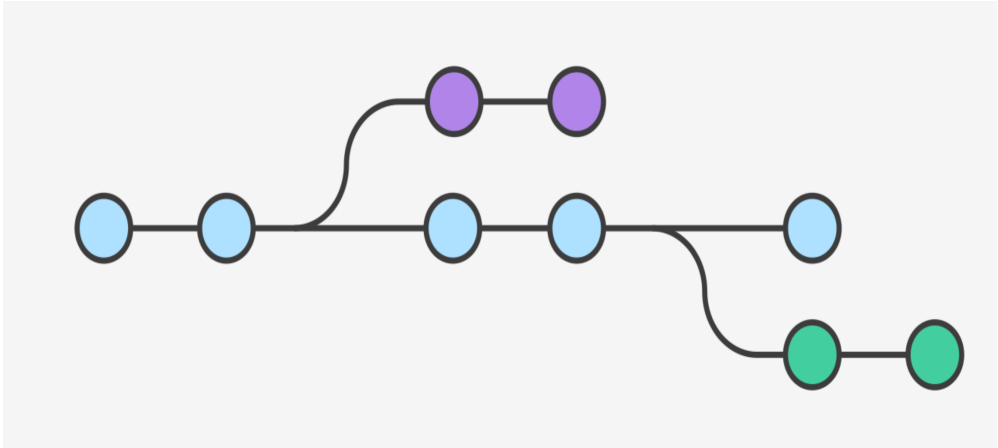
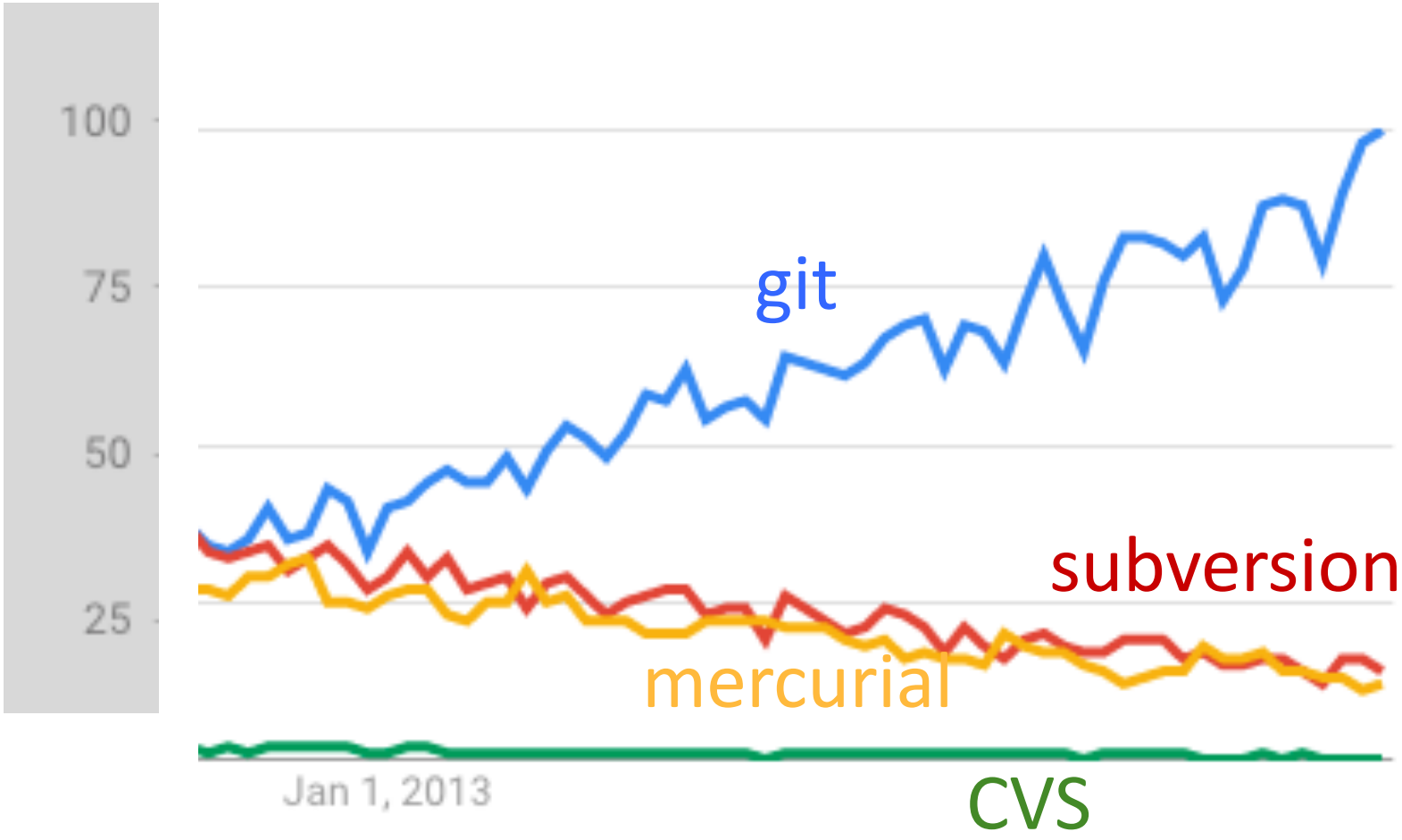
Bonnes Pratiques



Libre adaptation du support de David Parsons, INRIA et du support d'A. Cordier et P.-A. Champin, IUT Lyon 1

V. Deslandres
CVDA semestre 2 – IUT de LYON
Université Lyon1

Pourquoi Git ?



Source : https://trends.google.com

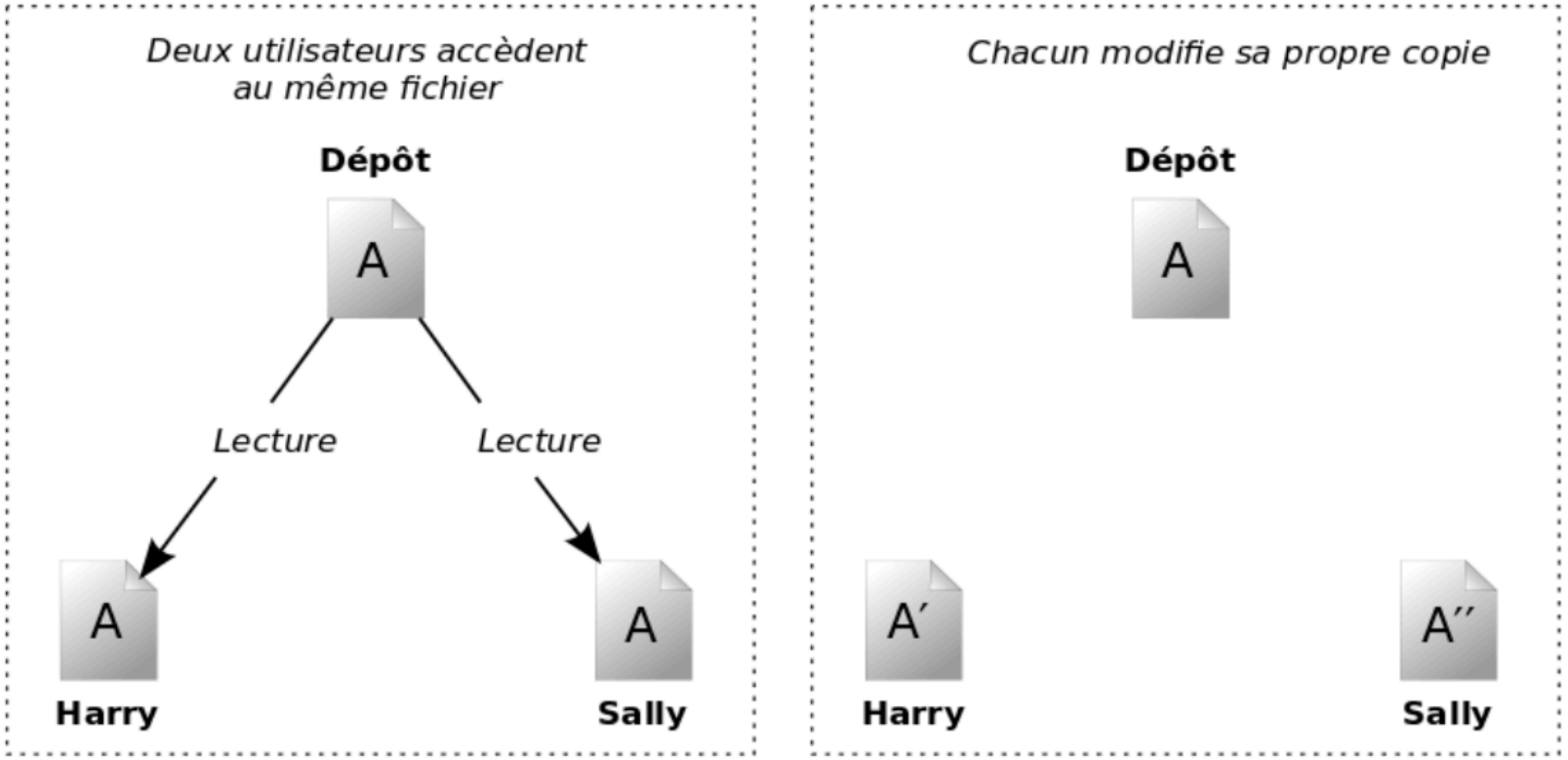
Pourquoi Git ?

Voici différents **points forts** de git (fév 2017) :

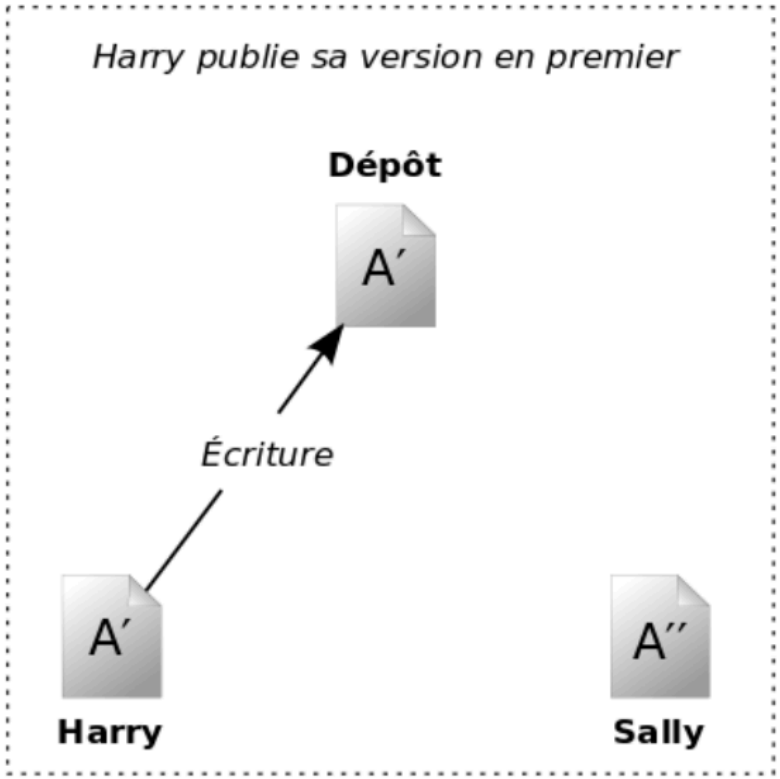
- Espace disque utilisé très faible comparé à ses concurrents
- Le système décentralisé est très flexible et répond à beaucoup de problèmes non résolus par les autres systèmes centralisés (branches)
- Licence **GNU** : git appartient au domaine publique, il est aussi très « suivi » (mis à jour).
- Git est de loin le système le plus populaire et il existe de nombreux services associés très populaires (Github, Gitlab, …)

