DevOps

GIT

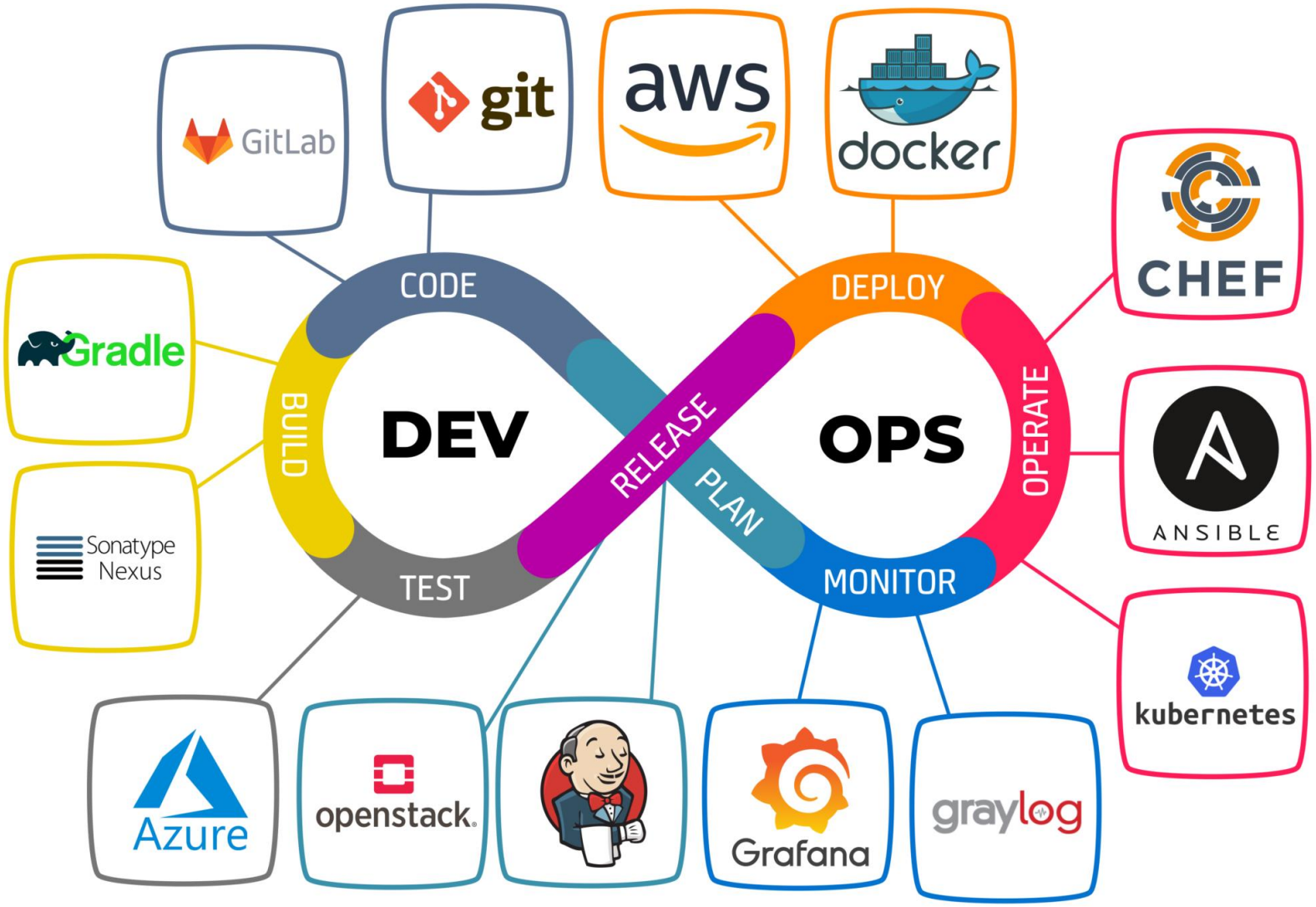
Jamel ESSOUSSI: Architecte logiciel

jamel.essoussi@gmail.com

https://github.com/jessoussi

2025

Outils du DevOps



Plan du cours

I. Introduction

II. GIT

III. Utilisation de GIT

IV. Les bonnes pratiques

V. Synchronisation avec les dépôts distants

Motivation

Une équipe de développeurs participe à la réalisation d'une application:

- Comment conserver un historique?
- Comment revenir en arrière?
- Comment travailler à plusieurs en parallèle sur le même code?
- Comment gérer plusieurs versions du code à la fois?
- Comment savoir ce qui a été modifié et par qui (et pourquoi)?

Motivation

Utilisation d'un VCS (Version Control Software)

- Un système permettant le suivi (dans le temps) des différentes modifications apportées à un ensemble de fichiers observés.

VCS: Version Control Software

SCM: Source Code Management

RCS: Revision Control System

Qui? Quand? Pourquoi? Où et Comment?

Maîtriser le code produit durant un développement logiciel implique de savoir ce qui a été fait :

- Par l'ensemble des développeurs (Qui ?)
- Dans le temps (Quand ?)
- Pour quel motif/fonctionnalité (Pourquoi ?)
- Impliquant de nombreuses fonctions, dans de nombreux fichiers (Où ? Comment ?)

  → Leur utilité ne se limite pas au travail à plusieurs.

Utilité

- Permettre la **traçabilité** d'un développement (historique des changements)
- Faciliter la **collaboration**, éviter les conflits ou aider à la leur résolution
- Garder une version du **code source toujours fonctionnelle**, tout en travaillant sur plusieurs fonctionnalités (notion de branche)
- Permettre des **schémas organisationnels structurant** les développements (workflows)

Historique

- Principaux SCM
  - cp -old (vieille blague, mais toujours utilisé)
  - CVS (1990) : centralisé, travaille sur des fichiers, limité
  - SVN (2000) : centralisé, travaille sur des fichiers, workflows limités
  - Git (2005) : décentralisé, travaille sur des arborescences de contenus
  - … et beaucoup d'autres, libres (Mercurial, Bazaar) ou propriétaires (Microsoft, HP, IBM)

→ Quelque soit le SCM utilisé, il est important d'en utiliser un et de maîtriser les opérations de base.

Ce qu'on y stocke

- Essentiellement des fichiers texte.
- Ce qu'on y met:
  - Fichier sources (.java, .c, .html, etc)
  - Certains fichiers binaires non dérivés des sources (images)
  - Fichiers de configuration, compilation (pom.xml)

- Ce qu'on n'y met pas
  - Fichiers temporaires (.class, …)
  - Fichiers générés

diff & patch

Un VCS repose sur un mécanisme permettant de calculer les différences entre 2 versions d'un fichier.

- diff
  - Comparaison de fichiers ligne par ligne
  - Indique les lignes ajoutées ou supprimées
  - Peut ignorer les casses, les tabulations, les espaces

- patch
  - Utilise la différence entre deux fichiers pour passer d'une version à l'autre

diff & patch

Illustration

- Sauvegarder dans un patch les modifications d'un fichier
  ```
  $ diff toto.java toto-orig.java > correction.patch
  ```
- Appliquer le patch à une autre version du fichier
  ```
  $ patch -p 0 mytoto.java < correction.patch
  ```

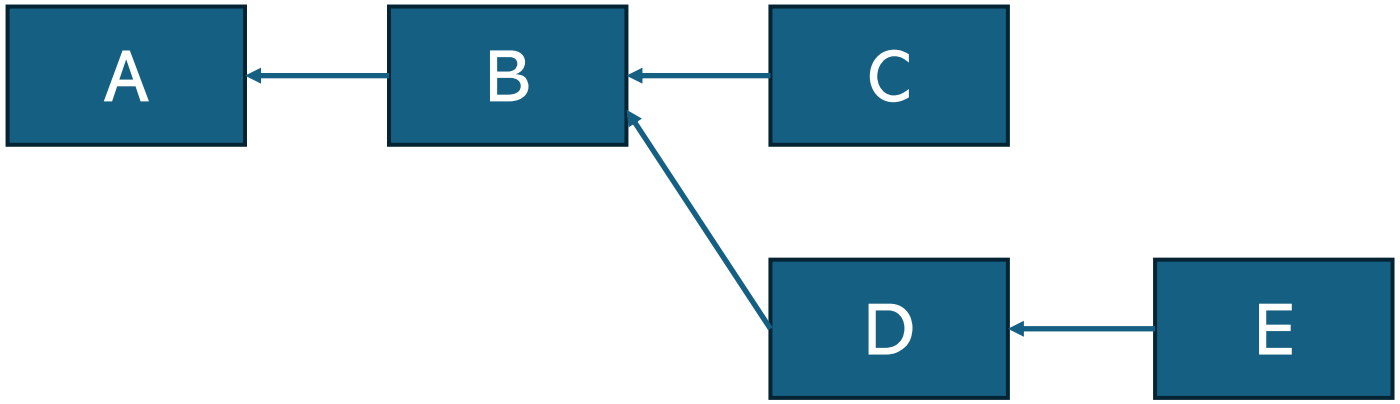
→ diff et patch peuvent être appliqués à une arborescence de fichiers

La notion d'historique

En plus de calculer la différence entre deux versions d'un fichier, il faut gérer un historique des diffs:

- **L'historique** est un graphe orienté composé d'un ensemble de versions pouvant être recalculées à partir des versions adjacentes en appliquant les patchs.
- L'historique peut inclure plusieurs **branches**, c'est-à-dire des sous-graphes qui évoluent en parallèle.

