *Projet* : dépôt git + autres informations (droit d'accès, wiki, *issues*, intégration continue, etc.).
- *Workspace* : **répertoire** contenant des dépôts git.
- *Utilisateur* : personne enregistrée sur le serveur. Chaque utilisateur a un **espace de travail (workspace) personnel** (à part les utilisateurs externes).
- *Groupe*: groupe d'utilisateurs travaillant sur un ensemble de projets communs. À chaque groupe correspond un **espace de travail (workspace)** contenant les dépôts de ces projets.

Accès direct à cette page

Page d'accueil

Paramètres Utilisateurs (clefs SSH)

The screenshot shows the GitLab homepage with several UI elements highlighted by red circles and annotations:

- Projects**: A button in the top navigation bar.
- Groups**: A button in the top navigation bar.
- Activity**: A button in the top navigation bar.
- Milestones**: A button in the top navigation bar.
- Snippets**: A button in the top navigation bar.
- Search or jump to...**: A search bar at the top right.
- Last updated**: A dropdown menu next to the search bar.
- New project**: A green button at the top right.
- Profile icon**: An icon in the top right corner.

Annotations in red text:

- Groupes auxquels j'appartiens**: Points to the "Groups" button in the top navigation bar.
- Recherche de projets**: Points to the search bar at the top right.
- Création de projets/groupes**: Points to the "New project" button at the top right.
- Liste des projets auxquels je participe**: Points to the "Your projects" section.

The main content area displays a list of projects:

Project Name	Type	Owner	Last Updated
rob-miscellaneous / cooperative-task-controller	Project	Owner	updated 40 minutes ago
multi-contact / mc_RTC	Project	Owner	updated 4 hours ago
rob-miscellaneous / eigen-extensions	Project	Owner	updated 1 day ago
mc-hrp4 / hrp4lirmm_doc	Project	Owner	updated 1 day ago
pid / pid-workspace	Project	Owner	updated 1 day ago
Robin Passama / pid-workspace	Project	Maintainer	updated 1 day ago
pioneer3DX / pioneer3DX-driver	Project	Owner	updated 1 day ago
rob-miscellaneous / ethercat-driver	Project	Owner	updated 4 days ago
raven2 / ultrasonix-interface	Project	Owner	updated 5 days ago

Page de groupe

Workspace du groupe

The screenshot shows the 'pid' group page on GitLab. The left sidebar has a red circle around the 'Members' link. A large red circle highlights the list of projects in the main content area. Red text annotations are overlaid on the image:

- Gestion des Membres (enregistrement, droit d'accès)** (Management of Members (registration, access rights)) points to the 'Members' link.
- Créer un nouveau projet dans le groupe** (Create a new project in the group) points to the 'New project' button.
- Accès aux projets contenus dans le groupe** (Access to projects contained in the group) points to the list of projects.

Project List:

- pid-workspace: This project defines a build and packaging system based on CMake and git.
- pid-rpath: A specific package of PID used to manage runtime path of executables and libraries.
- pid-config: Libraries and tools to manage configuration files.
- pid-log: Logging libraries and tools.
- ext-boost: A project to manage the port into PID system of installable versions of boost.
- ext-yaml-cpp: A project to manage the port into PID system of installable versions of yaml-cpp project.

Workspace
contenant le projet

Page de projet

The screenshot shows a GitLab project page for 'pid-workspace'. The left sidebar lists project details like 'Details', 'Activity', 'Cycle Analytics', 'Repository', 'Issues' (21), 'Merge Requests' (1), and 'Settings'. A red arrow from the text 'Workspace contenant le projet' points to the 'Repository' section. Another red arrow from 'Gestion des issues' points to the 'Issues' count. A red circle highlights the 'Repository' link. A red arrow from 'Accès aux informations détaillées sur le dépôt' points to the repository details area. A red circle highlights the repository icon. A red arrow from 'Visibilité du projet (public, interne ou privé)' points to the visibility dropdown. A red circle highlights the visibility icon. A red arrow from 'Adresse du dépôt git' points to the SSH URL. A red circle highlights the SSH icon.

pid-workspace

Project ID: 202

Files (8.6 MB) Commits (1,375) Branches (5) Tags (2) Readme LICENSE

Add Changelog Add Contribution guide Set up CI/CD

History Find file Web IDE

Name	Last commit	Last update
environments	clean headers with license	1 year ago
external	changed the structure of gitignoring to preserver hierarchy...	4 years ago
install	changed the structure of gitignoring to preserver hierarchy...	4 years ago

Informations sur le dépôt

The screenshot shows a Git repository interface with a sidebar on the left and a main content area on the right.