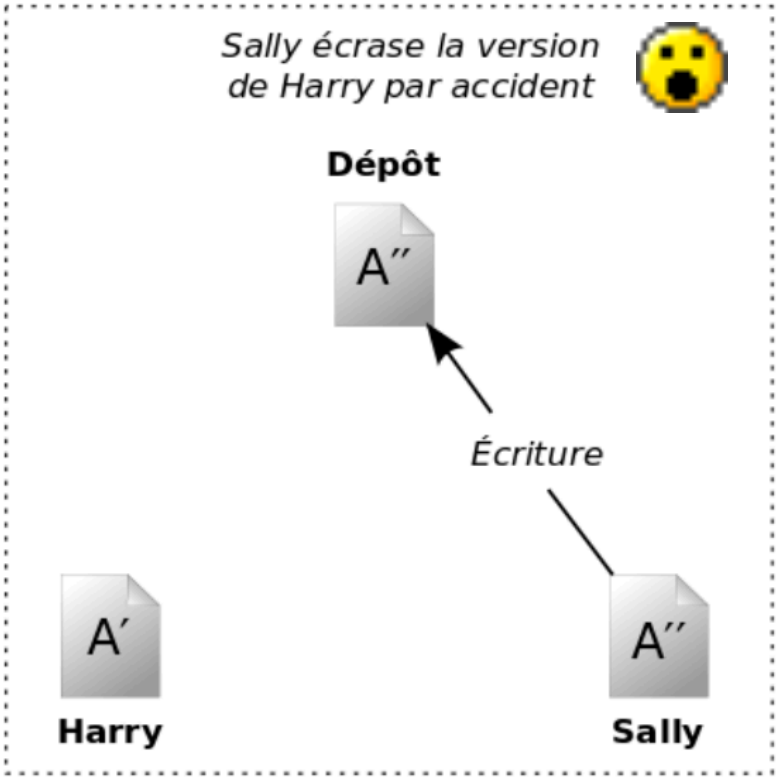
Le problème des SDB

- (Systèmes de dépôt tout bêtes) type Pydio, DropBox



Dropbox

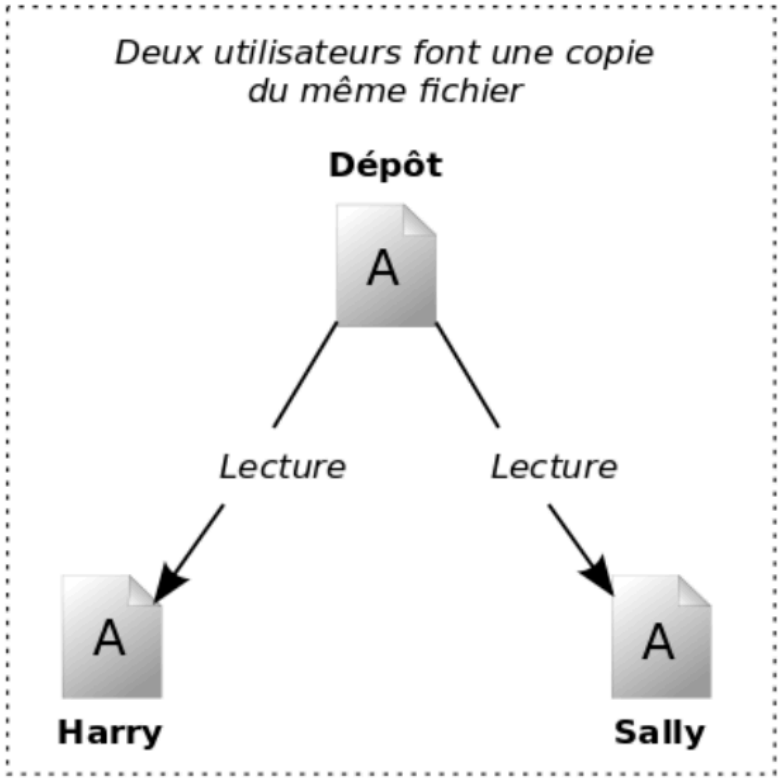

Harry publie sa version en premier
Dépôt
A′
Écriture
A′
A″
Harry
Sally

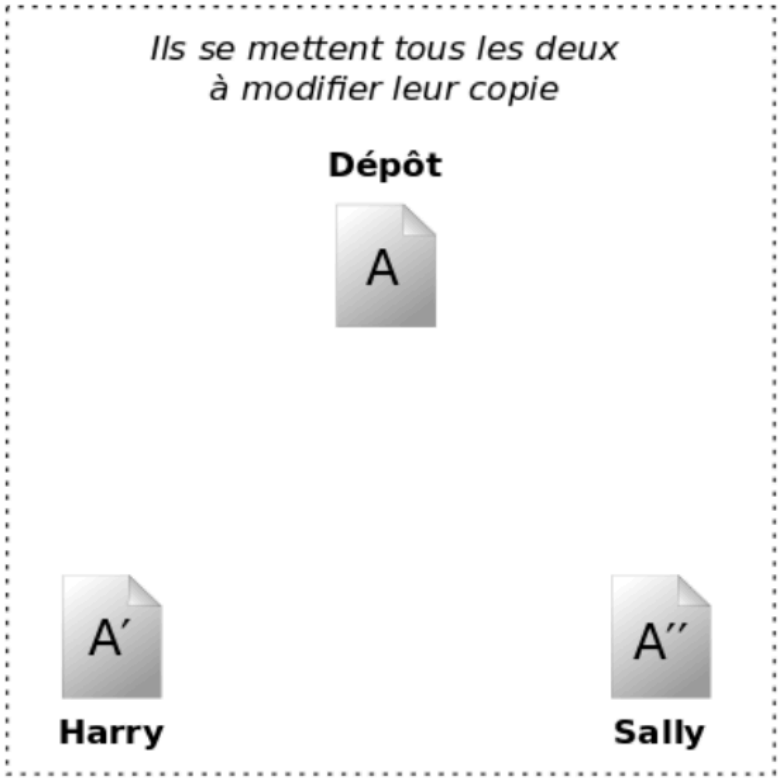

Sally écrase la version
de Harry par accident
Dépôt
A″
Écriture
A′
A″
Harry
Sally

Dropbox

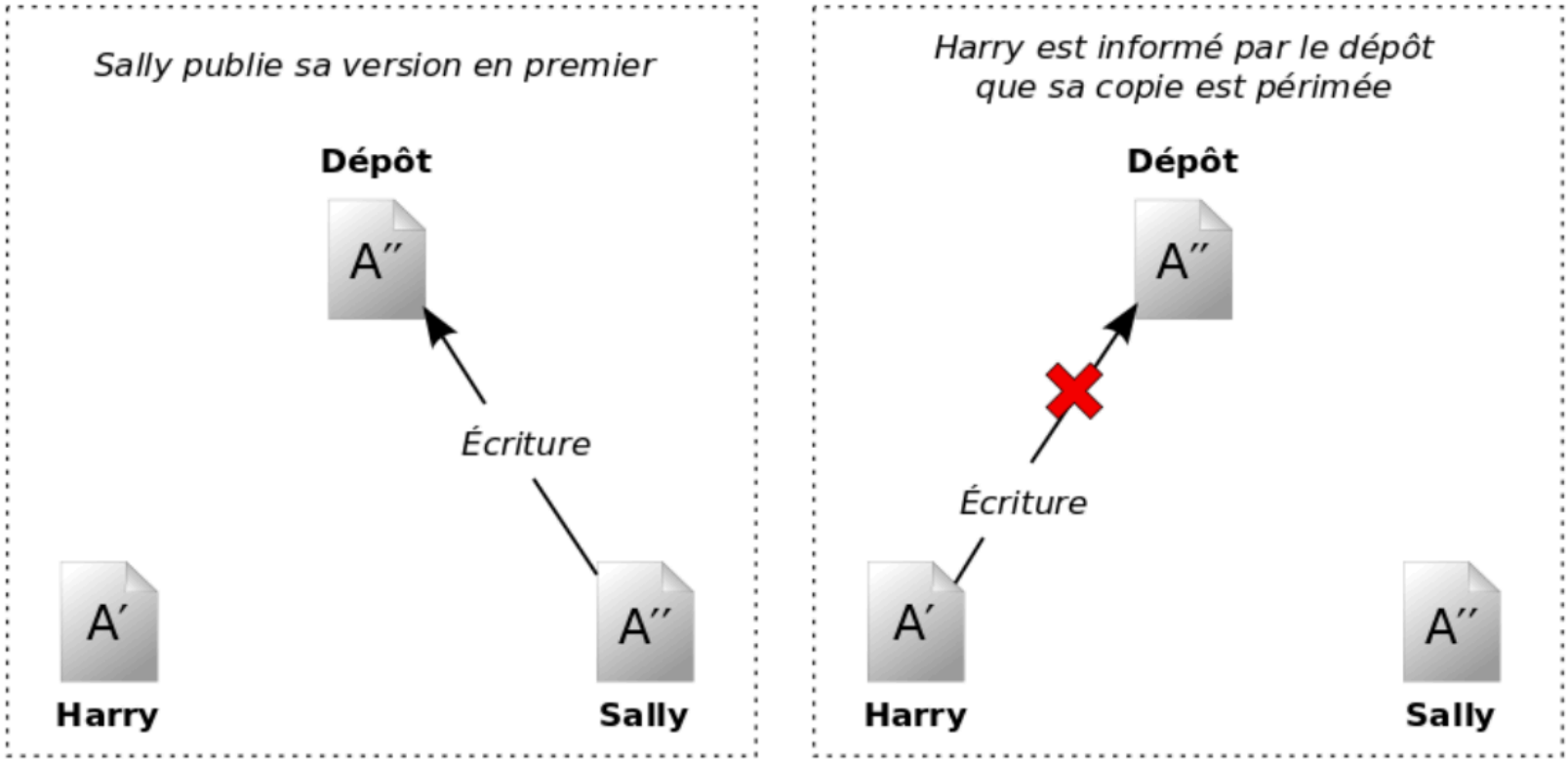
Avec un VCS (1)

Version Control System

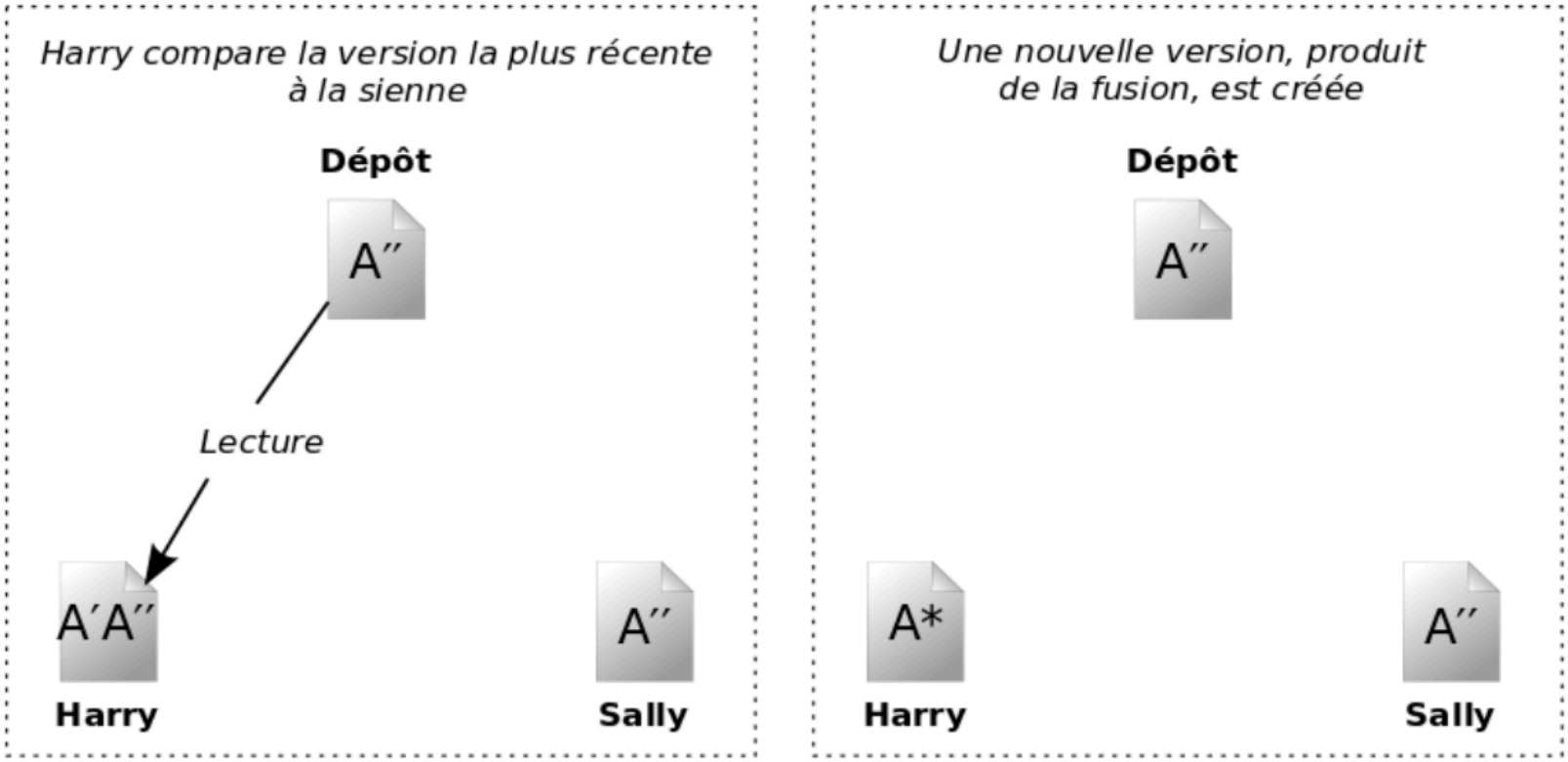




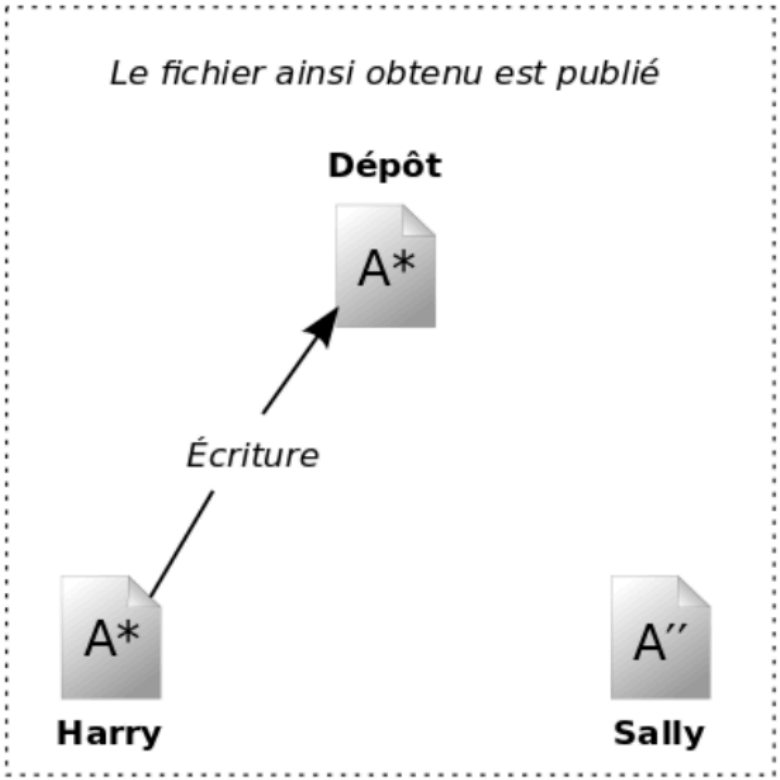
Avec un VCS (2)