La notion d’historique



Plan du cours

I. Introduction

II. GIT

III. Utilisation de GIT

IV. Les bonnes pratiques

V. Synchronisation avec les dépôts distants

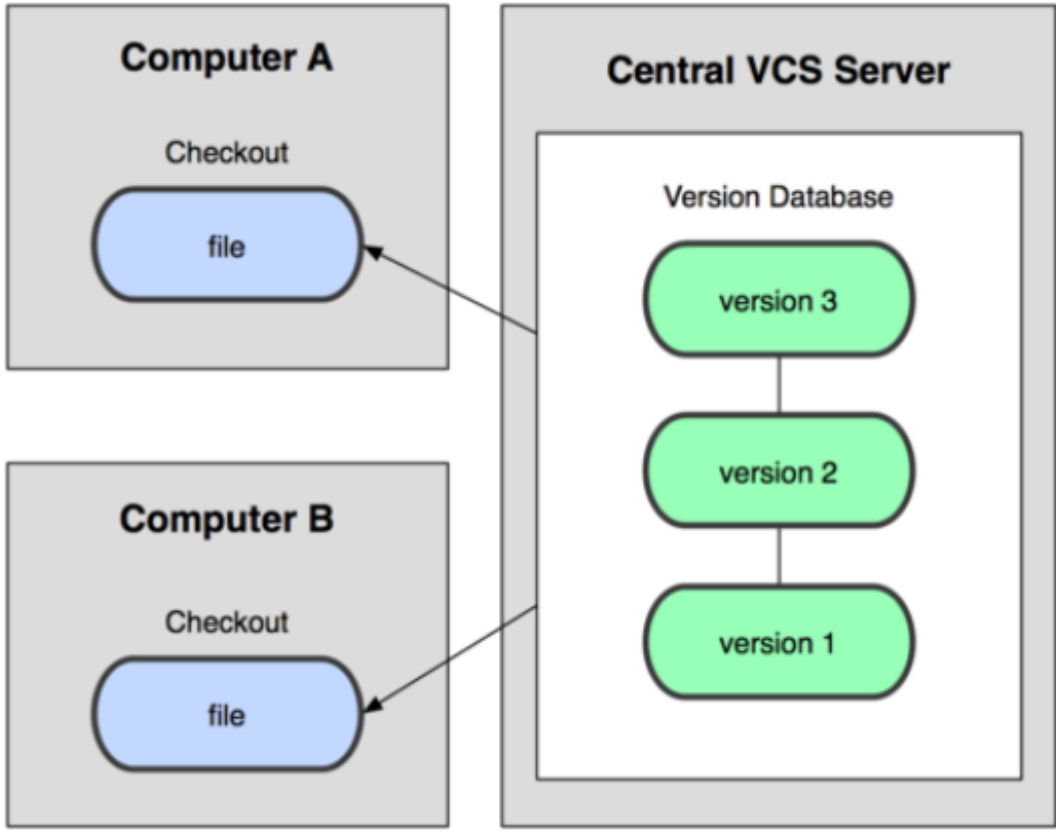
GIT

- Créé par Linus Torwarld pour gérer les sources du noyau Linux
- Très souple et puissant mais complexe et peu intuitif → apprentissage ardu
- SCM le plus utilisé actuellement, libre (GPL), communauté très active

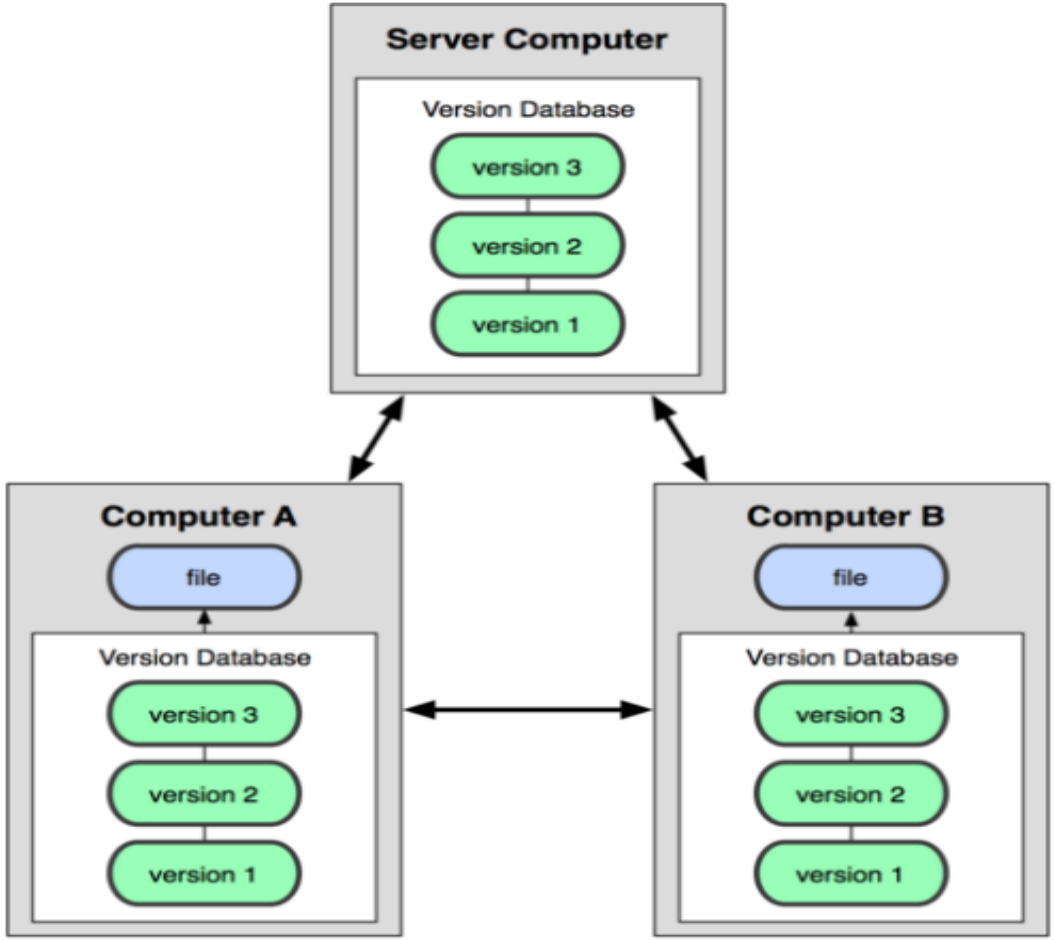
GIT

- Pas de serveur central (élément critique)
- Utilisable même si déconnecté
- Organisation du travail plus souple

Gestionnaire de version Centralisé



GIT- Gestionnaire de version décentralisé



GIT- Gestionnaire de version décentralisé

- Chaque client git exécute son propre dépôt en local
- Chaque utilisateur d'un dépôt partagé possède une copie de tout l'historique des changements enregistrés (full mirroring)

→ But : faciliter les développements parallèles, permettre au code de diverger/converger rapidement

Fondé sur une fonction de hachage

SHA-1

- Secure Hash Algorithm (cryptographie)
- Génère une empreinte des données d'entrée
  - Contenu du fichier
  - en-tête
- Propriétés:
  - Hash de 160 bits
  - Très faible probabilité de collision
- Identifie de manière unique chaque objet

Fondé sur une fonction de hachage

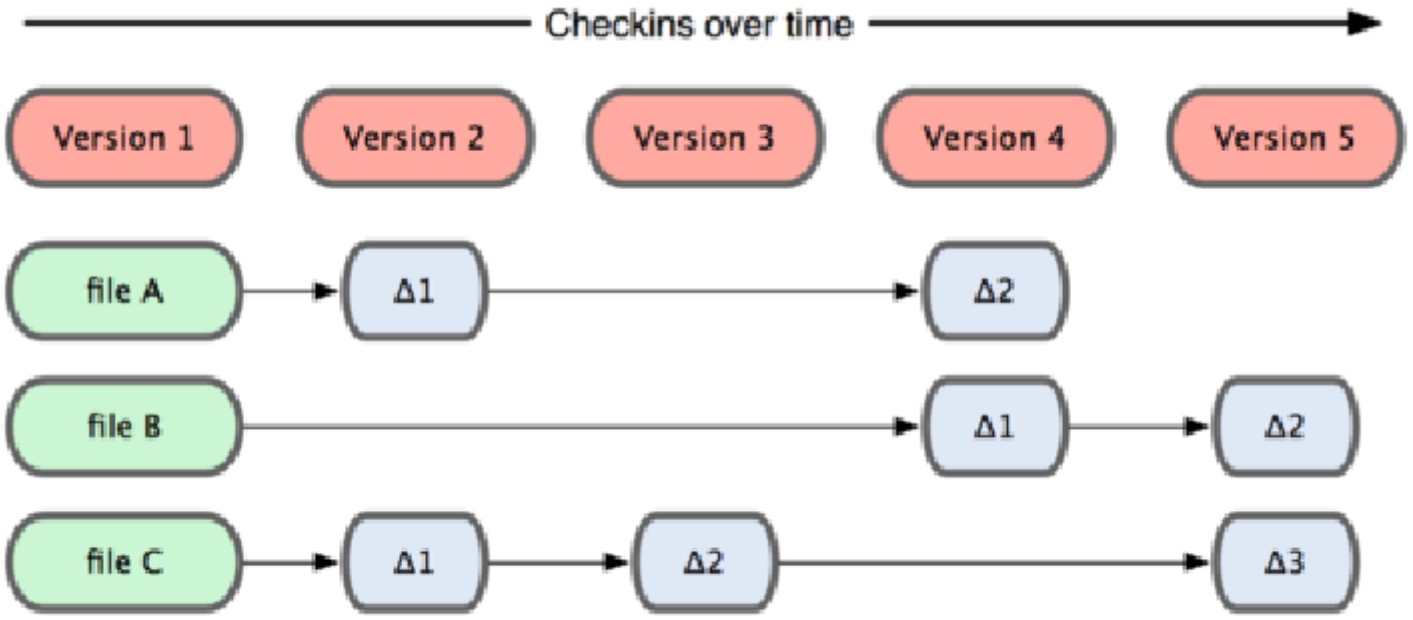
Exemple

```
$ echo a > toto
$ sha1sum toto
        3f786850e387550fdab836ed7e6dc881de23001b toto
$ echo b >> toto
        05dec960e24d918b8a73a1c53bcbbaac2ee5c2e0 toto
```

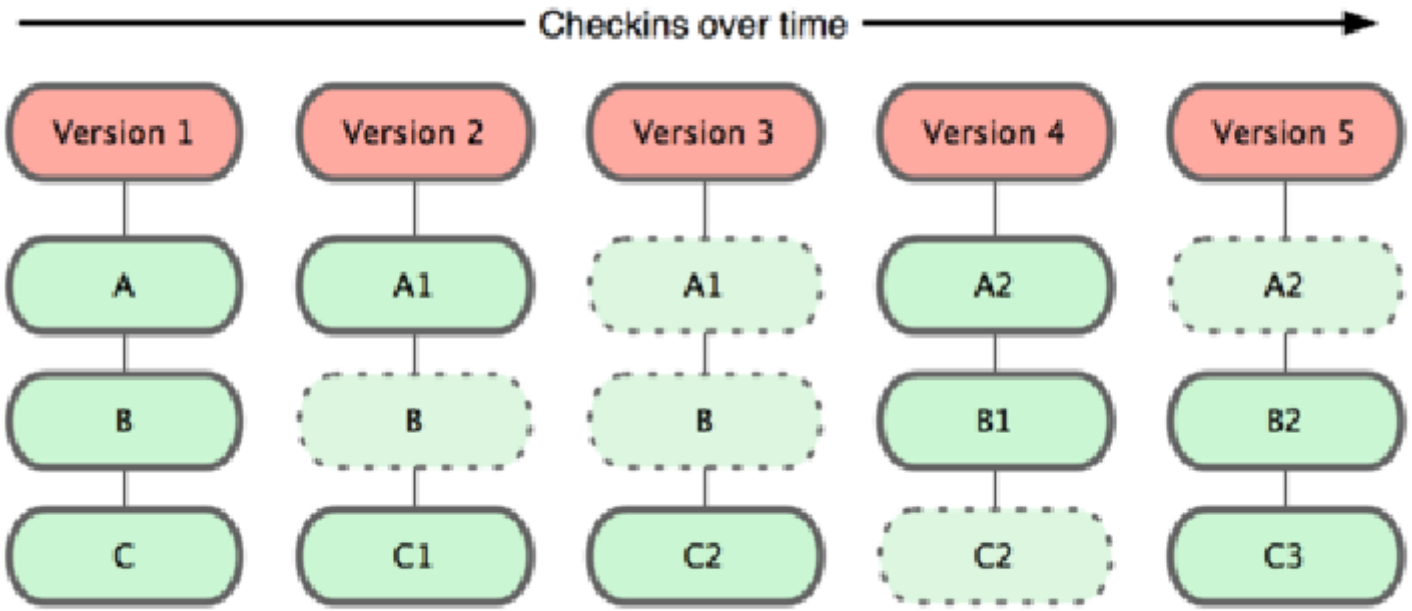
Stockage par Delta VS snapshot

- CVS, SVN ne stockent que les deltas des fichiers modifiés par un commit
- Git stocke tout le contenu du répertoire versionné à chaque commit (mais utilise une compression intelligente basée sur la version antérieure la plus proche)
- Permet une grande souplesse

Stockage par Delta: CVS - SVN



Stockage par snapshot: GIT



Les objets dans Git

- Blob
- Tree
- Commit
- Tag

Blob

On appelle Blob, l'élément de base qui permet de stocker le contenu d'un fichier.