Left Sidebar (Repository Overview):

- File
- Commits
- Branches
- Tags
- Contributors
- Graph** (highlighted with a red circle)
- Compare
- Charts
- Issues (21)
- Merge Requests (1)
- Settings

Right Content Area:

Visualisation des fichiers (highlighted with a red circle) points to the sidebar item "Graph".

Information détaillée sur chaque commit (highlighted with a red circle) points to the list of commits on the right.

Graphe des commits (highlighted with a red circle) points to the commit graph visualization.

Commit Graph Visualization: A vertical timeline of commits on the "master" branch. The commits are numbered from 18 at the bottom to 24 at the top. A green line highlights a specific commit between numbers 18 and 20.

Commit List:

- sphinx comments for PID version utilities
- sphinx comments for documentation utilities
- sphinx comments for deployment utilities
- Merge branch 'master' into development
- solved BUG with remove command
- Solved BUG induced by change in `is_External_Package_Defined` signature
- sphinx comments for finding utilities
- sphinx comments for platform utilities
- sphinx comments for general utilities
- Merge branch 'development' into 'development'
- fixing function call
- sphinx comments for git utilities
- Prepare work on documentation of internal commands
- sphinx comments for framework API
- sphinx comments for external API
- solved previous merge problem (unknown cause)
- Merge branch 'master' into development
- solved issue #92 (potential problem at realease time due to package initialization at creation)

Les Issues

- Discussion entre développeurs/utilisateurs
 - Déclarer des BUGS, proposer des améliorations, etc.

The screenshot shows a GitLab interface for a project named "pid-workspace". The left sidebar is circled in red, highlighting the "Issues" tab which has 21 items. The main content area displays issue #93 titled "bug with alternative versions". A large red circle highlights the title "Discussion d'une issue". To the right, another red circle highlights the "Labels" section where "Need Tests" is listed. The status bar at the bottom right shows "47".

Associer des labels pour caractériser l'issue (e.g. BUG)

Discussion d'une issue

bug with alternative versions

Sonny a rencontré un problème très certainement dû à un bug au niveau de la gestion des versions alternatives. La situation était la suivante:

```
Dependencies of ati-force-sensor-driver, version 1.1.1
target platform: x86_64_linux_abi11 (type=x86, arch=64, os=linux, abi=abi11)
* posix
-----Release Mode-----
+ api-ati-daq: 0.2.1
+ comedilib: 0.3.0
+ dsp-filters: 0.2.1
+ eigen-extensions: 0.10.0
- eigen: 3.3.4
+ pid-network-utilites: 2.1.1
- boost: 1.55.0
+ pid-rpath: 2.1.1
- boost: 1.55.0
+ pid-os-utilites: 2.0.0
- yaml-cpp: 0.6.2
```

pid-network-utilites et pid-rpath dépendent tous les deux de boost et la version affichée est 1.55.0 pour les deux. Sauf qu'en fait, pid-network-utilites était configuré et compilé avec la 1.64.0, ce qui générait une erreur à la compilation de ati-force-sensor-driver du genre (j'ai plus l'erreur sous les yeux):

```
make: no rule to make target <path-to-external>/boost/1.55.0/lib/libboost_system.so.1.64.0
```

je viens d'essayer de reproduire le problème dans un nouveau workspace en faisant un make deploy package=ati-force-sensor-driver mais j'ai un problème différent finalement mais toujours lié aux alternatives:

```
CMake Error at /tmp/test-workspace/share/cmake/system/api/PID_Package_Finding_Functions.cmake:843 (message):
[PID] CRITICAL ERROR : impossible to find compatible versions of dependent
```

Todo

Add todo

Assignee

Robin Passama
@passama

Milestone

None

Time tracking

No estimate or time spent

Due date

No due date

Labels

Need Tests

Confidentiality

Not confidential

Lock issue

Unlocked

2 participants

Notifications

Reference: pid/pid-workspace...