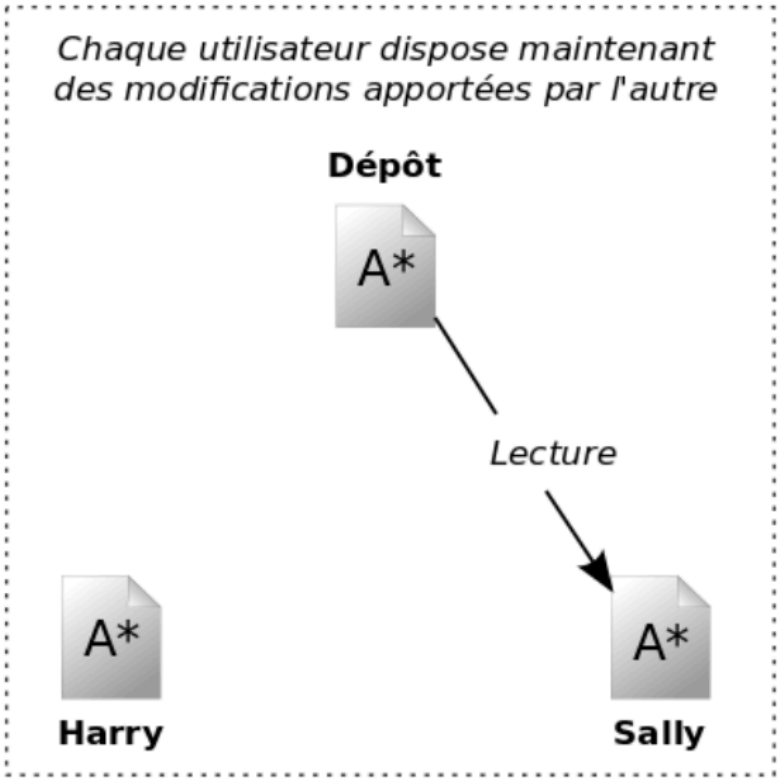

Sally publie sa version en premier
Dépôt
A″
Écriture
A′
A″
Harry
Sally
Harry est informé par le dépôt
que sa copie est périmée
Dépôt
A″
Écriture
A′
A″
Harry
Sally

Avec un VCS (3)

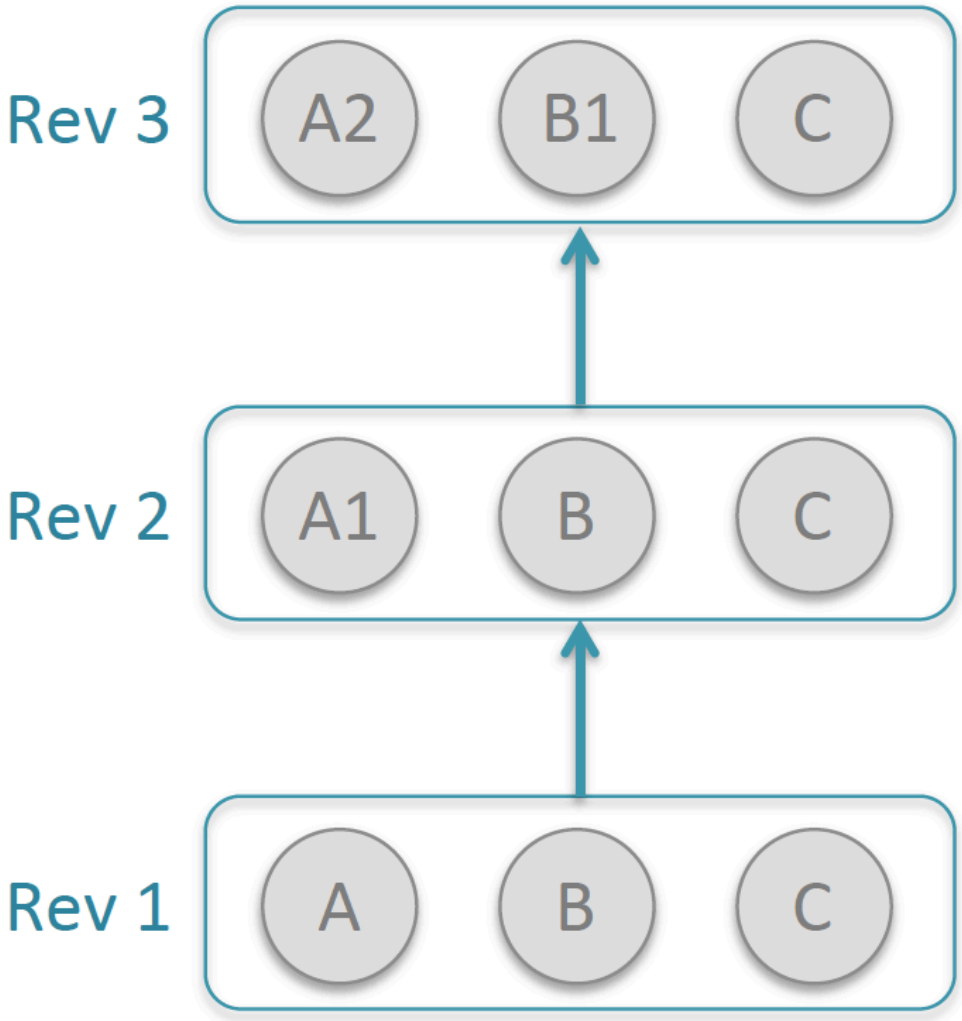

Harry compare la version la plus récente à la sienne
Dépôt
A″
Lecture
A′A″
A″
Harry
Sally
Une nouvelle version, produit de la fusion, est créée
Dépôt
A″
A*
A″
Harry
Sally

Avec un VCS (4)


Le fichier ainsi obtenu est publié
Dépôt
A*
Écriture
A*
A''
Harry
Sally

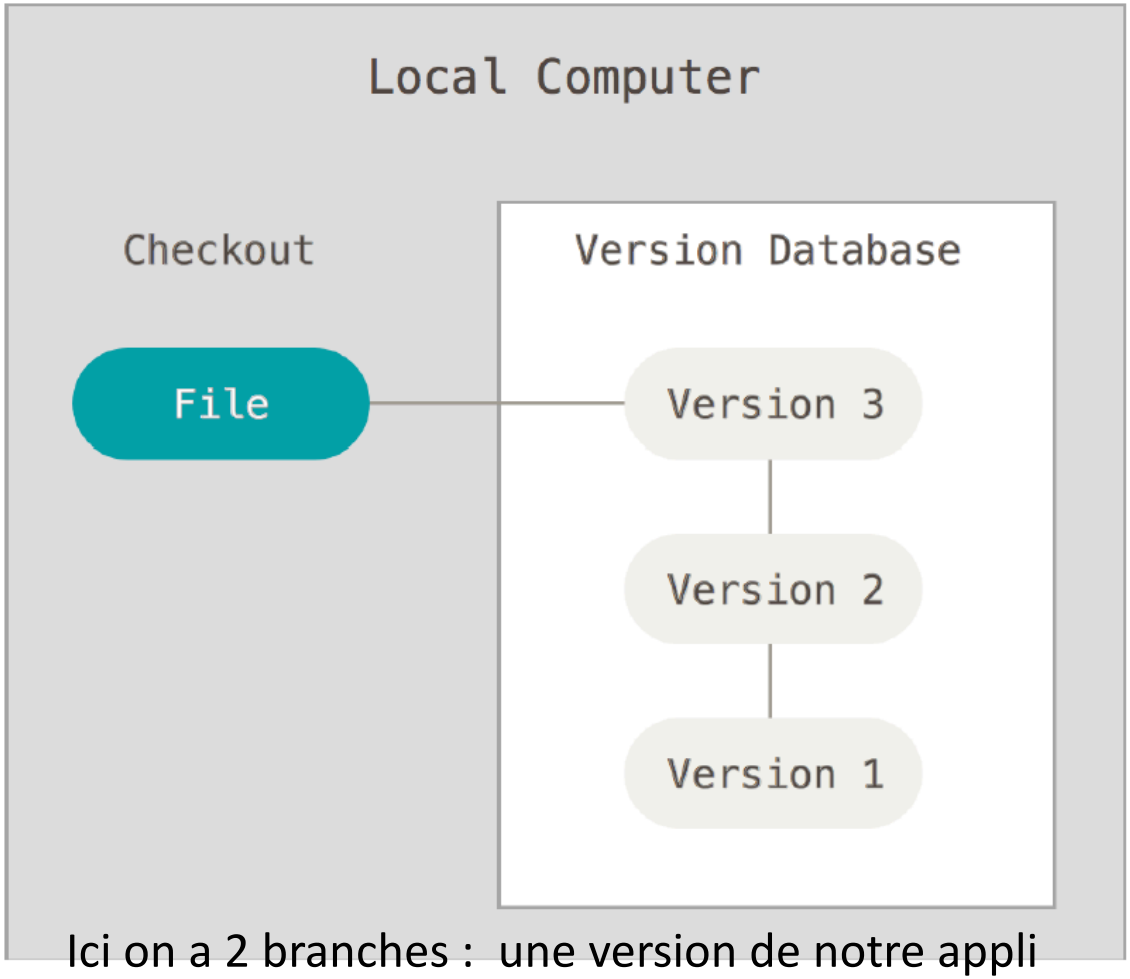

Chaque utilisateur dispose maintenant des modifications apportées par l'autre
Dépôt
A*
Lecture
A*
A*
Harry
Sally

Dépôt et Révision



- Dépôt (repository)
  - Contient **l'historique** complet du projet (ie toutes les révisions)
- Une révision (commit)
  - Un état des fichiers suivis, avec leur **différence** par rapport à une autre révision (antérieure)
  - Propre à un **auteur** et une **date**
  - Contient un **msg** expliquant la révision

Vue Locale

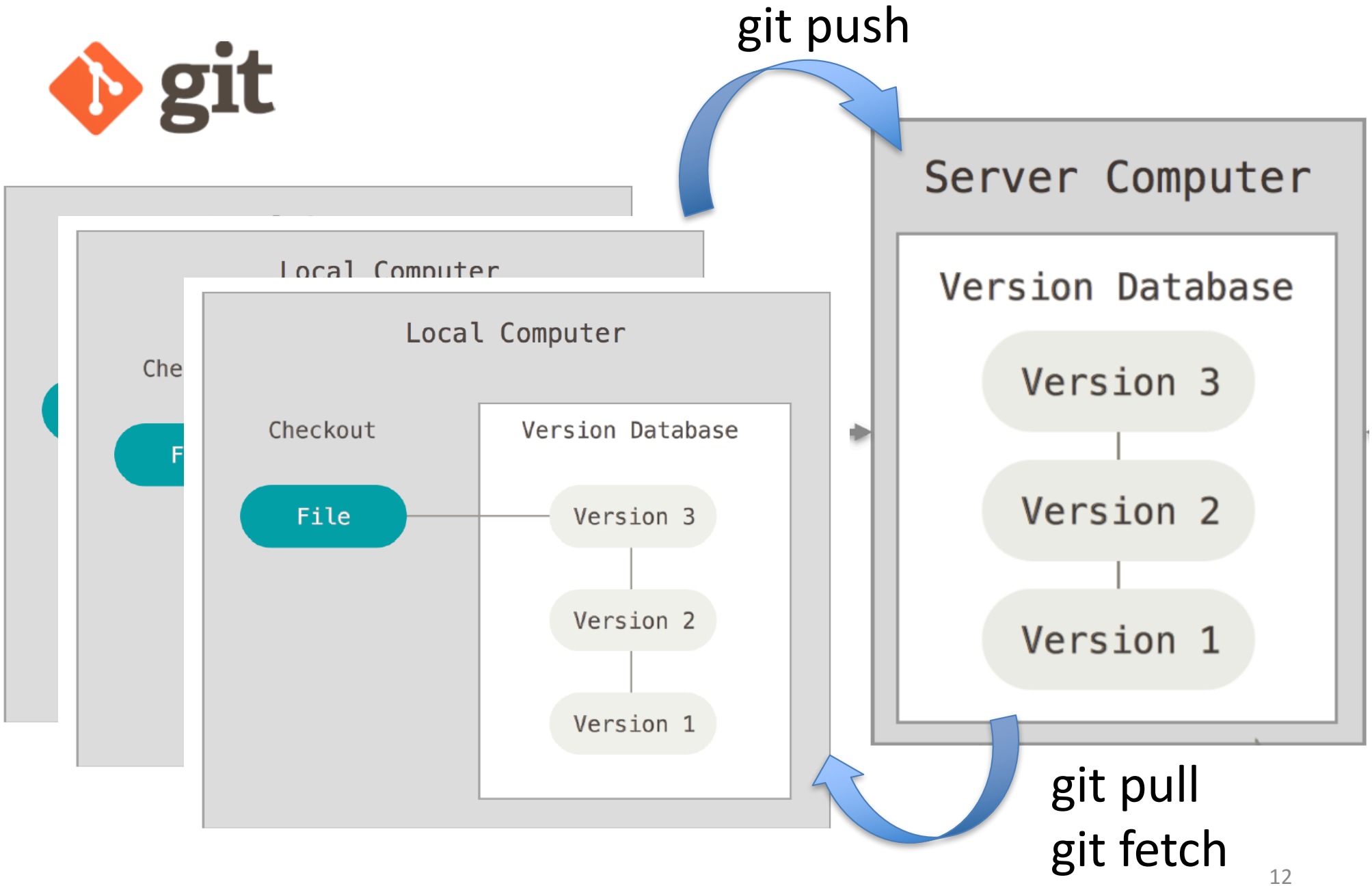


Répertoire (caché)
.git

dans lequel on a une copie du dépôt

Ici on a 2 branches : une version de notre appli avec une gestion simple de fichiers, et une avec un SGBD.