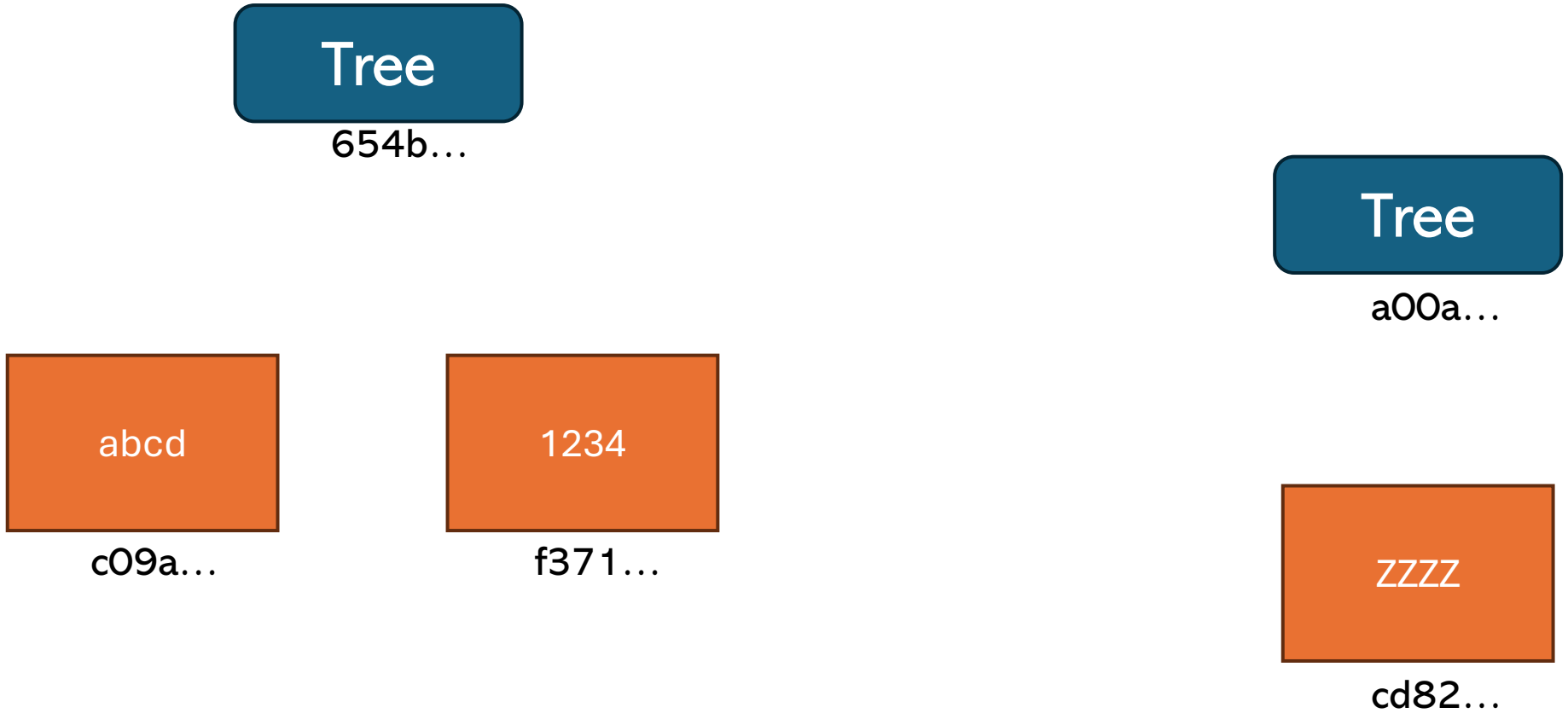
- Chaque Blob est identifié de manière unique par sa clé
- A chaque révision du fichier correspond un nouveau Blob
  - Le blob stocke le contenu entier du fichier (pas seulement un diff)
- Le Blob ne dépend pas du nom ou de l'emplacement :
  - Si un fichier est renommé ou déplacé pas de nouveau Blob
- Le contenu du Blob est compressé avec zlib. Il contient:
  - Le type d'objet (blob)
  - La taille du fichier initial
  - Le contenu du fichier

Tree

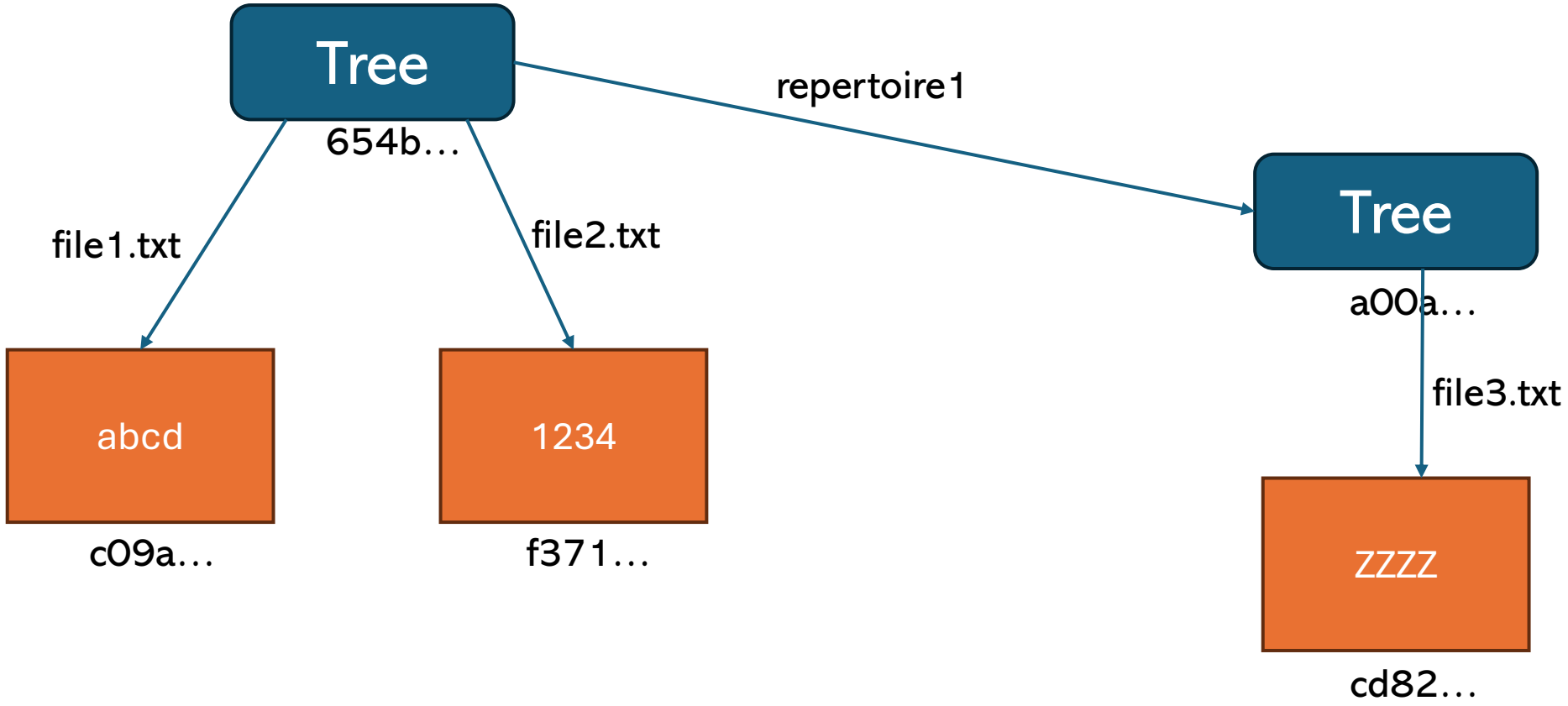
Un Tree stocke la liste des fichiers d'un répertoire.

- Un Tree est un ensemble de pointeurs vers des Blobs et d'autres Trees.
- Un Tree associe un nom de fichier à chacun des pointeurs de Blobs
- Un ensemble de Trees permet de décrire l'état d'une hiérarchie de répertoires à un moment donné.