Les Issues

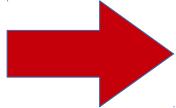
- *Board* : organiser le traitement des issues.

The screenshot shows a GitLab interface for a project named "pid-workspace". The left sidebar has a red circle around the "Board" option under the "Issues" section. The main area displays an issue board with three columns:

- Non traitées**: Contains 5 issues, including "Command to remove unused installed versions" and "Option to directly deploy the integration branch of a package".
- En cours**: Contains 5 issues, including "Little improvements:" and "About binaries compatibility".
- En test**: Contains 11 issues, including "PROBLEM with external packages version resolution" and "Better dependencies resolution when considering binaries and alternatives".

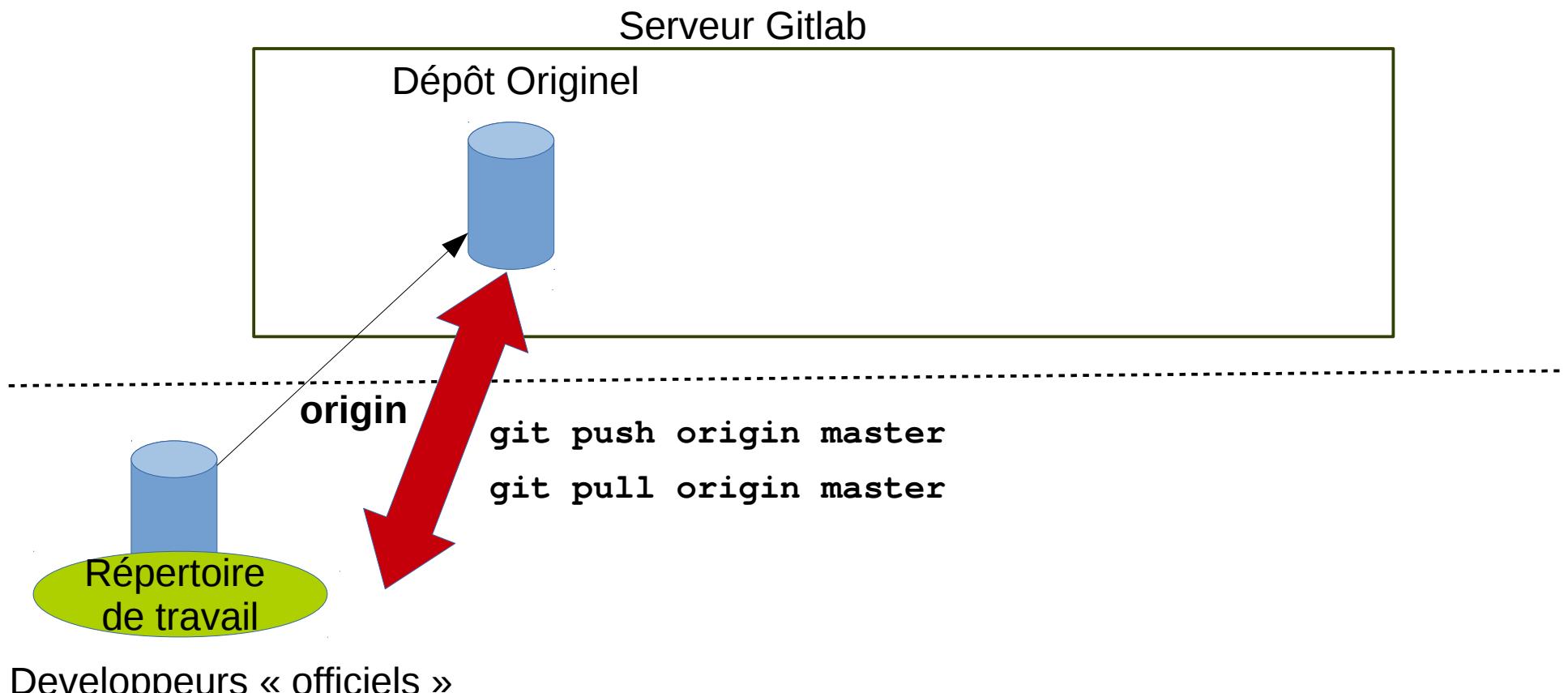
Each issue card includes a title, a "Work In Progress" status indicator, and labels like "enhancement", "bug", "confirmed", and "critical".