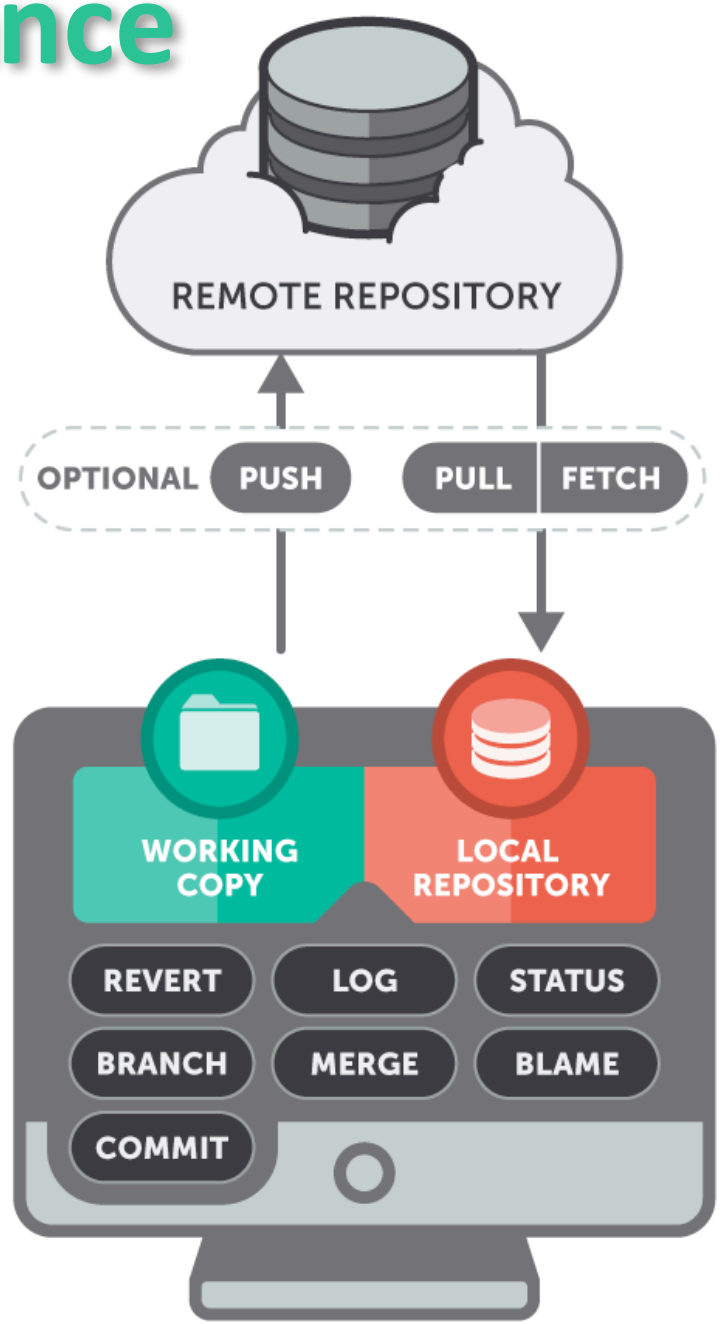
Lien éventuel avec un Dépôt distant



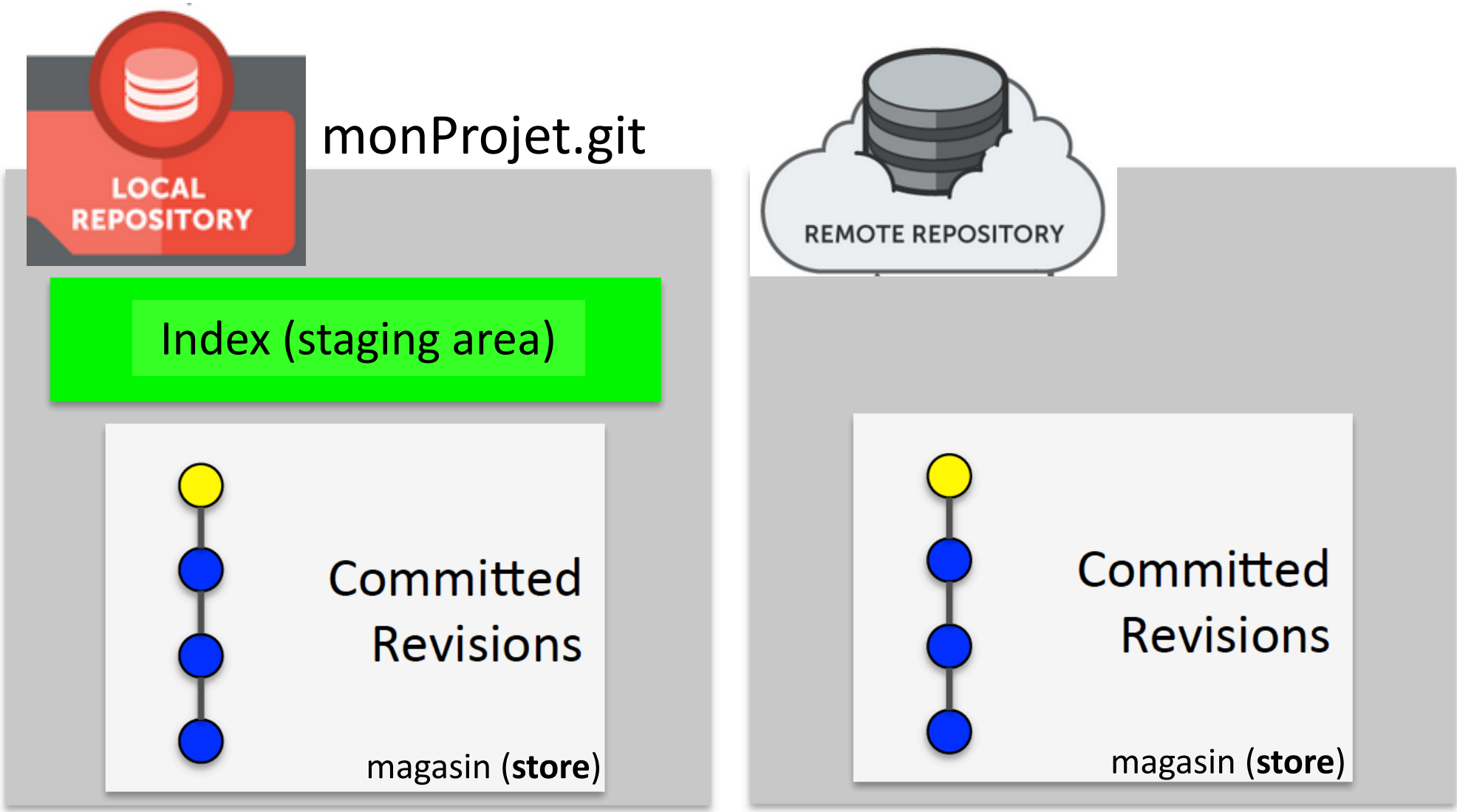
Vision locale / à distance

- CVS *décentralisé* :

  On dispose d'une copie intégrale du dépôt, en local
- Espace de travail = fichiers suivis ou non



Vision locale / à distante



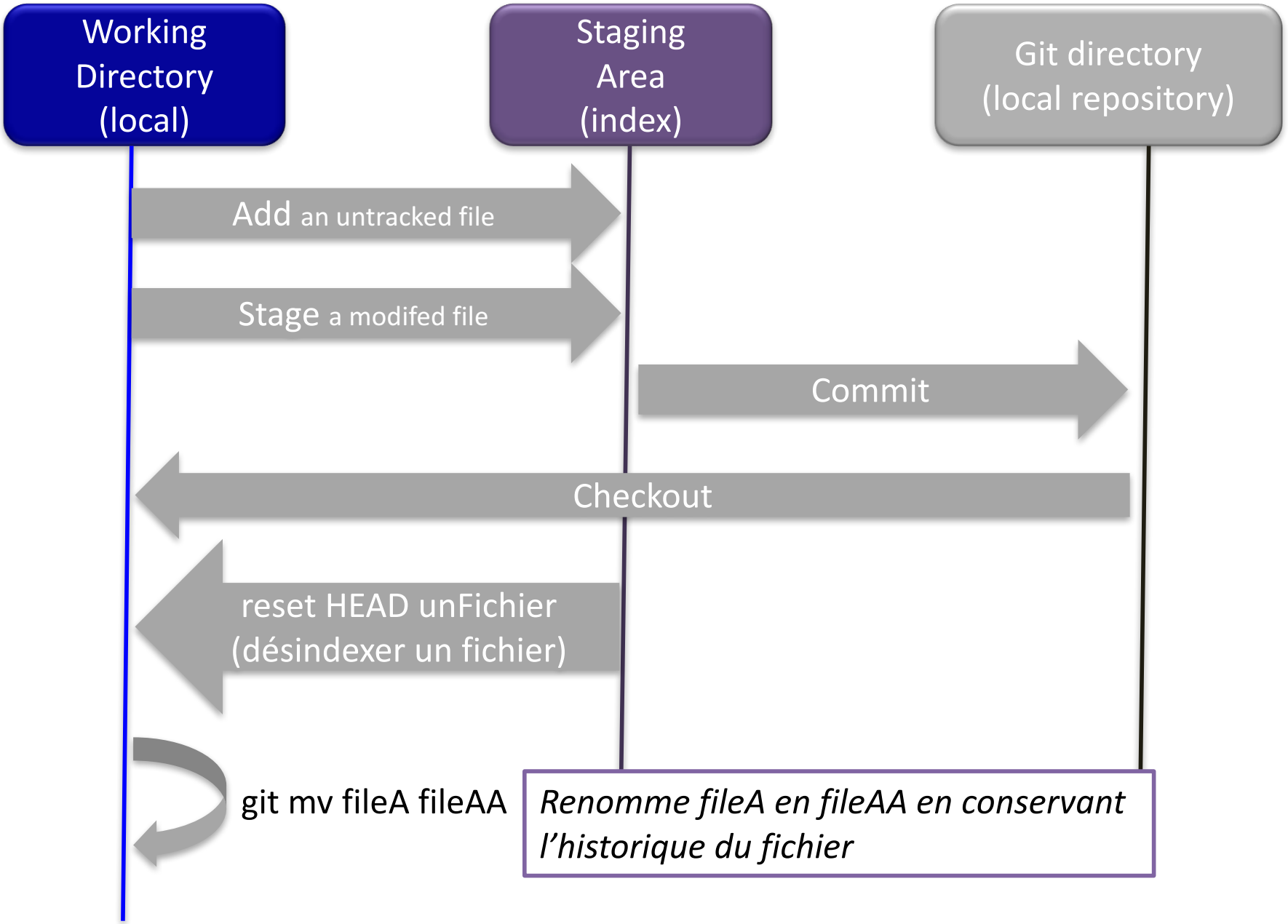
Créer un dépôt

- Pour un **nouveau** projet
  - (mkdir monProjet)
  - cd monProjet
  - git init

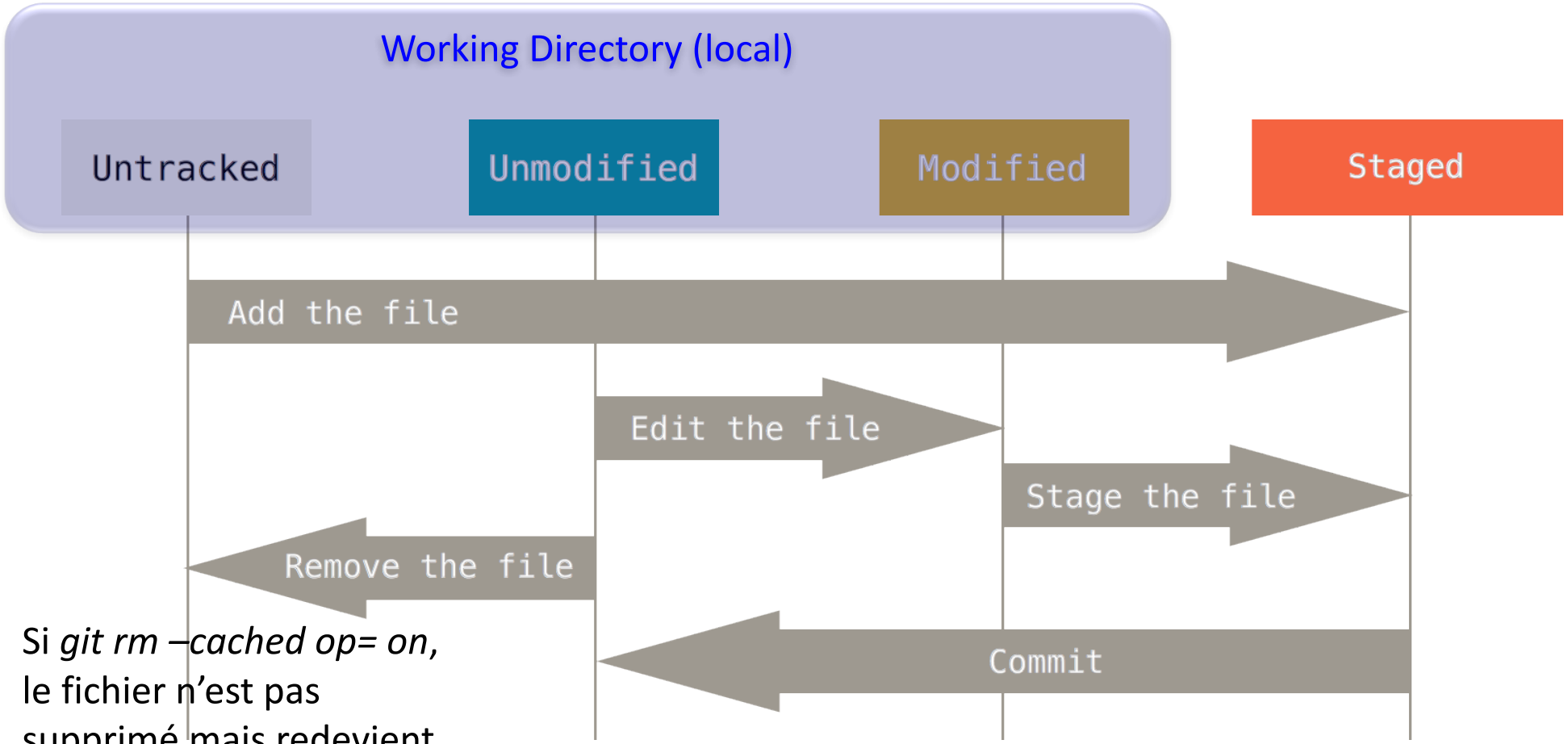
- Pour repartir d'un projet **existant**
  - git clone git://www.unexample.net/unProjet
  - cd unProjet