Blob & Tree: Exemple



Blob & Tree: Exemple

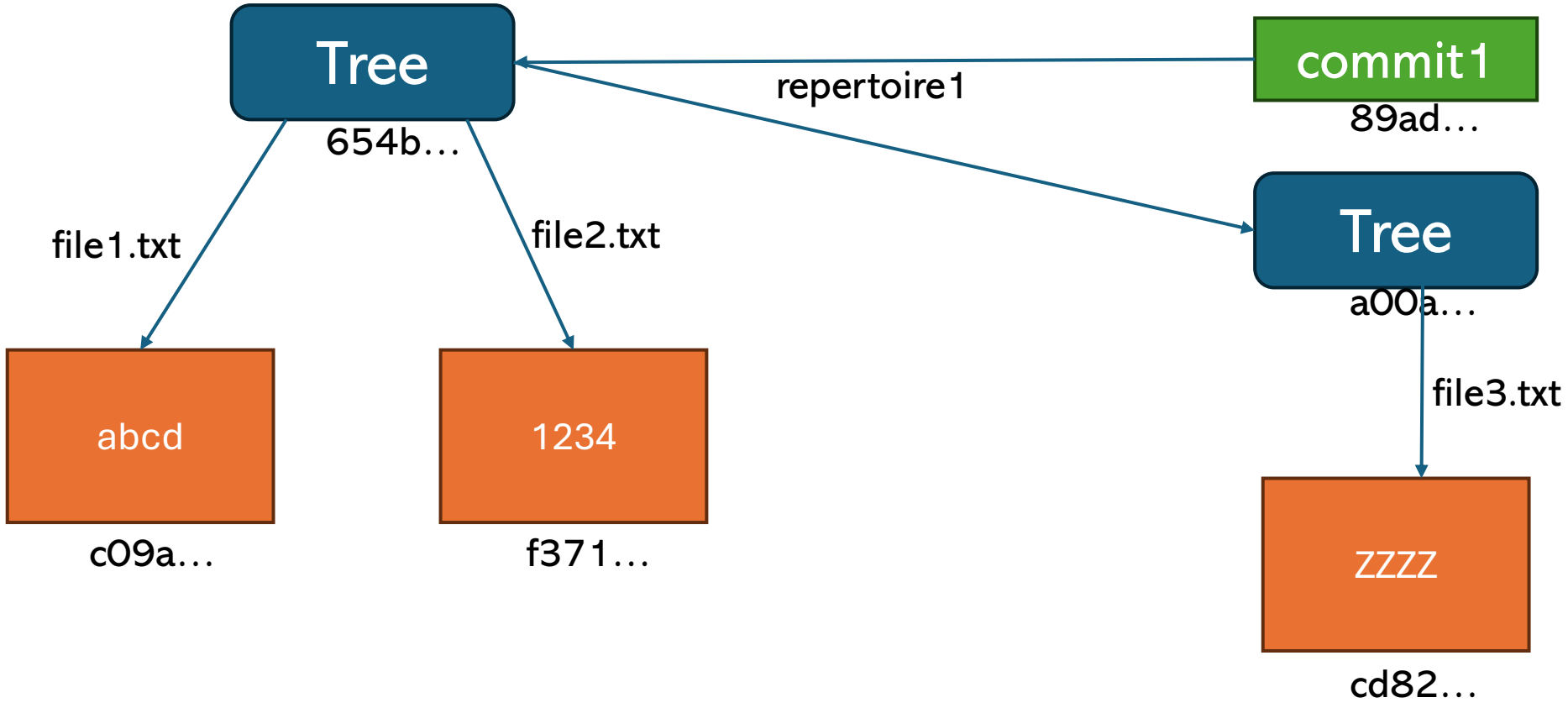


Commit

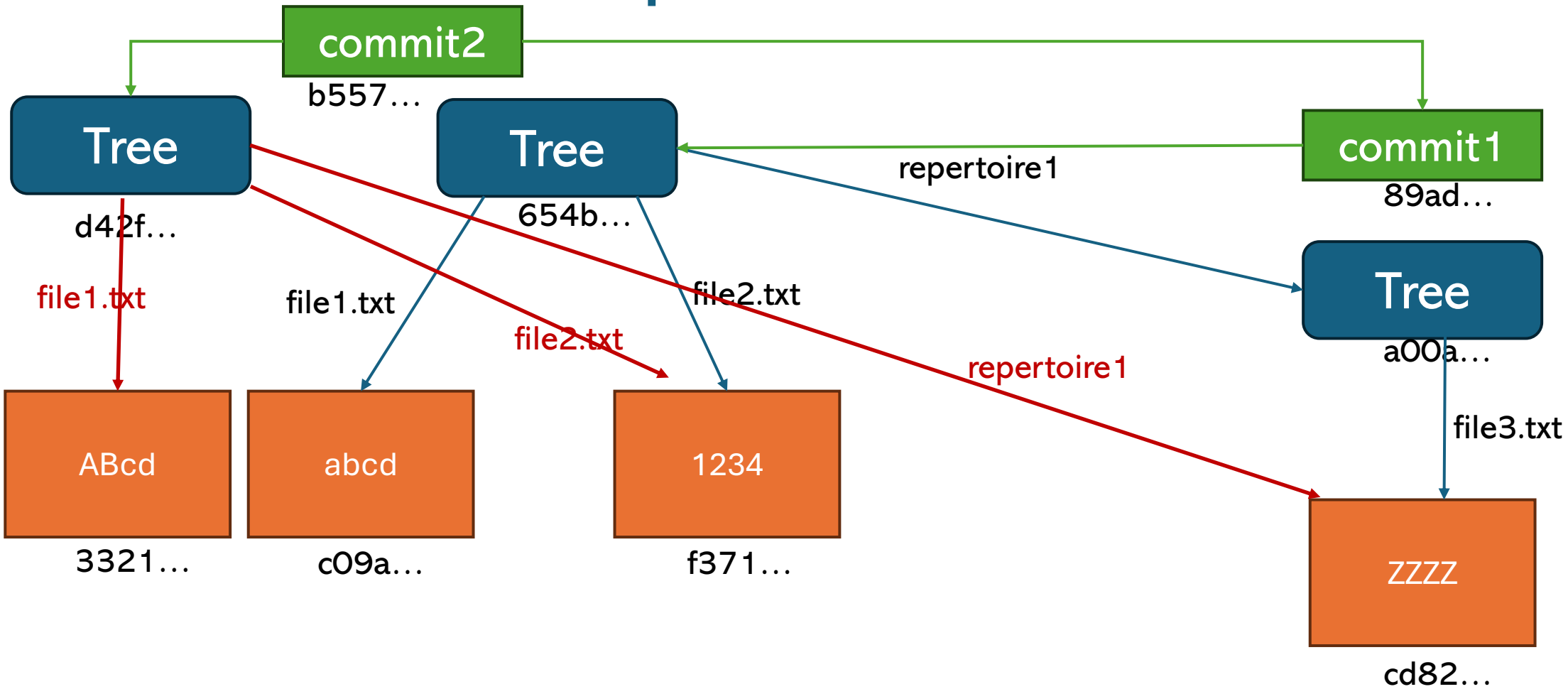
Un **Commit** stocke l'état d'une partie du dépôt à un instant donné.
Il contient :

- Un pointeur vers un Tree (arbre racine) dont on souhaite sauver l'état.
- Un pointeur vers un ou plusieurs autres Commits pour constituer un historique.
- Les informations sur l'auteur du Commit.
- Une description sous forme d'une chaîne de caractères.