fork et merge request

- Work-flow « à la github »
 - Fork = cloner un dépôt directement dans Gitlab (le dépôt créé existe dans le serveur)
 - Le dépôt **originel** « connaît » ses dépôts **clonés**.
 - Merge request = proposer de fusionner une branche du dépôt **cloné** dans une branche du dépôt **originel**.
 - La requête peut être discutée, approuvée ou rejetée.
 - Possibilité de voir les changements proposés.
 - Seuls les développeurs du dépôt originel peuvent valider la requête.
-  Permet de recevoir des contributions de personnes extérieures au projet

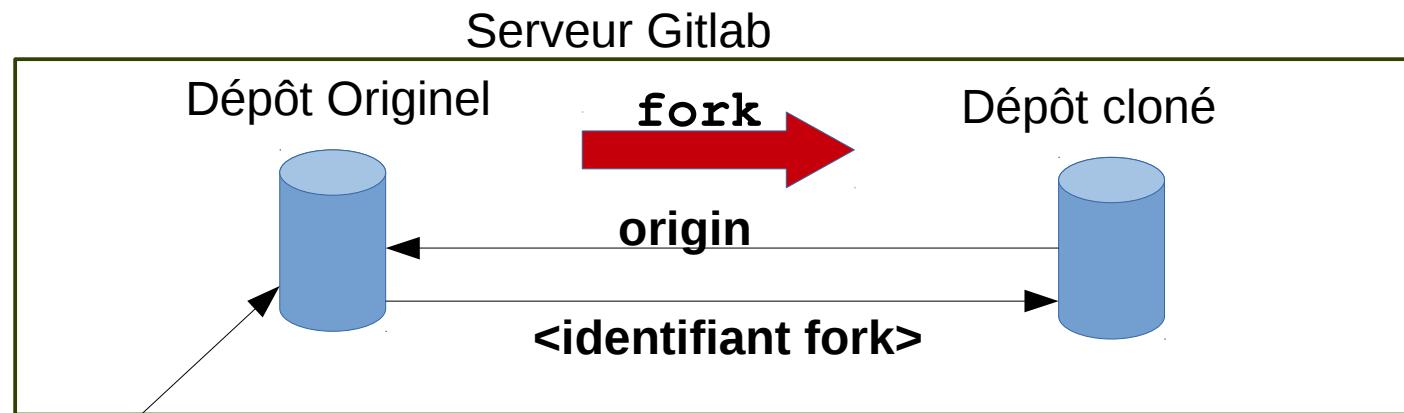
fork et merge request

- Situation initiale

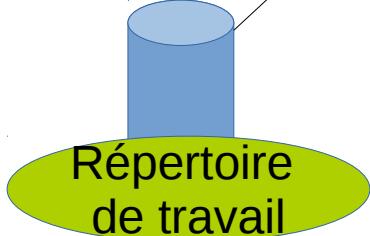


forker

- Cloner un dépôt dans un autre *workspace*



origin



Developpeurs « officiels »

forker

- Cloner un dépôt dans un autre workspace

The screenshot shows the GitLab interface for cloning a project. On the left, the sidebar shows the project 'doc-git'. A red arrow points from the word 'Forker le dépôt' to the 'Fork' button in the top navigation bar. Another red arrow points from the word 'Accès aux projets clonés' to the main project details page. A third red arrow points from the word 'Sélection du workspace dans lequel le dépôt cloné sera placé' to the 'Fork project' dialog on the right, which lists various workspace namespaces like 'hilecop' and 'contract'.