Configuration de Git .gitconfig

- Se trouve dans son répertoire personnel (`~/.gitconfig` sous Linux)
- On peut définir des alias dans la section [alias] de .gitconfig :
  - `ci = commit`
  - `co = checkout`
  - `st = status`
  - `br = branch`
- Ainsi au lieu de taper « git checkout », on pourra préférer « git co »


Working Directory (local)
Staging Area (index)
Git directory (local repository)
Add an untracked file
Stage a modifed file
Commit
Checkout
reset HEAD unFichier (désindexer un fichier)
git mv fileA fileAA
Renomme fileA en fileAA en conservant l'historique du fichier

États d'un fichier selon git



Si *git rm –cached op= on*,
le fichier n'est pas
supprimé mais redevient
Untracked

Convention messages Commit

- Obligatoire
  - Si vous ne mettez pas de message de commit, celui-ci sera annulé
- Règles pour les messages de commit :
  - Il est écrit au présent
  - Il tient en une phrase ; si nécessaire on saute une ligne puis suit une description complète longue.
- ex. :
  - « Améliore la visibilité des post-it sur le forum. » ;
  - « Simplifie l'interface de changement d'avatar. » ;
  - « Résout bug #324 qui empêchait valider tutoriel à plusieurs »
- Modifier son dernier message de commit (typo, erreur)
  - `git commit --amend`

Git log

- Pour consulter l'historique des commits locaux
  - Parcours avec les touches « *Page up* », « *Page down* » et les flèches directionnelles,
  - On quitte en appuyant sur la touche « Q »

- Version détaillée : `git log -p`
- Version synthétique : `git log --stat`

- git log

```
wu-pers-01164:cours-CVDA admin$ git log
commit 15593d8394bf9a67ea2c8ecb4e3e8cccc2be5167
Author: vde <veronique.deslandres@univ-lyon1.fr>
Date:   Tue Mar 14 15:16:58 2017 +0100

    premier commit
```

- git log -p

```
commit 15593d8394bf9a67ea2c8ecb4e3e8cccc2be5167
Author: vde <veronique.deslandres@univ-lyon1.fr>
Date:   Tue Mar 14 15:16:58 2017 +0100

    premier commit

diff --git a/1-CVDA-introModule.pptx b/1-CVDA-introModule.pptx
new file mode 100644
index 0000000..71b9da3
Binary files /dev/null and b/1-CVDA-introModule.pptx differ
diff --git a/2-CVDA-presentationNB_2016.pptx b/2-CVDA-presentationNB_2016.pptx
new file mode 100644
```

- git log --stat

```
    premier commit

 1-CVDA-introModule.pptx            | Bin 0 -> 349247 bytes
 2-CVDA-presentationNB_2016.pptx    | Bin 0 -> 2870734 bytes
```

```
    Ajout demo git log

 3-cours_GIT_CVDA_VDe.pptx | Bin 2134062 -> 2361591 bytes
 1 file changed, 0 insertions(+), 0 deletions(-)
```

Annuler un commit

- Annulation soft
  - git reset HEAD^ : revient à l'avant-dernier commit
  - Le fichier reste modifié
- Annulation hard
  - git reset --hard HEAD^

Annule le dernier commit et toutes les modifications des fichiers transmises par ce commit

- Pour annuler les modifications faites sur un fichier depuis un dernier commit :
  - git checkout monFich

monFich redevient tel qu'il était au dernier commit

Cfhttps://git-scm.com/book/fr/v2/Les-bases-de-Git-Annuler-des-actions

Supprimer un fichier du dépôt

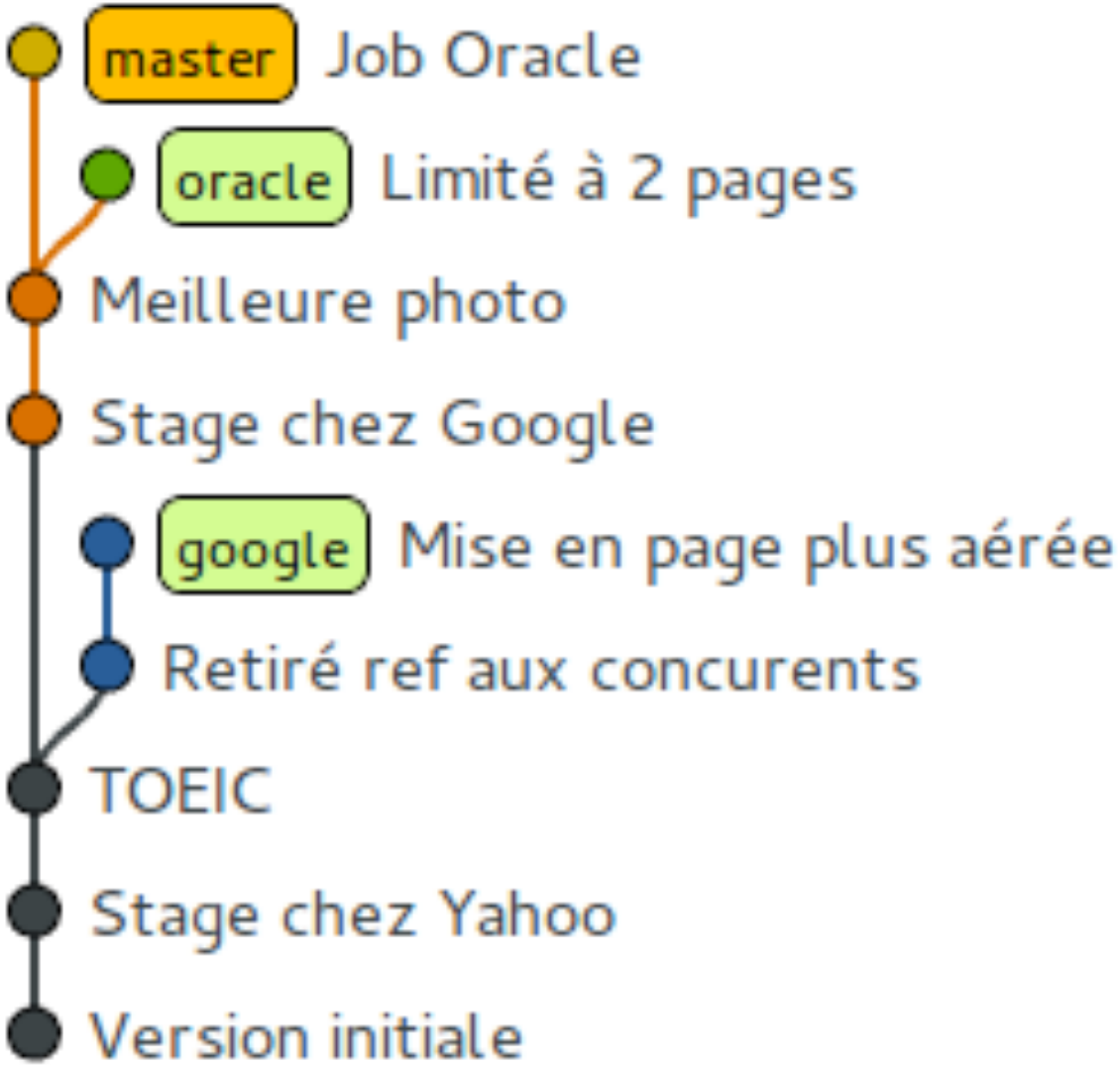
- Pour supprimer un fichier du dépôt, le processus peut sembler déroutant !
- Il faut :
  - le supprimer normalement
  - puis ajouter son entrée à l'index (avec la commande *git add*, bien que nous voulions supprimer un fichier)

Les **branches**

La gestion de branches : un point fort de git

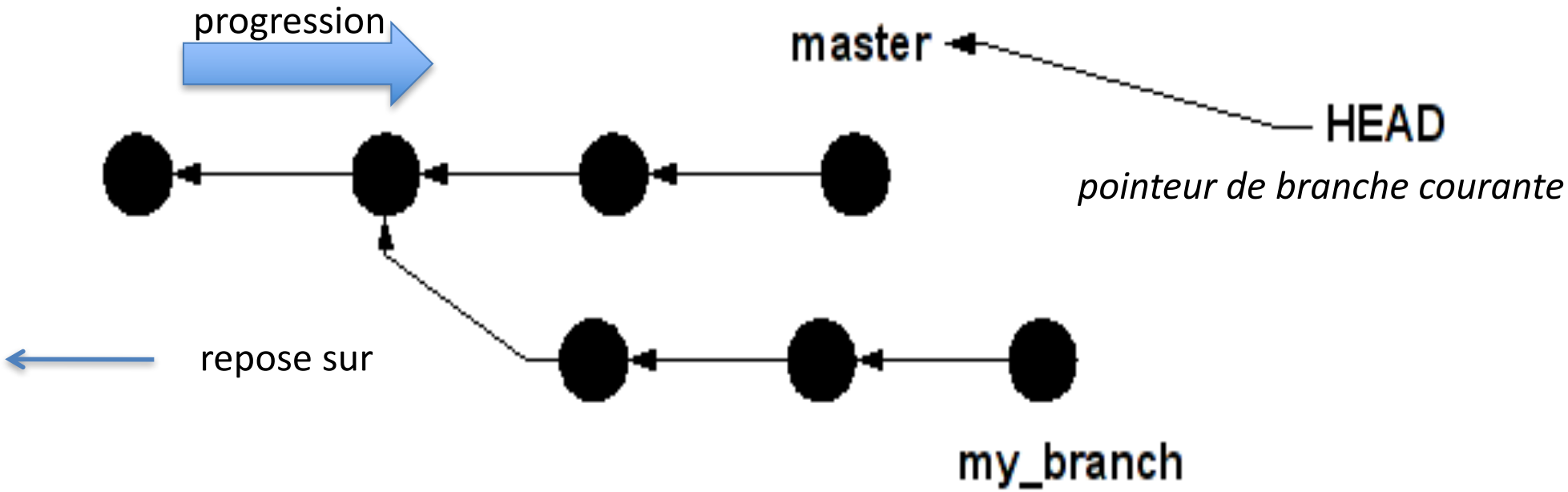
Illustration de n versions d'un CV, qui est mis à jour sur la branche master

Ici les branches ont vocation à rester



Travail collaboratif

- Le travail collaboratif de plusieurs personnes sur un même projet implique l'utilisation des branches.



Itération classique de travail

- **Modification** du code source du projet
- **Ajout** des fichiers modifiés à **l'index**
  - git add
- L'utilisateur **consulte l'état** de l'index
  - git status
- Si les changements contenus dans l'index lui vont, alors il les **commite**
  - git commit
- La **branche courante** est alors mise à jour, ainsi que le pointeur HEAD

Autre itération possible

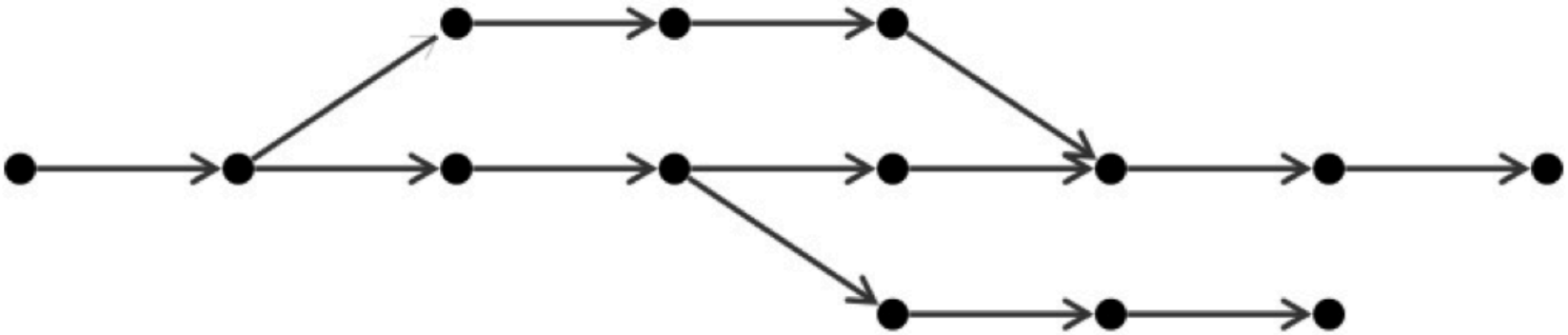
- Travailler sur une version **publiée**
  - Par ex. pour un site web
- Avoir une version de **travail**
  - dans laquelle on apporte des modifications incrémentales.
- Les deux versions mènent leur existence **en parallèle**
- La version publiée est régulièrement **mise à jour** par rapport à la version de travail

Exemple en cas de pb

- Créer une branche B1 pour un nouvel article sur lequel vous souhaitez travailler ;
- Réaliser quelques tâches sur cette branche
- Un problème critique a été découvert !