Commit: Exemple



Commit: Exemple modification de file1.txt

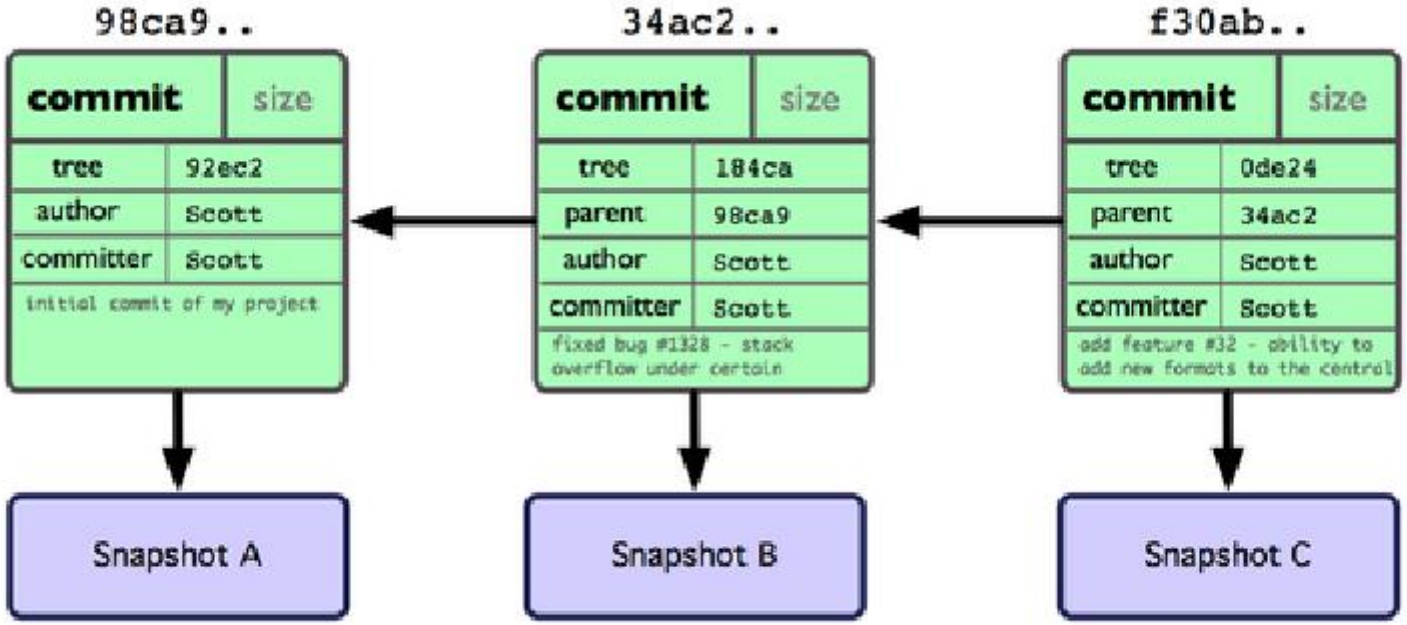


Tag

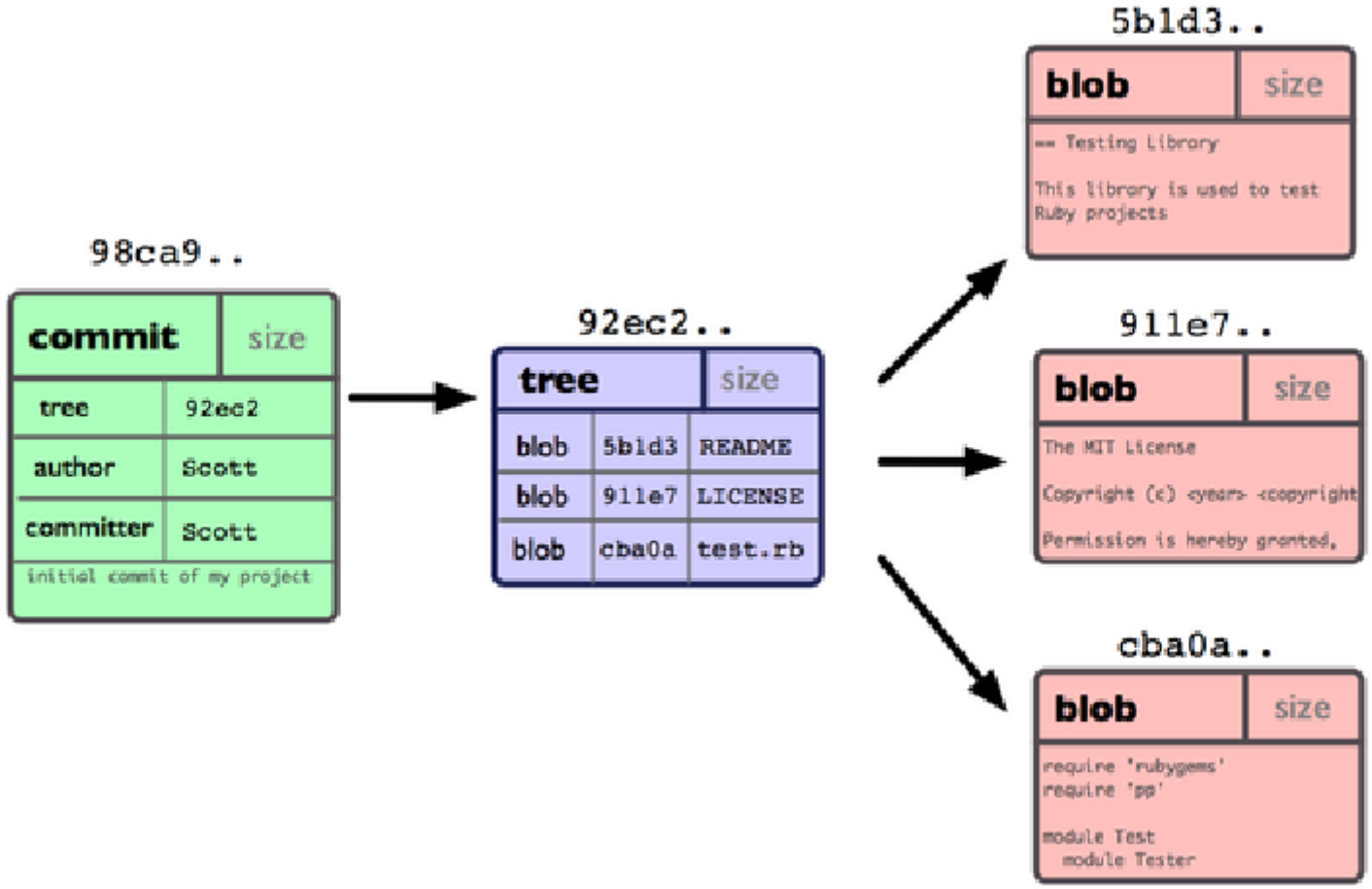
Un Tag permet d'identifier un des objets précédents à l'aide d'un nom.

- Il contient un pointeur vers un Blob, un Tree ou un Commit.
- Typiquement utilisé pour identifier l'état du dépot au moment d'une release.

Organisation des informations



Organisation des informations

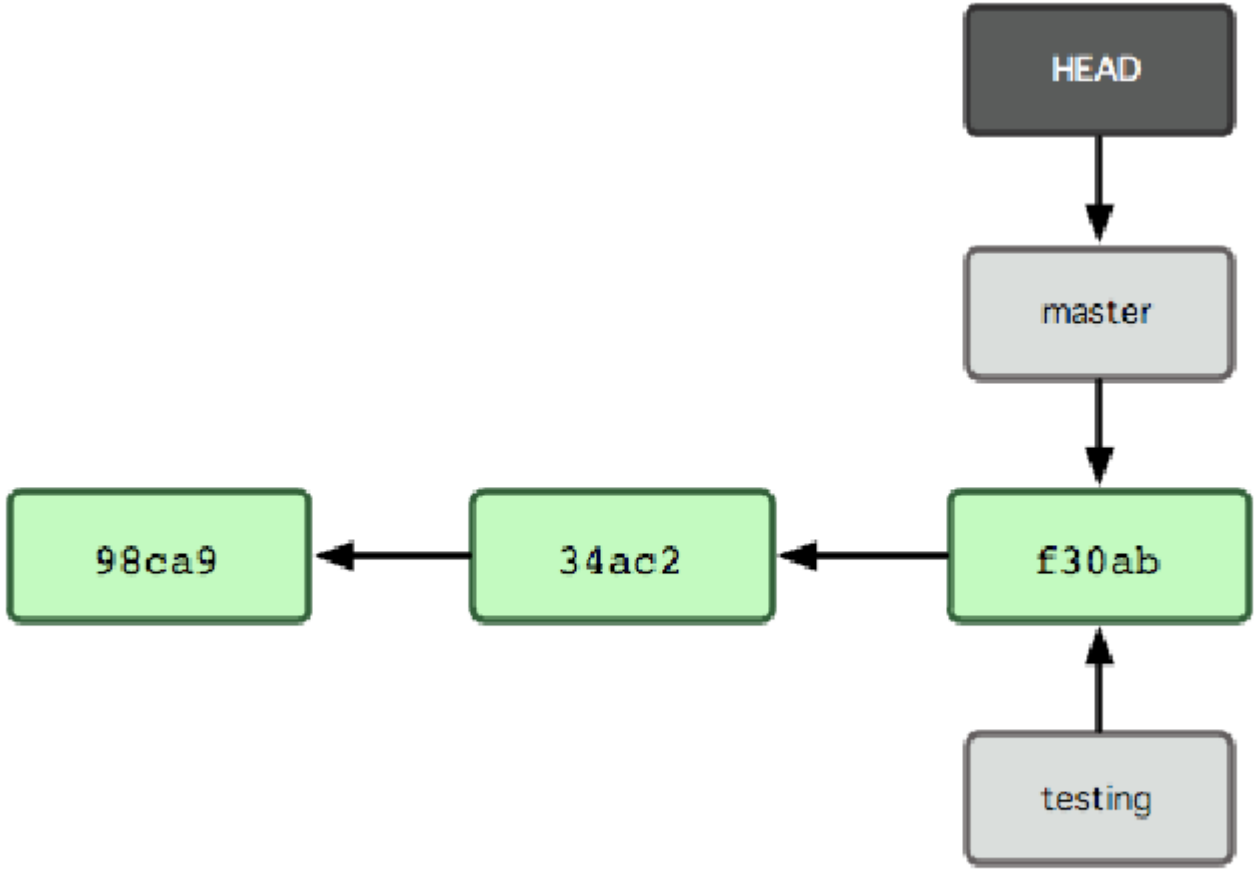


Références

L'historique du projet est un graphe de commit. Certaines références sur ce graphe sont utiles :

- master : référence la branche principale
- HEAD : par défaut, référence le commit le plus récent de la branche courante (sera le parent du prochain commit)
- HEAD-2 : deux commits avant la HEAD

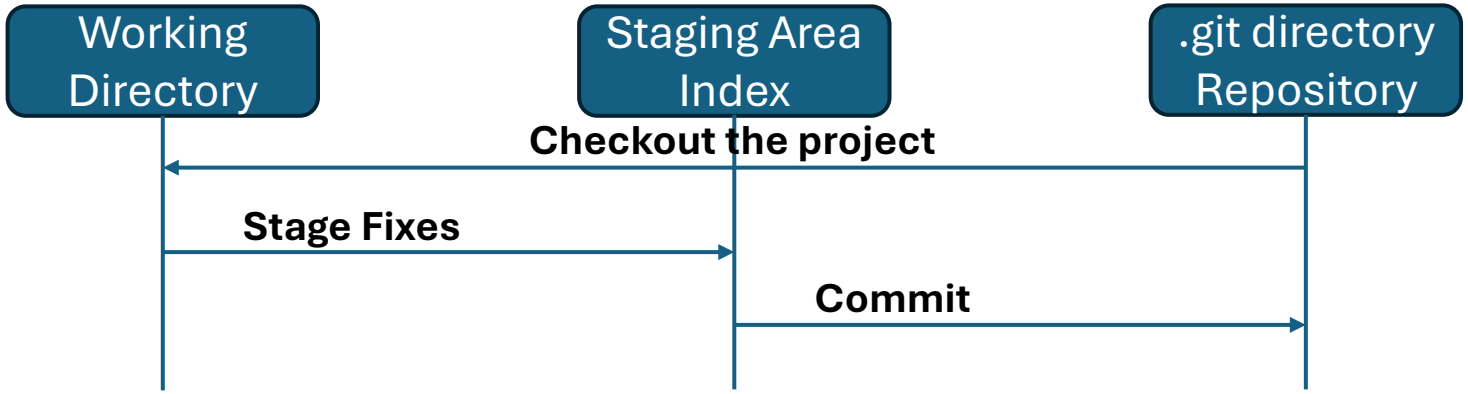
Références



Zone de stockage dans GIT

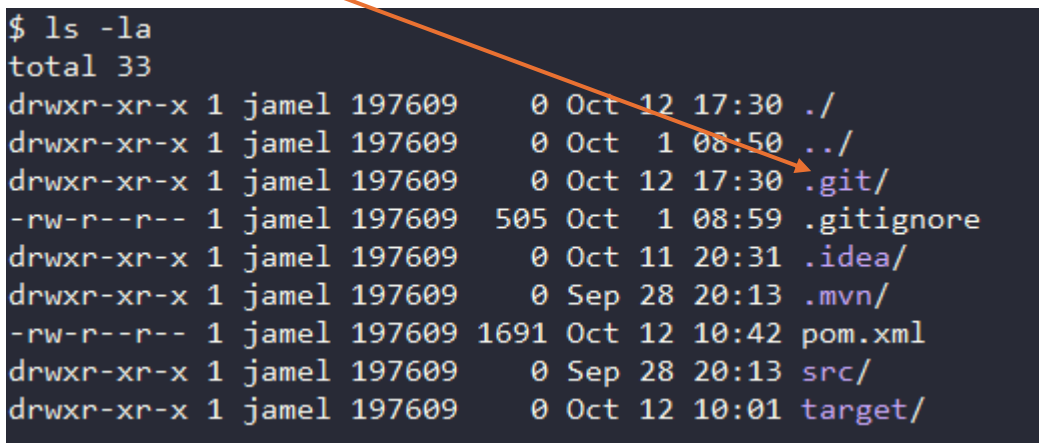
Découpage interne en trois zones:

- le répertoire de travail (working directory) local où sont réalisés les changements
- la ”staging area” (aussi appelé index) où sont pré-enregistrés les changements (en attente de commit)
- le dépôt git où sont enregistrés les changements