Forker le dépôt

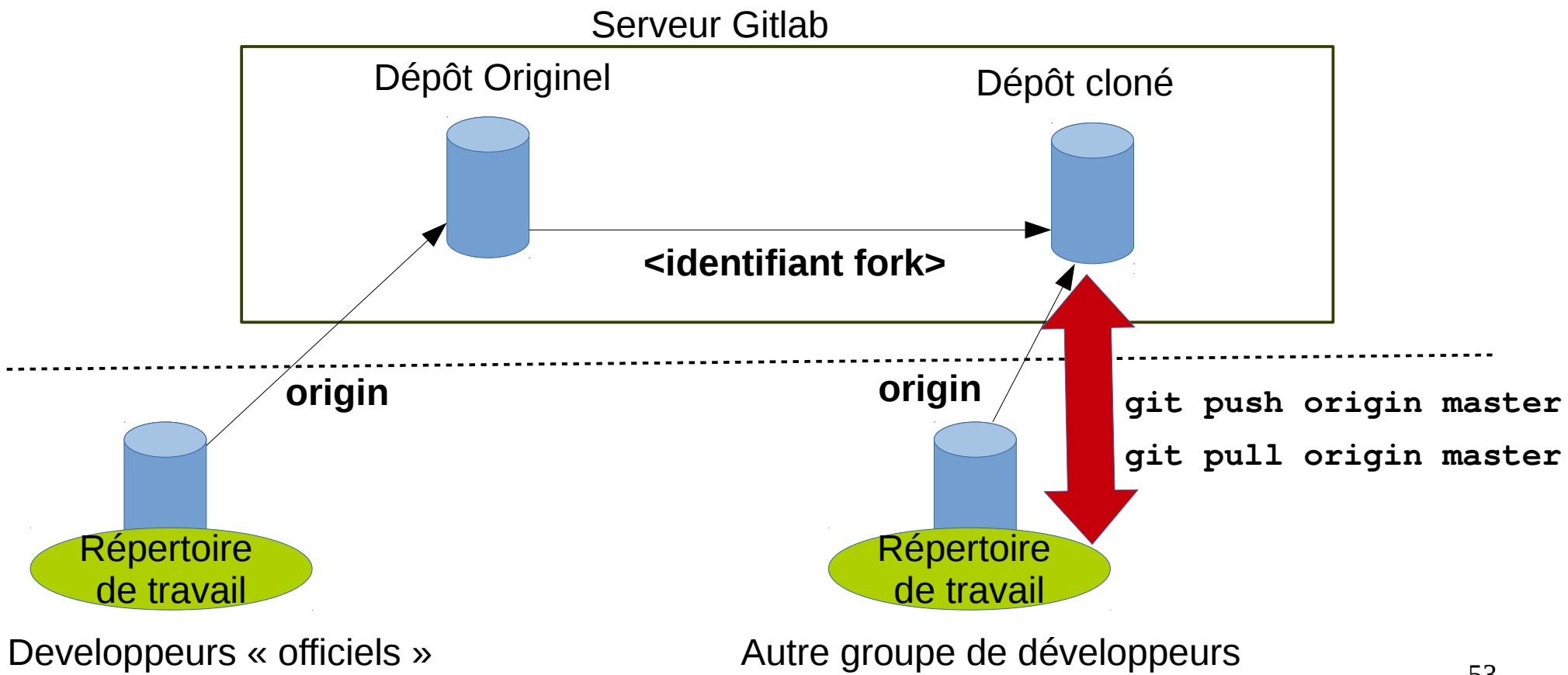
Accès aux projets clonés

Sélection du workspace dans lequel le dépôt cloné sera placé

fork

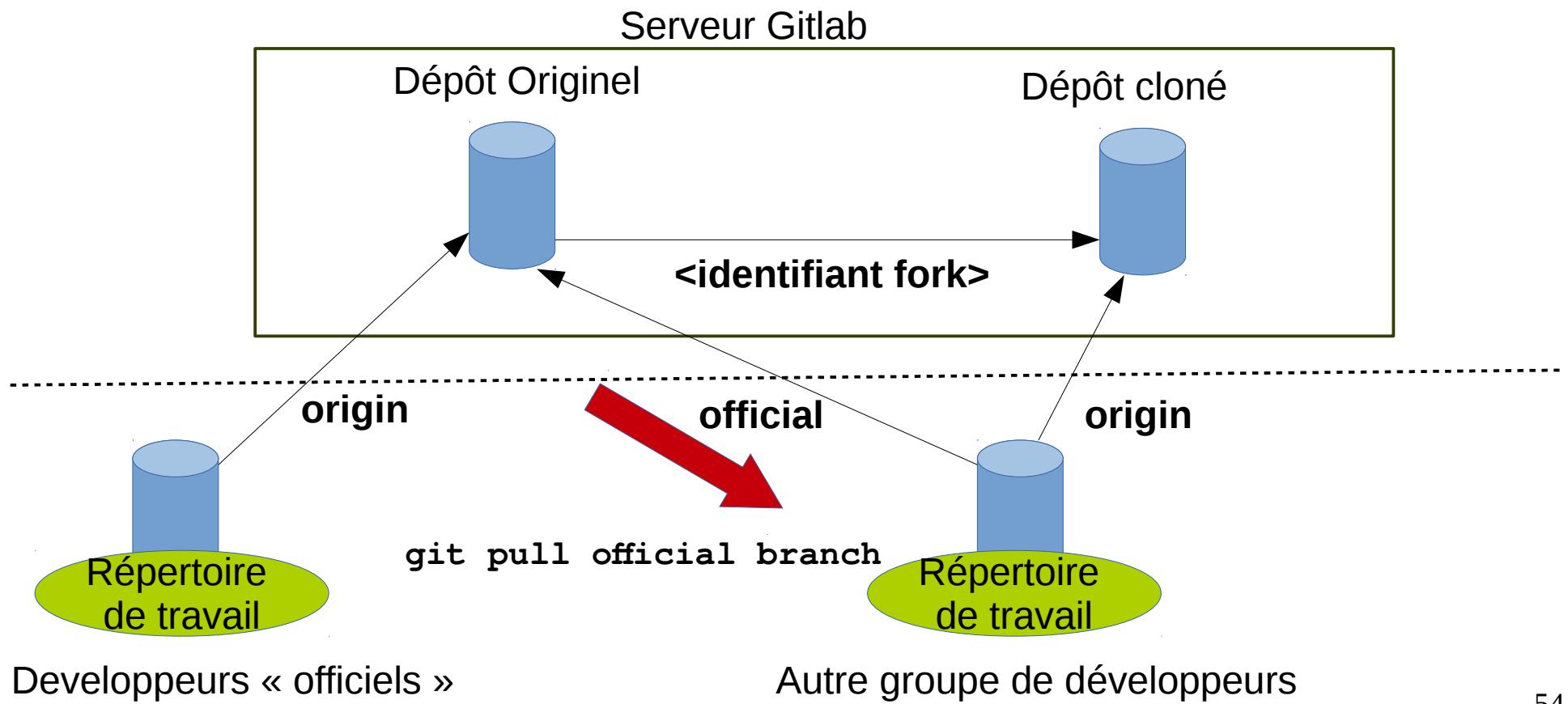
forker

- Développeurs extérieurs travaillent de manière isolée



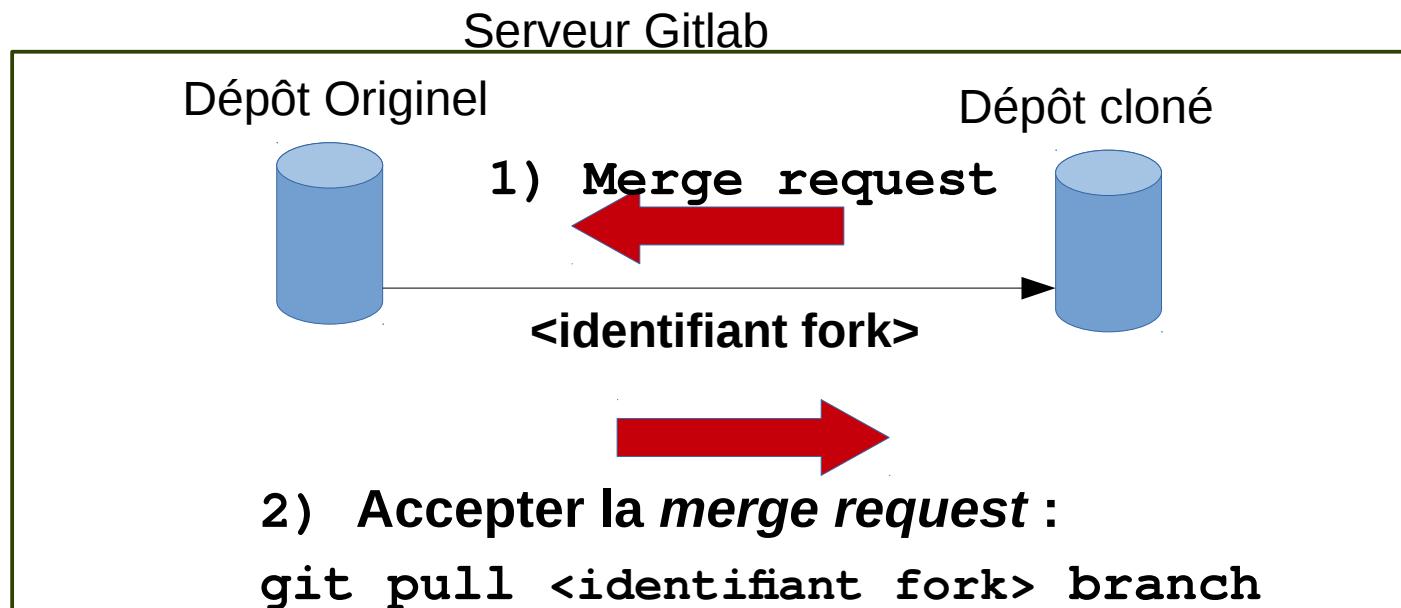
forker

- mettre à jour un dépôt cloné à partir du dépo originel



merge request

- Proposition de **fusion de branches** depuis dépôt cloné vers dépôt originel
 - Développeurs du dépôt cloné proposent la fusion d'une branche.
 - Développeurs du dépôt originel décident si la fusion est réalisée.



Proposer la fusion

1) Sélectionner les branches source (du dépôt cloné) et cible (originel)

The screenshot shows the 'Merge Requests' section of a GitLab project named 'doc-git'. A red circle highlights the 'Merge Requests' button in the sidebar. Red arrows point from the text 'Détailler sa merge request' to the 'Compare branches and continue' button at the bottom and to the 'Source branch' and 'Target branch' dropdown menus. The 'Source branch' is set to 'passama/doc-git' and 'master'. The 'Target branch' is set to 'common-docs/doc-git' and 'master'. Below these fields, two merge requests are listed:

- Update README.md** by Robin Passama, authored 34 seconds ago. SHA: 4bba75fe.