- Revenir à la branche de production P
- Créer une branche B2 et y développer le correctif ;
- Après un test, fusionner la branche B2 et pousser le résultat à la production
- Rebasculer sur la branche B1 et continuer le travail

Fusion de branches



Il existe deux façons de réintégrer une branche dans la principale (master) :

– Le **merge**, qui crée un historique **parallèle**

(le commit de fusion possède 2 parents)

– Le **rebase**, qui crée un historique **en série**

(git rejoue l'histoire pour 'aligner' les modifications)

Fusion de branches

AVEC MERGE

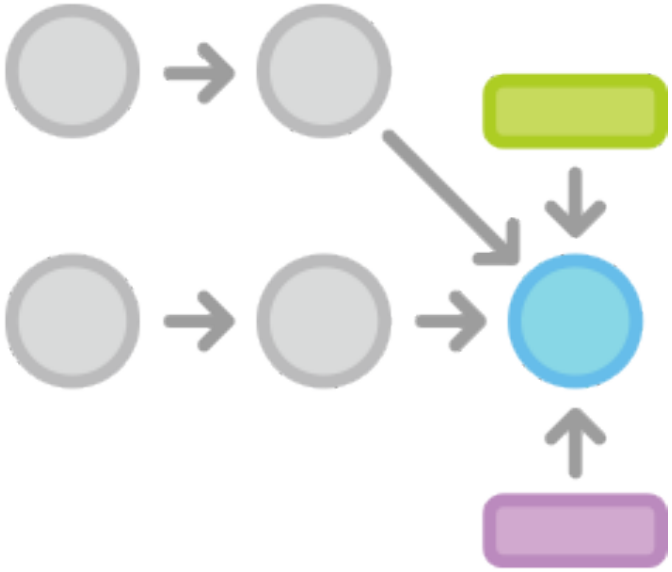
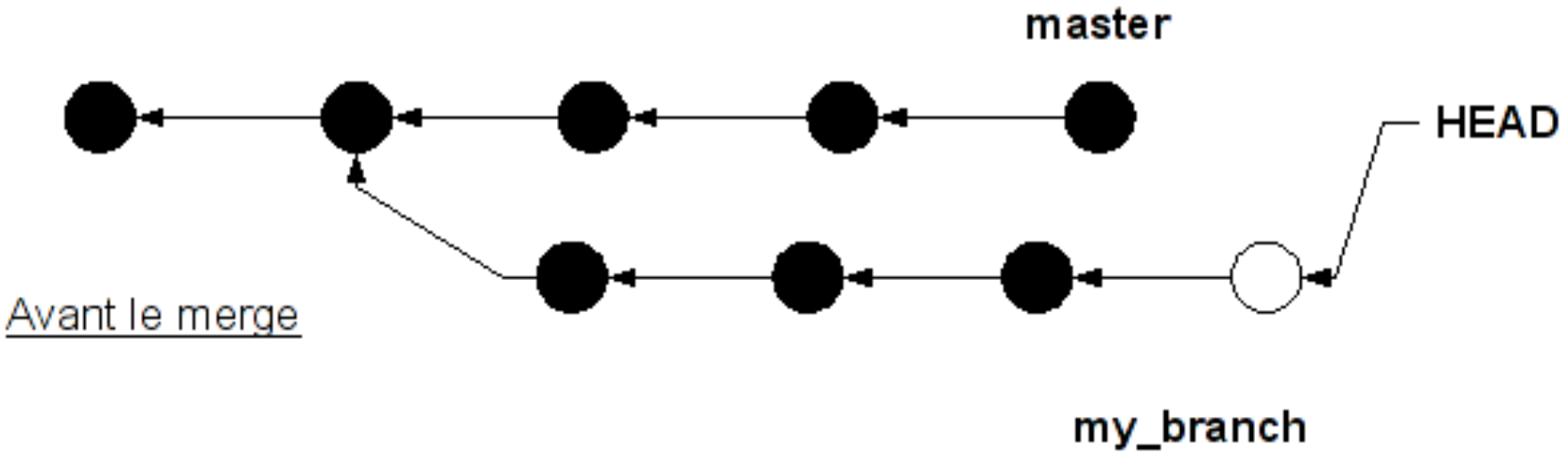


Illustration du **merge**

- On crée une nouvelle branche, par exemple :
  - git checkout **-b my_branch** master
  - On effectue différents *commits* sur **my_branch**
- Supposons que des changements ont été également effectués sur la branche master



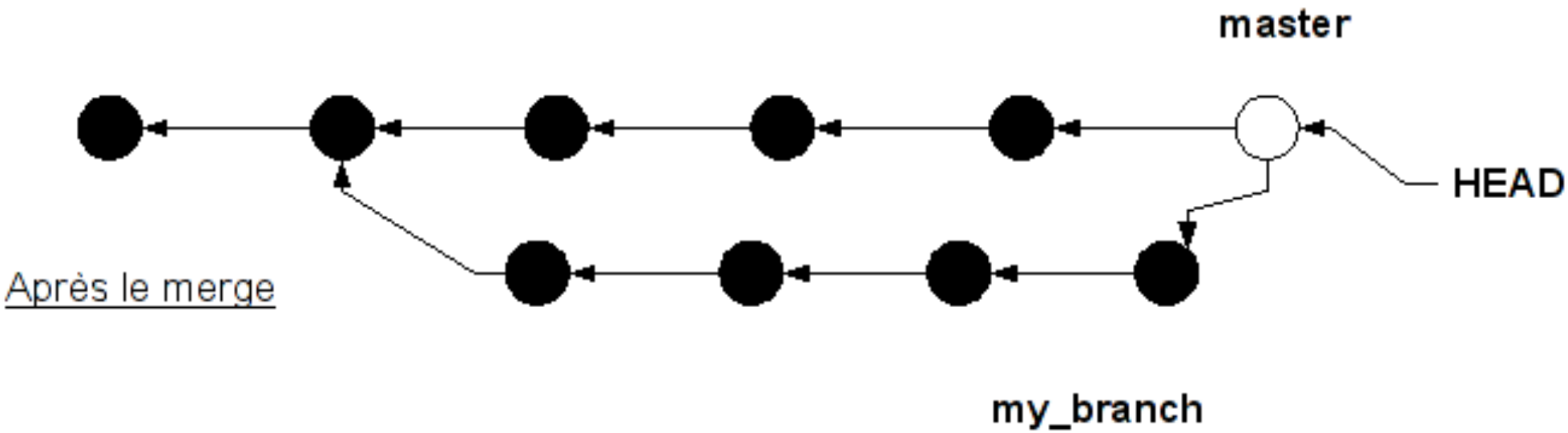
Merge / fusion

- La branche nous paraît stable, nous décidons alors de la fusionner avec la branche principale :
  - **git checkout master** pour repasser dans la branche master
  - **git merge my_branch** fusionne la branche my_branch avec la branche courante
  - **git branch -d my_branch** supprime la branche my_branch devenue inutile

« On se déplace sur la branche pour accueillir l'autre »

Après la fusion

Cas simple, sans conflit, appelé "fast-forward"



Fusion de branches

REBASE

ON RÉÉCRIT L'HISTOIRE

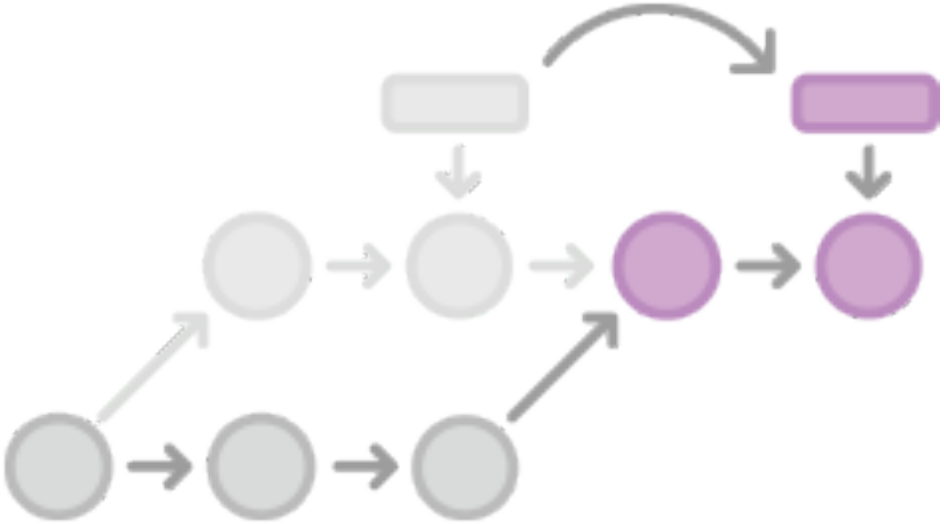
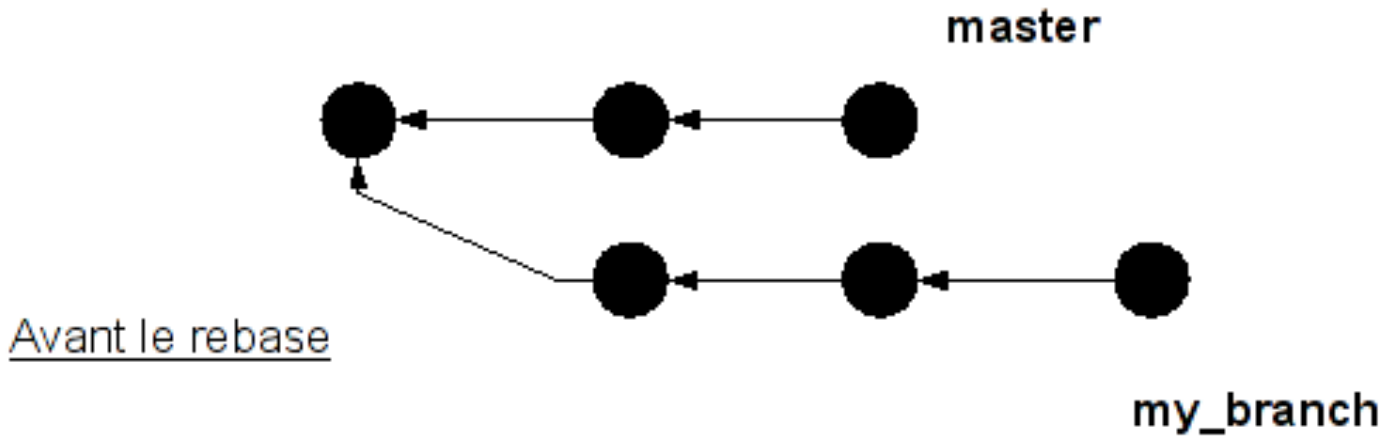


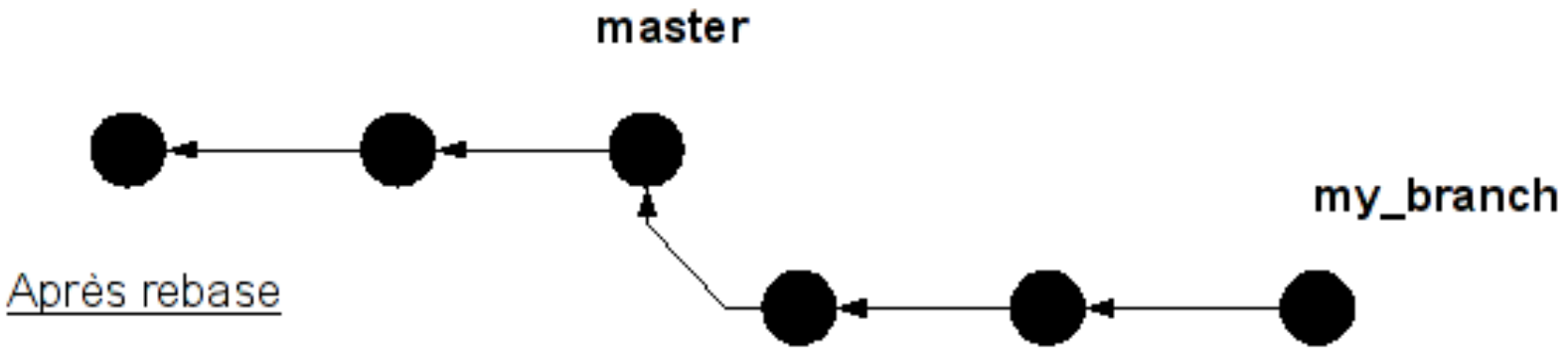
Illustration du **rebase**

- C’est un **processus**
- Consiste à ré-écrire l'historique de la branche, de manière à ce qu'il s'adapte à la dernière version de la branche avec laquelle on veut fusionner
- Une fois l'historique ré-écrit, on peut **fusionner** la branche avec *merge*, aucun conflit ne pouvant alors plus apparaître.