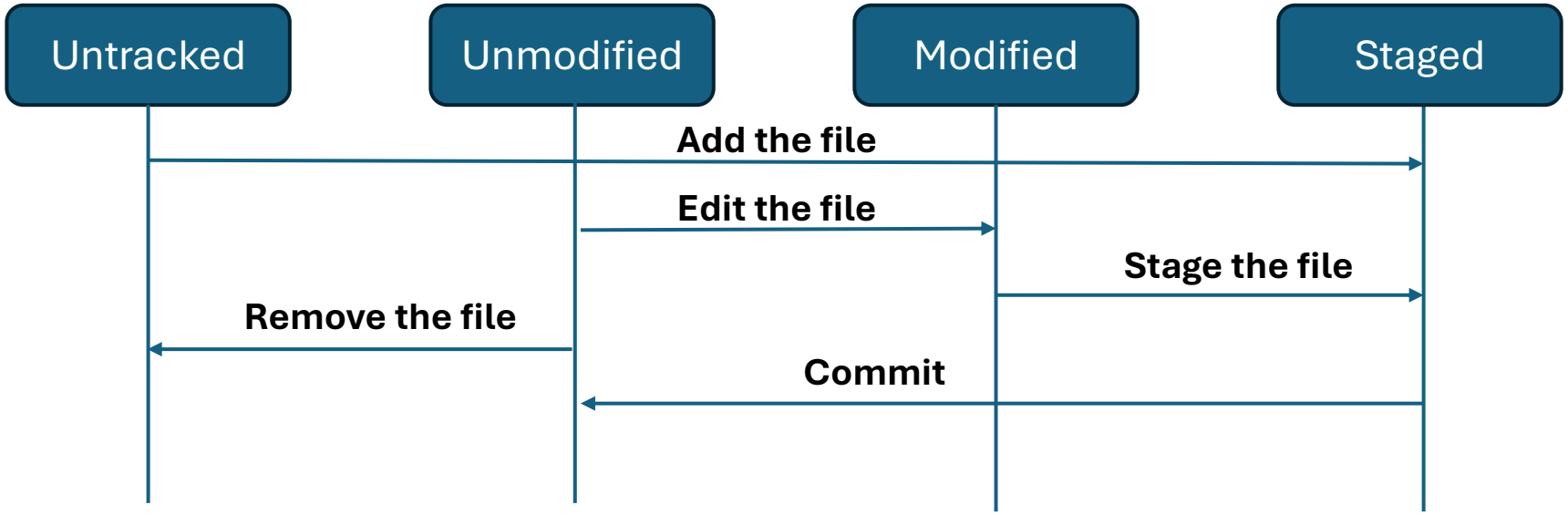
Emplacement du dépôt local

- Les données propres à git sont stockées dans un unique répertoire **.git** à la racine du projet. C'est le dépôt local.



Staging Area

- Sert à préparer les commits progressivement
- git commit enregistre les modifications indexées
- La staging area peut être bypassée : git commit -a



Plan du cours

I. Introduction

II. GIT

III. Utilisation de GIT

IV. Les bonnes pratiques

V. Synchronisation avec les dépôts distants

Les commandes

- Git est un ensemble de commandes. Les commandes sont de la forme:

```
$> git commande options
```

Exemple: `$> git add file1.java`
Manuel d'utilisation: `$> man git`
Aide générale: `$> git help`
Aide d'une commande: `$> git help command`

Configuration globale

- Configuration commune à tous les dépôts d'un utilisateur, voir le fichier $HOME/.gitconfig

- git config –global user.name "Jamel ESSOUSSI"
- git config –global user.email jamel.essoussi@gmail.com
- git config –global color.ui true

Création d'un dépôt serveur

- $ mkdir projet.git
- $ cd projet.git
- $ git --bare init

- Ne contient pas les fichiers versionnés mais juste l'historique
- On ne travaille pas sur cette version

Création d'un dépôt – initiation d'un dépôt

- $> cd myproject
- $> git init
- $> git add .
- $> git commit -m 'initial commit'
- $> git remote add origin git@gitserver:/XX/XX/project.git
- $> git push origin master

- Créer un répertoire local myproject pour stocker notre version du projet.
- Associer le dépôt local avec le dépôt distant
- Envoyer l'état initial du dépôt vers le serveur
- A partir de ce moment, tout le monde peut obtenir sa copie locale du dépôt en utilisant git clone

Cloner un projet existant

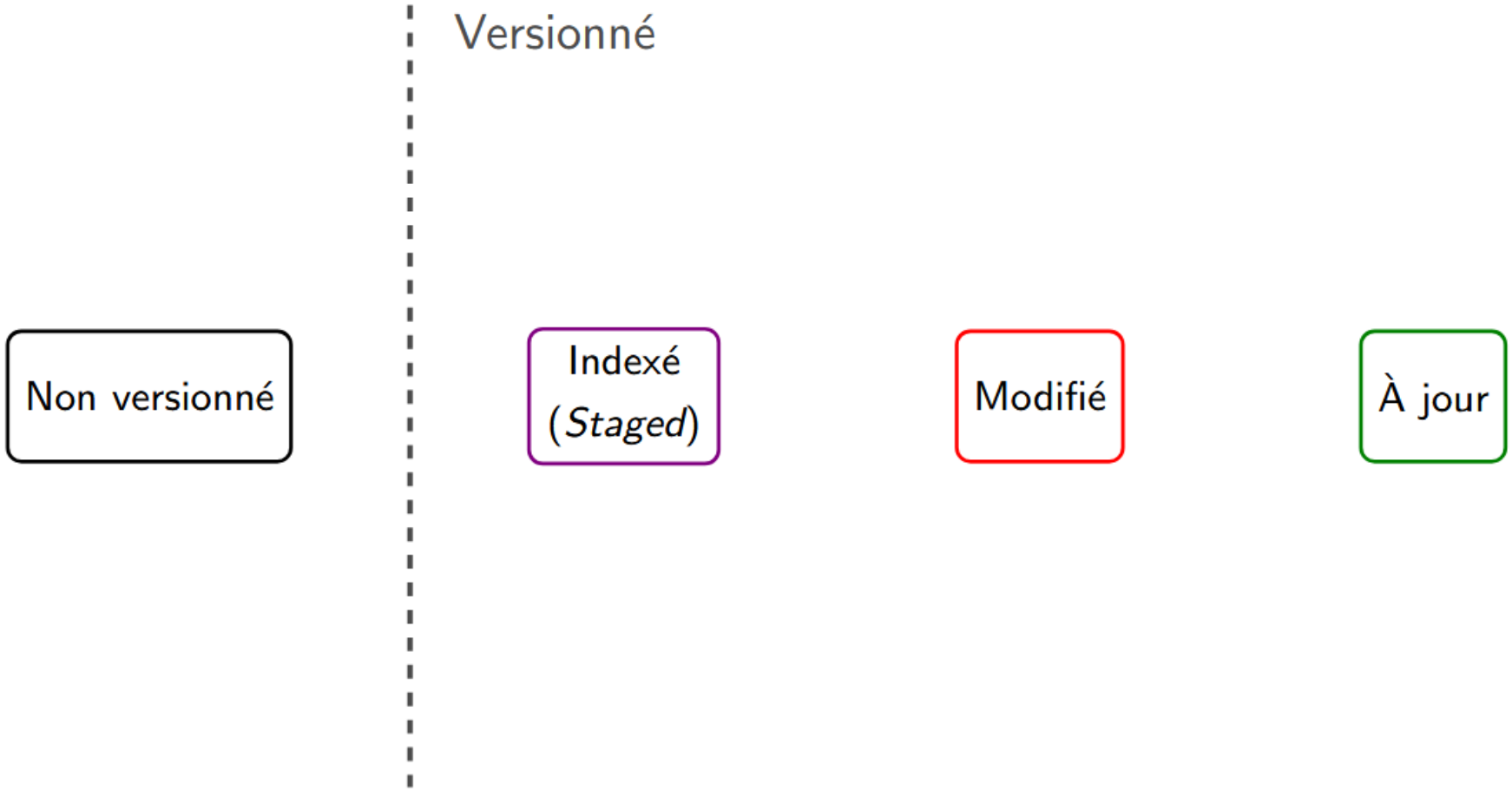
Très souvent, un dépôt existe déjà. On veut alors récupérer une copie de ce dépôt.

Cloner un dépôt

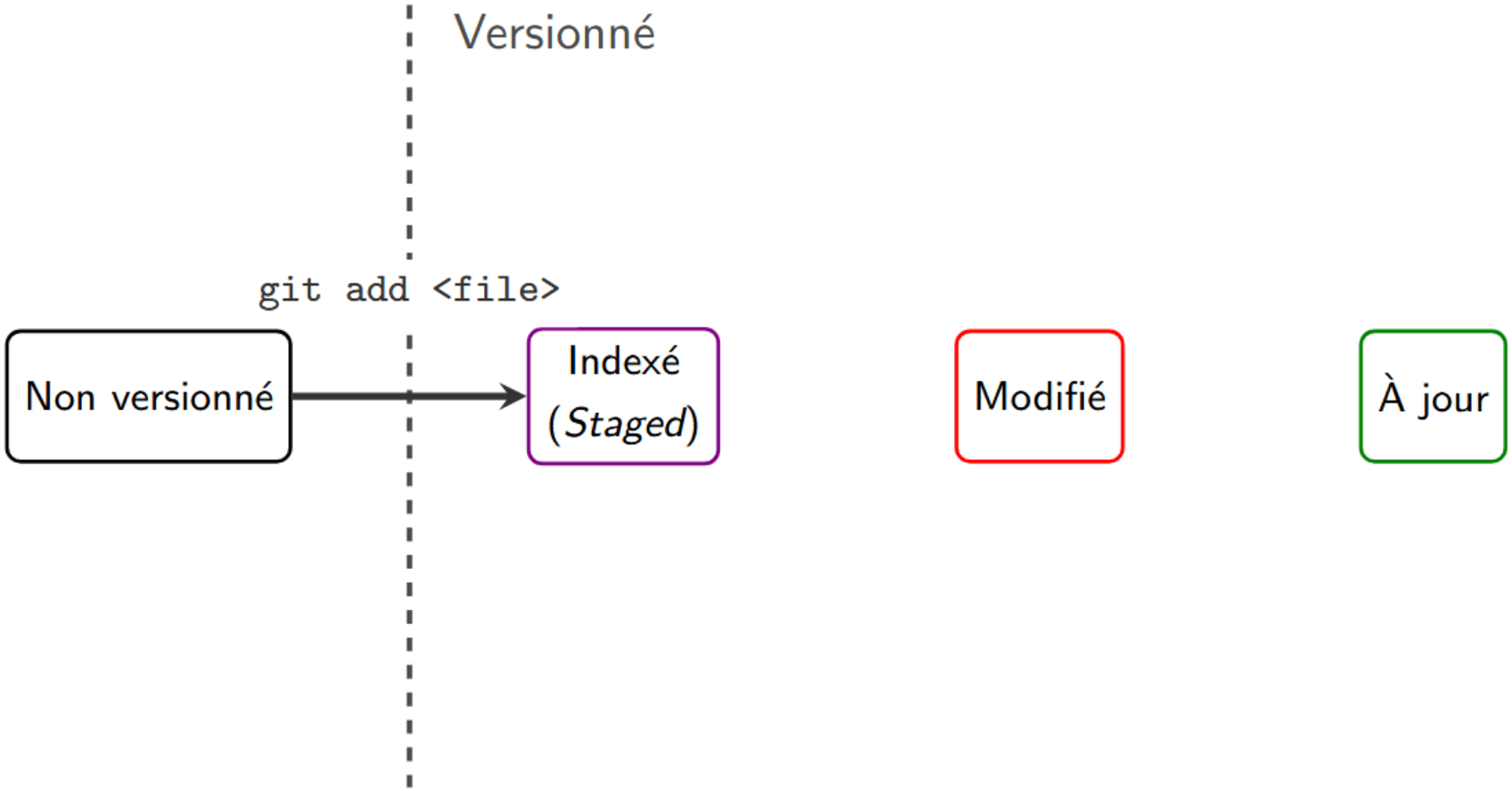
- $> git clone « URL »