- writing README that points to wiki** by PID GitlabRunner, authored 2 years ago. SHA: 2d718891.

Détailler sa merge request

Proposer la fusion

New Merge Request

From passama/doc-git:master into common-docs/doc-git:master

Change branches

Title

Update README.md

Start the title with **WIP:** to prevent a **Work In Progress** merge request from being merged before it's ready.

Add description templates to help your contributors communicate effectively!

Description

Write Preview

bla bla bla

Explication de la *merge request*

Markdown and quick actions are supported.

Attach a file

Assignee Robin Passama

Milestone Milestone

Labels Labels

Source branch master

Target branch master

Change branches

Squash commits when merge request is accepted. [About this feature](#)

Contribution

Allow commits from members who can merge to the target branch. [About this feature](#)
Not available for protected branches

Submit merge request

Cancel

Commits 1 Changes 1

05 Sep, 2018 1 commit

Update README.md
Robin Passama authored 3 minutes ago



Assignee : développeur du dépôt originel en charge de traiter la *merge request*

Commits qui seront ajouté à la branche originelle si la *merge request* est acceptée

Accepter ou refuser la fusion

The screenshot shows a GitLab merge request interface for a project named "doc-git". The merge request is titled "Update README.md" and was opened by Robin Passama 15 seconds ago. The source branch is "passama:master" and the target branch is "master". The interface includes sections for "Origine de la requête" (Request origin), "Accepter la requête (fusionner)" (Accept the request (merge)), "Commits et modifications proposés" (Proposed commits and changes), "Fil de discussion entre développeurs" (Developer discussion thread), and "Refuser la requête (fermer)" (Reject the request (close)). Red circles and arrows highlight specific UI elements like the merge button, commit message, and close request button.

Origine de la requête

Accepter la requête (fusionner)

Commits et modifications proposés

Fil de discussion entre développeurs

Refuser la requête (fermer)

Conclusion

- Les concepts et commandes de base ont été montrés...
- ... mais il y a quantité de raffinements à découvrir !!
- Gitlab permet de gérer des dépôts git hébergés sur un serveur ...
- ... mais il permet de faire beaucoup d'autres choses : intégration & déploiement continu ; génération de sites statiques ; édition de wiki ; etc.

Merci pour votre attention