Rebase

- On lance l'opération de rebase
  - **git rebase master my_branch**
- Des conflits peuvent apparaître, il faut alors les corriger, ajouter les changements à l'index via **git add**, puis relancer le rebase via **git rebase --continue**



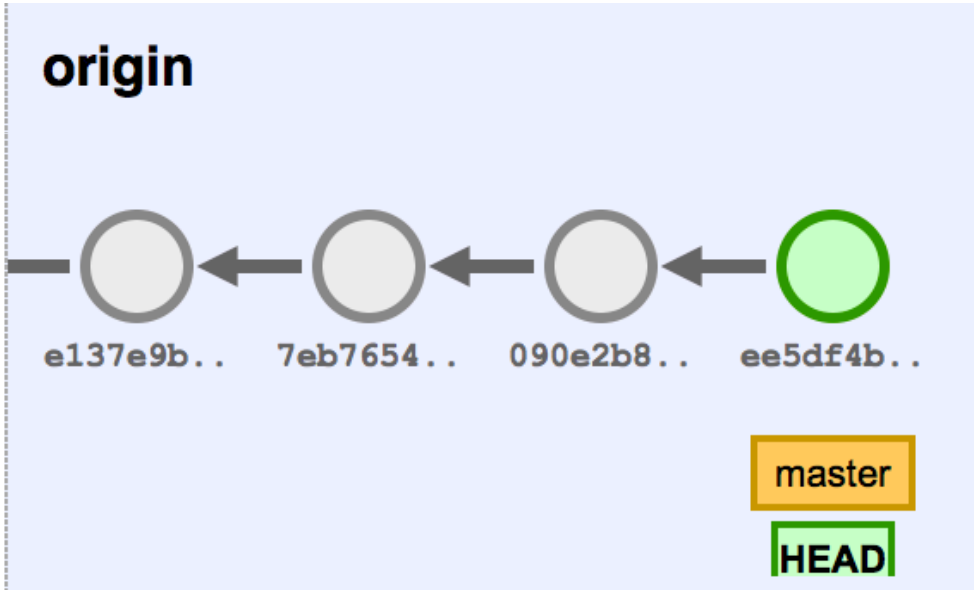
Suite **Rebase**

- On peut alors fusionner la branche my_branch dans master :
  - **git checkout master** : on passe à la branche master
  - **git merge my_branch**, fusionne la branche
  - aucun conflit possible (traité avant)

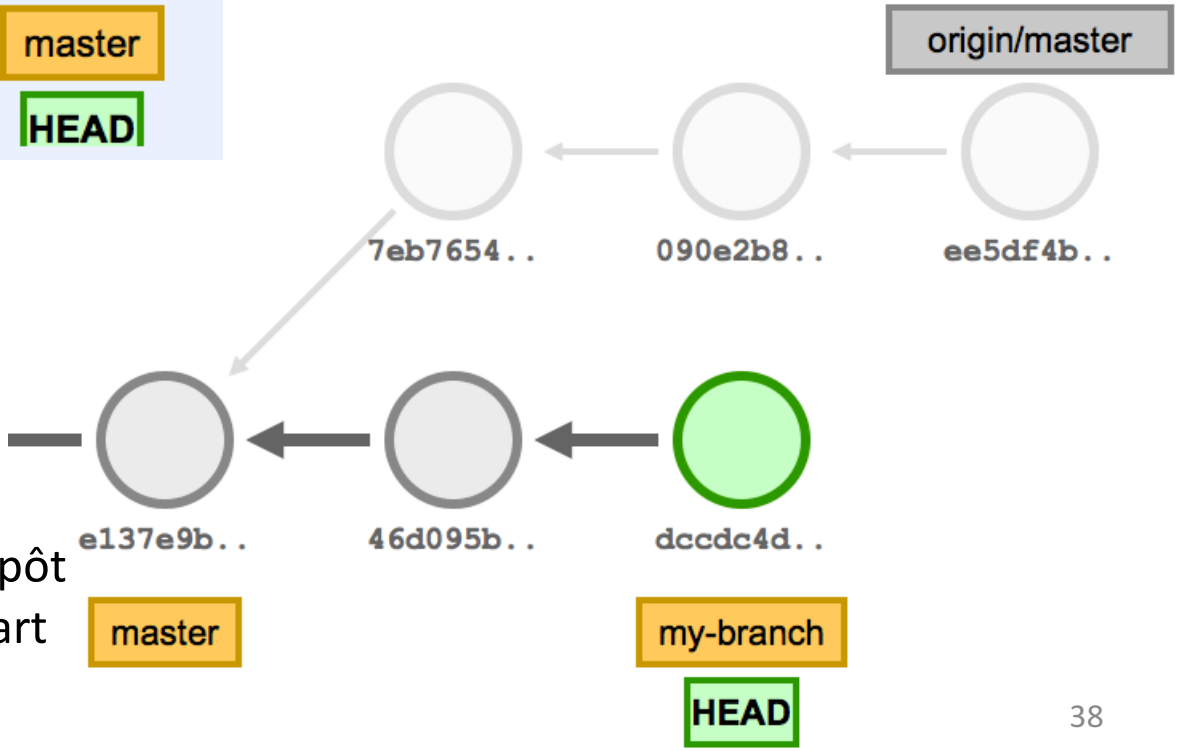
Après le merge

Ex. de rebase

Dépôt distant :



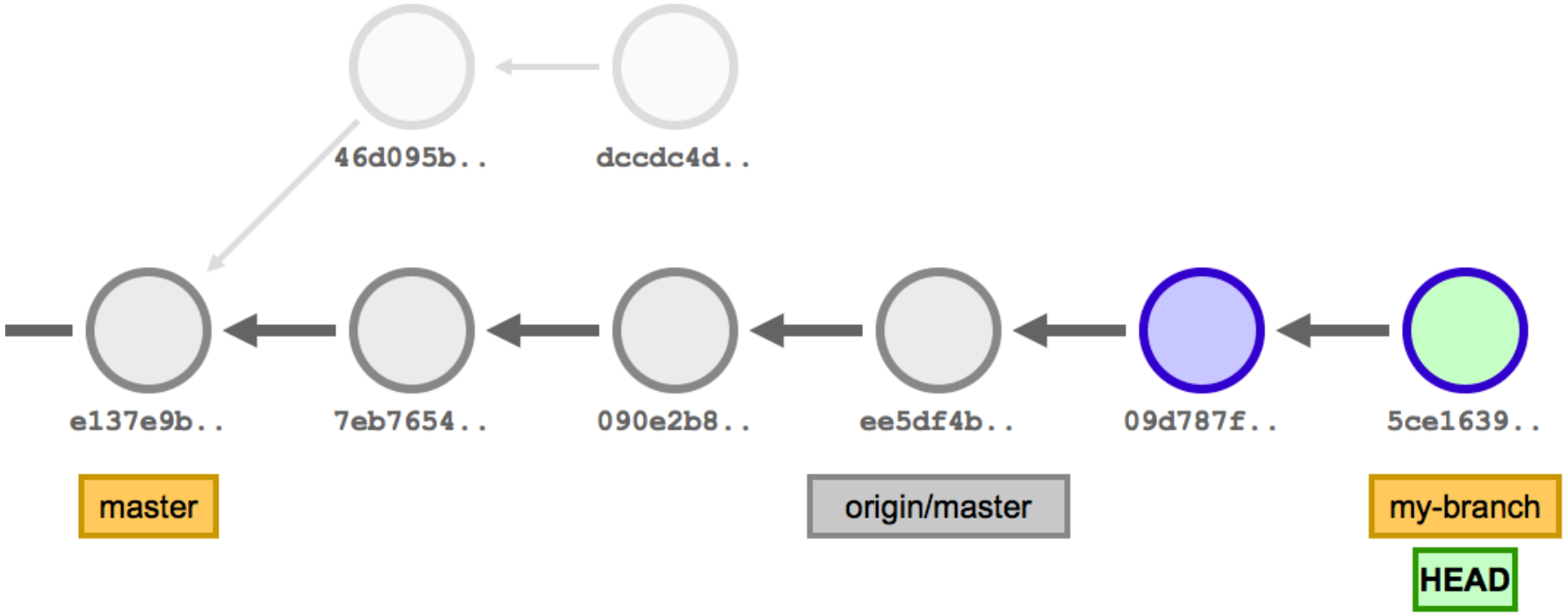
git fetch :



git fetch importe les commits d'un dépôt distant en local sous une branche à part

Ex. Rebase (suite)

- git rebase origin/master



https://onlywei.github.io/explain-git-with-d3/#fetchrebase

Merge vs. Rebase

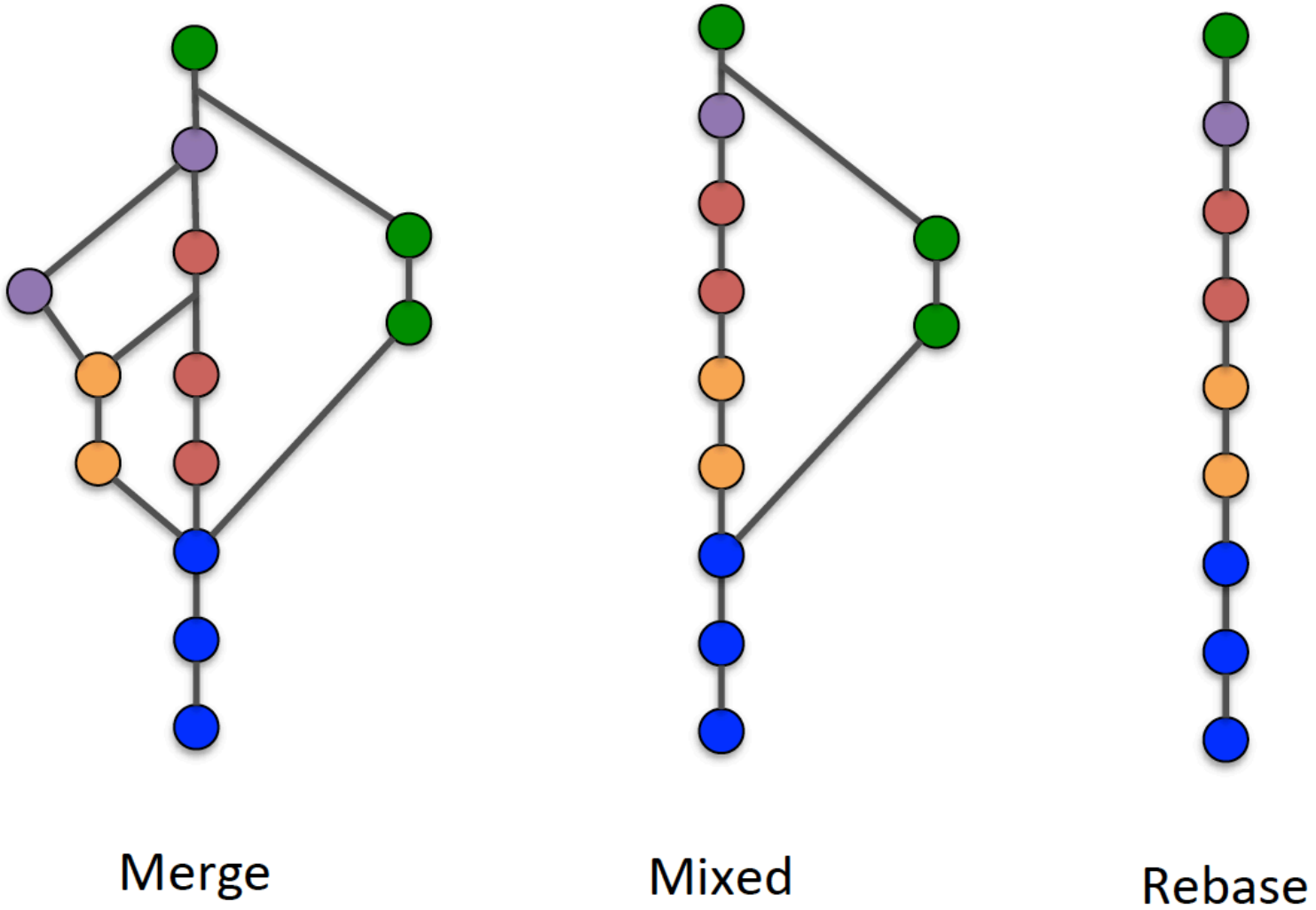


Image : David Parson, INRIA

Envoyer au dépôt distant

- La bonne démarche est la suivante :
  - Télécharger les nouveautés : git pull
  - Attention quand on récupère le dépôt distant, seuls les changements de la *branche courante* sont récupérés

    ➔Bien se placer sur la branche **master**

    - soit vous n'avez effectué aucune modification depuis le dernier pull, dans ce cas la mise à jour est simple (*fast-forward*) ;
    - soit vous avez fait des commits en même temps que d'autres personnes. Les changements qu'ils ont effectués sont alors fusionnés intelligemment aux vôtres automatiquement

Envoyer au dépôt distant (2)

- Vérifiez que tout les commits que vous allez envoyer sont conformes et qu'il n'y a pas d'erreur qui saute aux yeux : git log –p
- Refaire un git pull pour vérifier que personne n'a envoyé de travail depuis (en effet le serveur ne peut régler les conflits à notre place s'il y en a). **Personne ne doit avoir fait de push depuis notre dernier pull.**
- Envoyer vos commits : git push

n dépots distants

- Le propre de git est de pouvoir utiliser *n* serveurs distants
- Dans ce cas, on peut définir le principal (upstream)
  - git push --set-upstream origin master

- Quelques commandes :
  - **git remote** : affiche la liste des dépôts enregistrés (simplement les noms)
  - **git remote -v** : affiche la liste des dépôts enregistrés (nom et adresse)
  - **git remote add <nom du dépôt> <adresse du dépôt>** : ajoute un nouveau dépôt
  - **git remote remove <nom du dépôt>** : supprime un dépôt

Bonnes Pratiques Générales

- Commiter souvent (en local)
  - Il est possible d'annuler ou modifier un commit
  - Même pour de petite modification. L'espace utilisé sur votre disque est très minime.
  - Une grande modification (touchant beaucoup de lignes/fichiers) est moins lisible si quelqu'un doit revenir dessus.
- Un seul Push par jour
  - Un push est irréversible. Une fois que vos commits sont publiés, il deviendra impossible de les supprimer ou de modifier le message de commit !