  - Crée une copie locale du dépôt entier.
  - L'URL peut être de la forme:
    - file://./myproject/project.git
    - https://github.com/jessoussi/authentication-service.git
    - git@github.com:jessoussi/authentication-service.git

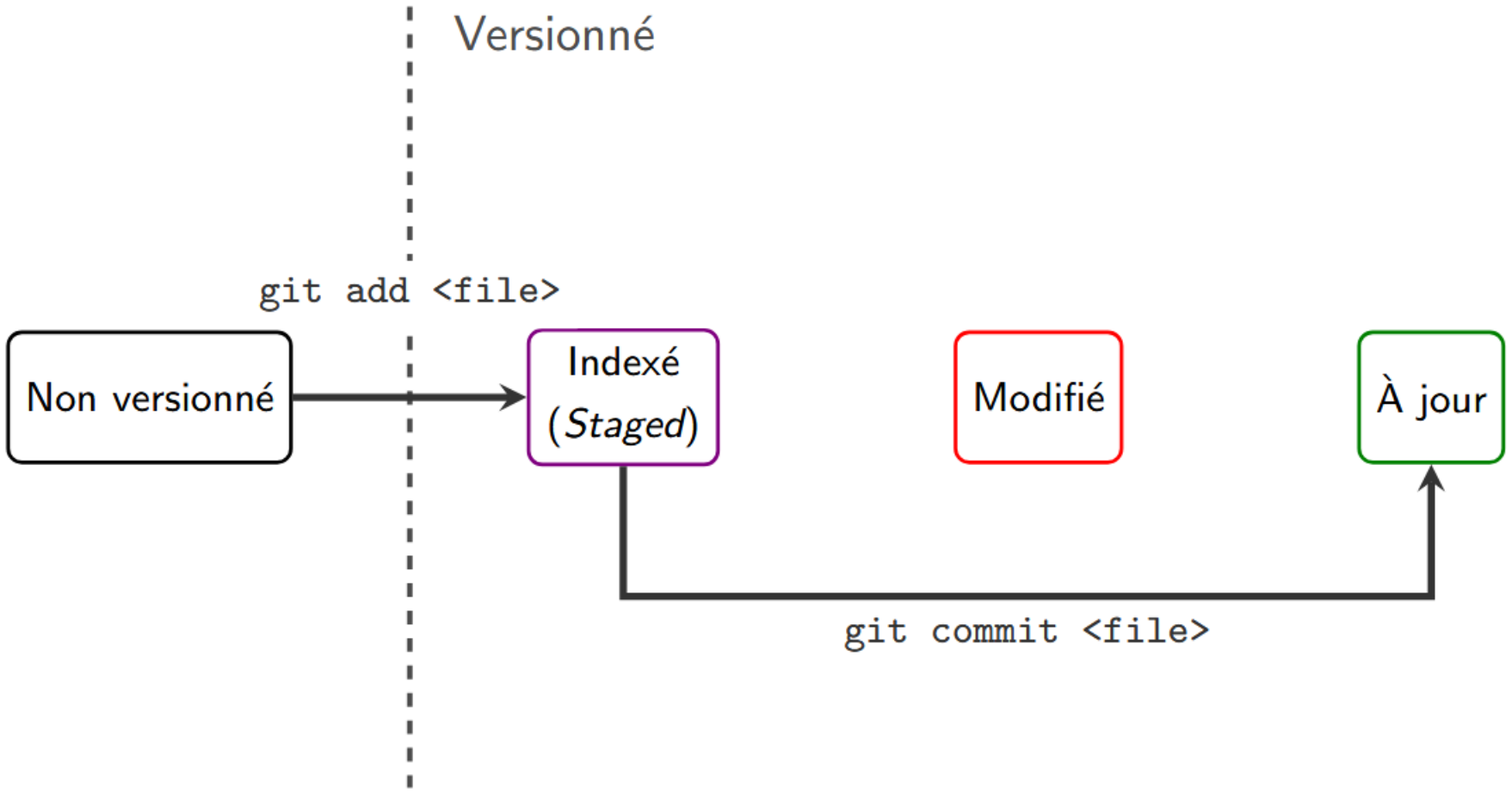
Le cycle de vie d'un fichier



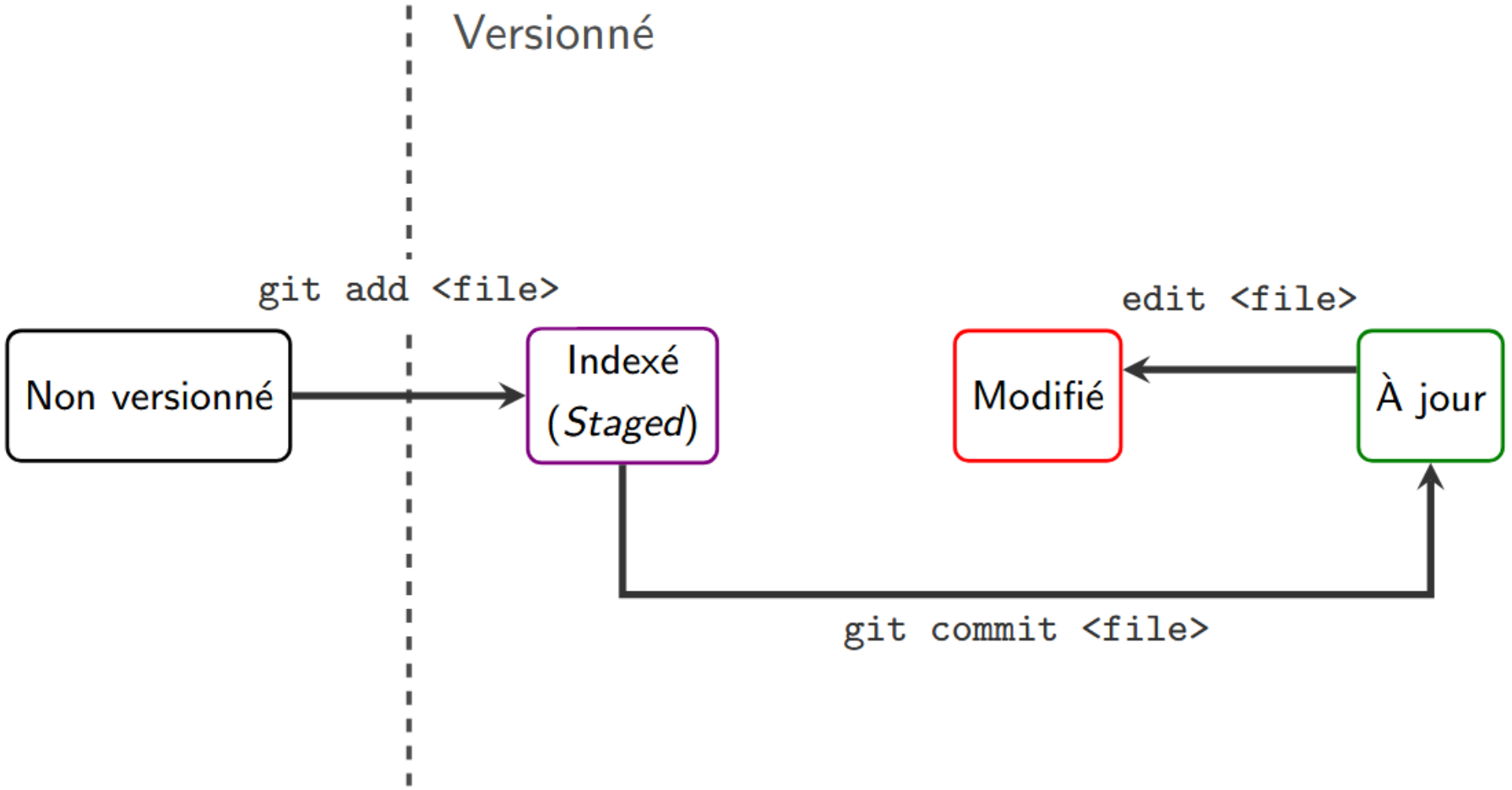
Le cycle de vie d'un fichier



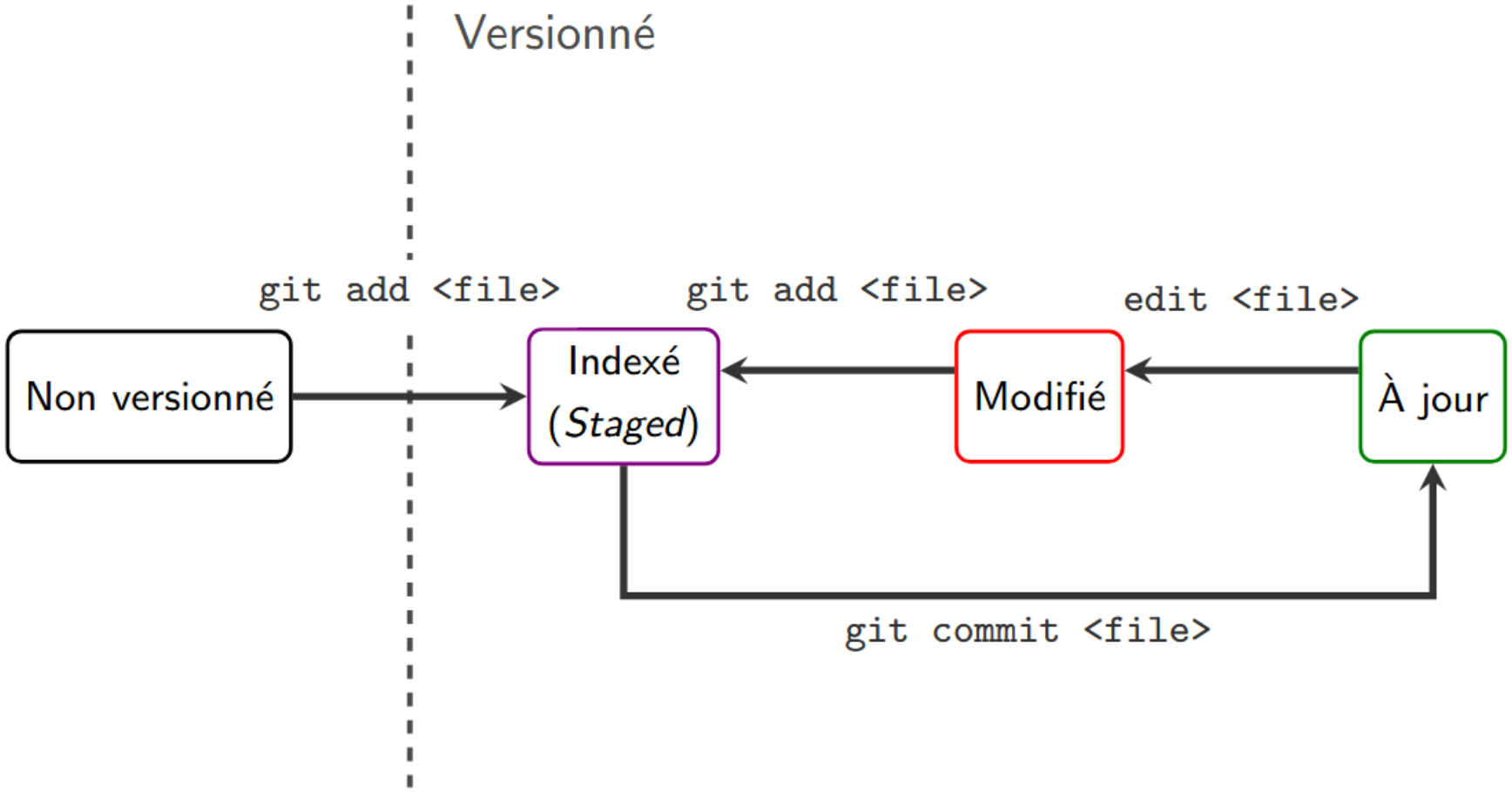
Le cycle de vie d'un fichier



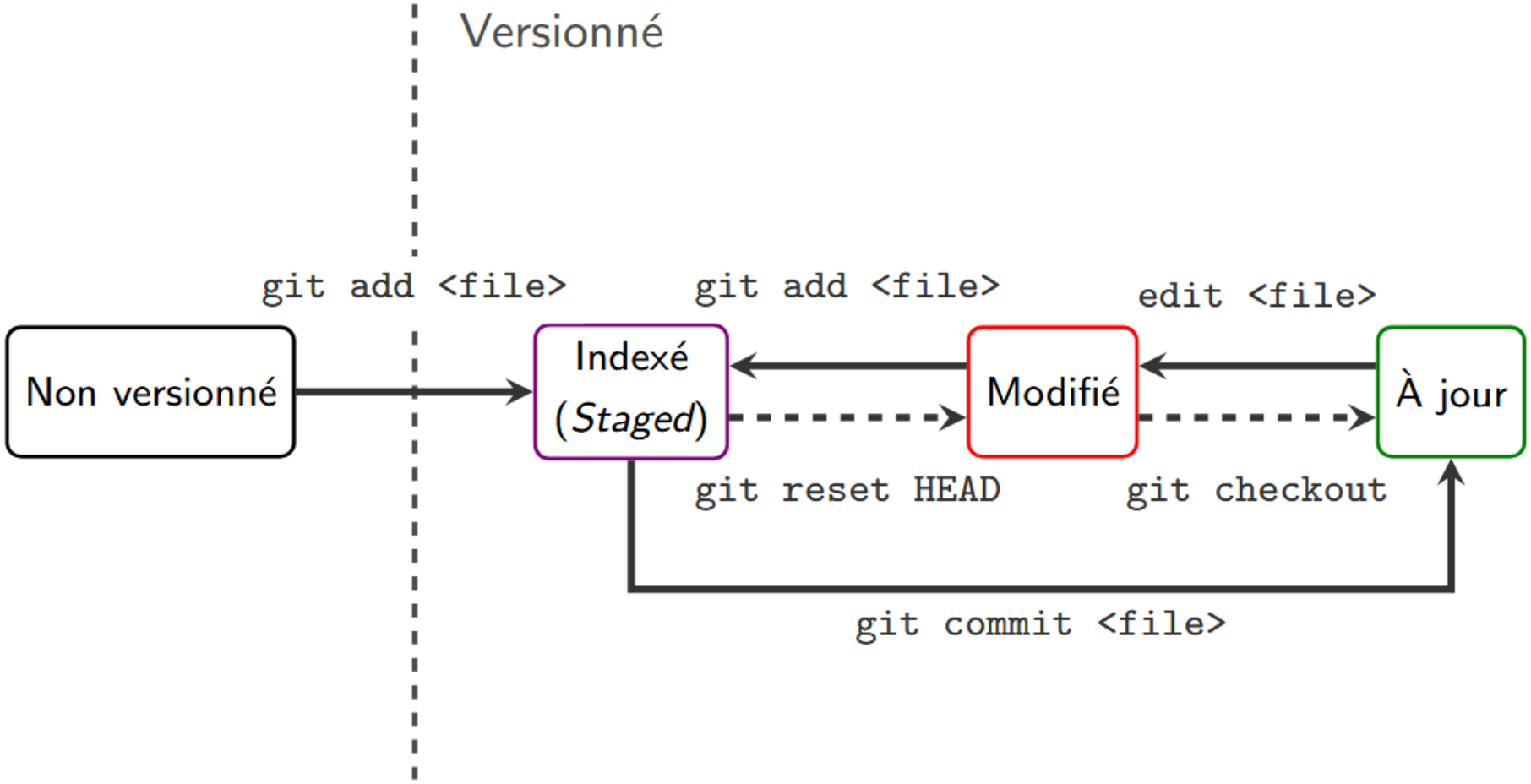
Le cycle de vie d'un fichier



Le cycle de vie d'un fichier



Le cycle de vie d'un fichier



Quelques commandes sur les fichiers

Commit d'un fichier

- $git add : Ajoute dans l'index un fichier à commiter dans son état actuel.
- $git commit : enregistre dans le dépôt local les modifications qui ont été ajoutées dans l'index par une commande add
- $git reset HEAD : supprime la référence d'un fichier de l'index ajouté par une commande add.

Souvent on veut simplement commiter toutes les modifications en cours

- $ git commit -a

Quelques commandes sur les fichiers

Exemple d'un commit

- $ echo "coucou" >hello.txt
- $ git add hello.txt
- $ git commit -m "description du commit«

- En l'absence de message d´ecrivant le commit, un fichier d´écrivant le commit est ouvert, vous invitant `a compl´eter la description.

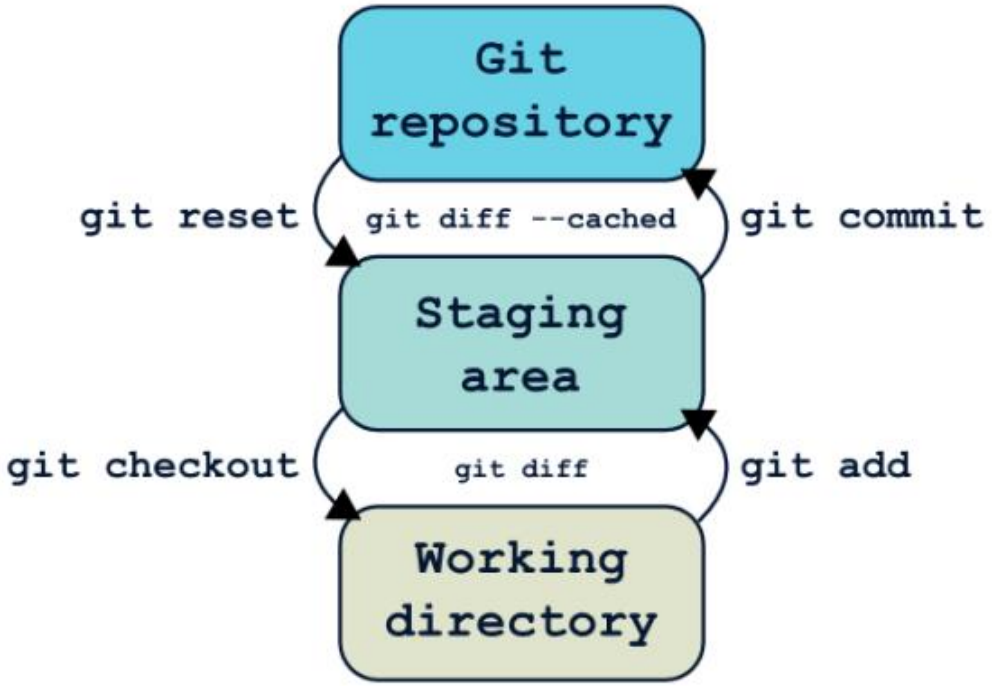
Quelques commandes sur les fichiers

Annuler des modifications

- Retirer un fichier indexé dans la staging area (inverse de git add) tout en gardant ses modifications dans le working dir :
  - $git reset nom fichier