BP suite

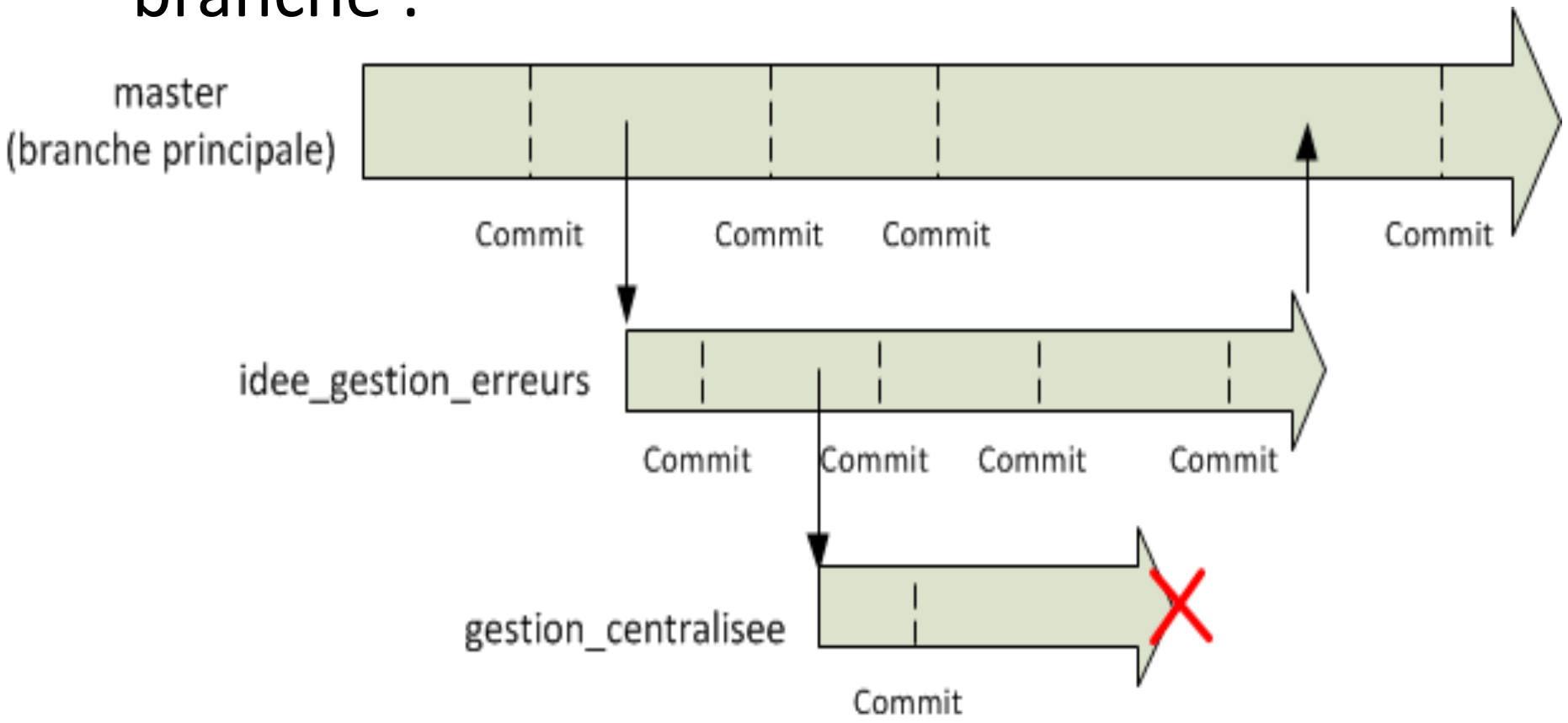
- Il est préférable d'avoir des commits reflétant l'ajout de fonctionnalités/correctifs
  - et non des commits partiels effectués par simple sécurité
- Passer par l'étape de staging (mise en index) avant de commiter
  - *git add *.php* par ex.
  - *git rm –cached <fichier/dossier>* enlève de l'index sans supprimer les fichiers
  - Utiliser .gitignore

Quand créer une branche ?

- Quand on veut modifier le code, se poser les questions suivantes :
  - « Ma modification sera-t-elle rapide ? » ;
  - « Ma modification est-elle simple ? » ;
  - « Ma modification nécessite-t-elle un seul commit ? » ;
  - « Est-ce que je vois précisément comment faire ma modification d'un seul coup ? ».
- Si la réponse à l'une de ces questions est « non », on doit probablement créer une branche
- Une branche peut être considérée comme un **dépôt intermédiaire**

Sous-branches

- On peut créer une sous-branche d'une branche :



Il y a plein d'utilisations possibles des branches (hiérarchie, thématique, personnelles…), ça dépend du projet et des développeurs

Supprimer / Lister les branches

- git branch **-D** gestion_centralisee
  - attention ! Cela supprime la branche *gestion_centralisee*, et tous les changements qui s'y rapportent

- Lister les branches locales :
  - git branch
- Lister toutes les branches distantes que le serveur connaît :
  - git branch –r (remote)

Récupérer une branche d'un dépôt distant

- Si le dépôt distant possède une autre branche, par exemple « origin/optionEnglish », et que vous souhaitez travailler dessus, il faut créer une copie de cette branche en local qui va « suivre » (*tracker*) les changements du serveur :
  - git branch --track optionEnglish origin/optionEnglish
  - git pull : **sur la branche** optionEnglish pour la mettre à jour avec le serveur

L’inverse : créer une branche sur le serveur

- Il est possible d’ajouter des branches sur le dépôt distant, pour y travailler à plusieurs :
  - git push origin origin:refs/heads/nom_nouvelle_branche

- Ensuite créer une branche locale qui « suit » la branche du serveur comme vu précédemment :
  - git branch –track new-branche origin/nom_nouvelle_branche

Astuce : rechercher dans le code source de son projet

- Pour connaître les noms des fichiers qui contiennent le mot TODO dans le code source, il suffit d'écrire :

  - git grep "TODO"
  - (utilise aussi les expressions régulières)

Taguer une version

- On peut donner un **alias** à un commit précis pour le référencer.
  - C'est utile par exemple pour dire « À tel commit correspond la version 1.6 de mon projet ». Cela permettra à d'autres personnes de repérer la version 1.6 plus facilement. C'est le rôle des tags.
- Pour ajouter un tag sur un commit :
  - git tag NOM-du-TAG IDCOMMIT
- Attention, un tag n'est pas envoyé lors d'un push, il faut préciser l'option *–tags* :
  - git push --tags

GitHub

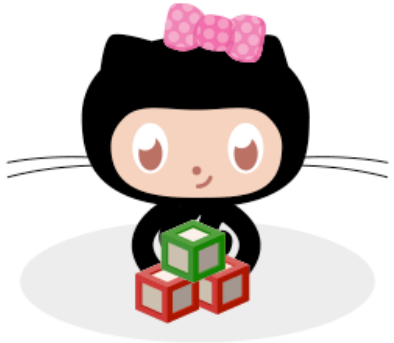
- Github peut servir de **dépôt distant** pour votre versionning
  - il est tout à fait possible d'utiliser git **sans** GitHub
  - Si on est seul, on peut utiliser une clef USB ou un serveur lambda comme serveur distant :
    - git remote add origin E:/versioning/monSuperProjet
  - Si on travaille à plusieurs, on peut aussi utiliser **bitbucket**, **gitlab** ou DropBox ou GoogleDrive, mais uniquement pour les infos de versionning, cf
    - https://www.grafikart.fr/blog/git-not-github

Procédure

1. **Créer** un compte github
2. Configurer github pour être **synchronisé** avec votre GIT
   - Les pull/push effectués sur votre poste se feront directement avec votre repository github
   - (Suivre une procédure créant une clef SSH, cf détails ci-après)
3. Sous github, récupérer le projet du tuto (**fork**)

   → depuis le compte gitHub fourni vers le vôtre
4. Sous git, **cloner** le projet

   → Copie du serveur distant dans votre répertoire local
5. Vous pouvez maintenant travailler en local avec git



Synchroniser GitHub et Git

- On peut soit se connecter avec HTTPS
  - If you clone with HTTPS, you can cache your GitHub password in Git using a credential helper.
- Soit se connecter via SSH (à préférer à l'IUT)
  - If you clone with SSH, you must generate SSH keys on each computer you use to push or pull from GitHub
  - Attention la **passPhrase** sera demandée à chaque synchronisation avec le serveur distant !
- Suivre la procédure :
  - https://help.github.com/articles/set-up-git/
  - Attention à bien configurer le proxy pour les contrôles du campus Lyon1 !
    - http://liris.cnrs.fr/~pchampin/enseignement/intro-git/#collaborer-avec-git

GitHub

- Sur GitHub, la plupart des dépôts public sont en mode lecture seule (*read only*)
  - On peut télécharger les fichiers, effectuer des modifications sur votre ordinateur (en local) mais on ne peut pas les envoyer sur le serveur sans demander un « pull request » au propriétaire
  - Il téléchargera nos modifications lui-même pour vérifier si elles sont correctes.
- Github 'fork' = récupérer un projet existant sous son compte Github
  - (C'est en fait un git clone + track de son ID pour un éventuel *pull request* ultérieur)
  - Pour l'avoir en local : sur votre PC, taper
    - git clone <URL_projet_sous_github> <répertoire-destination>

- **La plateforme des développeurs modernes**, elle offre la possibilité de gérer ses dépôts Git
- Permet la gestion de tout le processus de développement « *From idea to production* »
- Permet une collaboration simple sur un même projet
- Est Open source et collaboratif
- Gratuit (pour la version de base déjà très complète)
- Aussi une solution pour les entreprises
  - Ex: pour gérer les retours clients directement sur mon repository via les *issues*

Fonctionnement GitLab

- Après avoir pushé votre projet sur Gitlab, vous permettez à d'autres utilisateurs d'accéder à ce projet.
- Dès qu'un des contributeurs du projet a une **idée**, il crée une **issue**. Cette issue apparaît ensuite sur le **tableau de bord** et la personne assignée à cette tâche est **notifiée**. Il ne lui reste plus qu'à **coder** pour réaliser l'idée initiale.
- Le contributeur doit ensuite **commiter** sa partie.

- Il faut ensuite **tester et valider** les changements effectués. Une fois que tout est bon, on envoie en production.
- Les clients ou autres collaborateurs peuvent donner des **retours** (feedback).
- Les utilisateurs finaux / clients peuvent aussi avoir un accès GitLab, pour y créer des '**issues**' relatives à leur retours, et les intégrer dans le processus de développement directement.

Serveur GitLab à l'IUT

- http://iutdoua-git.univ-lyon1.fr
- login et mot de passe : ceux de l'université

- Avantages :
  - public : solo aux personnes de l'IUT
  - Privé : toujours possible
  - Interne (propre à ceux qui ont un login)
- Inconvénients :
  - Dépôts perdus une fois parti de l'IUT

Pour ajouter un membre

- Se positionner sur le projet
- Choisir Members
- Chercher la personne à ajouter
  - Par son nom ou identifiant
- Choisir son type parmi :
  - Guess
  - Developper
  - Master

Installation logicielle, configuration

TP GIT

Fonctionnement

- Ce TP-tutoriel utilise GIT en langage de commandes, et nécessite un **binôme** de développeurs, **chacun ayant son poste**.
  - Les 2 développeurs seront connectés au même dépôt gitHub
  - Nous allons utiliser une fenêtre Terminal et travailler en commandes Unix et GIT
  - (Sous Windows, il existe le terminal GIT Bash qui est un shell propre à GIT)

Au boulot !

Pour aller plus loin

- https://openclassrooms.com/courses/gerez-vos-codes-source-avec-git
- Visualiser les commandes de Git : https://onlywei.github.io/explain-git-with-d3/#checkout
- Fusion de branches : https://git-scm.com/book/fr/v1/Les-branches-avec-Git-Brancher-et-fusionner%C2%A0%3A-les-bases
- Utiliser GitHub pour Git sous Windows
  - https://www.grafikart.fr/tutoriels/divers/git-github-131
  - https://aurnyweb.com/synchronisation-github-shell/ (clef SSH)
- Cours d'intro GIT de l'IUT :http://liris.cnrs.fr/~pchampin/enseignement/intro-git/