- Annuler les modifications courantes d'un fichier ou répertoire du working dir en rappelant celles :
  - de la staging area : $git checkout nom fichier
  - d'un commit : $git checkout SHA1 nom fichier

Quelques commandes sur les fichiers

Annuler des modifications



Quelques commandes- Méta-données

- $git status
  - Permet de connaître l'état courant de l'index, les modifications indexées, non indexées, non versionnées.
- $git diff
  - Consulter les modifications en cours du contenu des fichiers suivis.
- $git diff sha1 autre sha1
  - Consulter les modifications du contenu des fichiers entre deux commits.
- $git log
  - Consulter l'historique des commits.

Les branches

Dans Git

- Les branches permettent de réaliser un développement en parallèle de la branche principale afin de limiter les impacts.
- Une branche est un pointeur sur un commit
- Chaque commit pointe vers son prédécesseur
- La variable HEAD pointe sur la branche sur laquelle on travaille actuellement.

Branches – les commandes

- $git branch Liste les branches avec une * pour la branche active.
- $git branch <nom> Crée une nouvelle branche <nom>
- $git branch –m Permet de renommer une branche.
- $git branch –d Permet de supprimer une branche.
- $git checkout Change (ou/et crée) de branche active.
- $git show-branch : Affiche les branches et leurs commits.

Branches – les commandes

```
$git branch
    * master
$git branch maNouvelleBranche
$git branch
      maNouvelleBranche
    * master
$git checkout maNouvelleBranche
$git branch
    * maNouvelleBranche
      master
```

Création d'une nouvelle branche

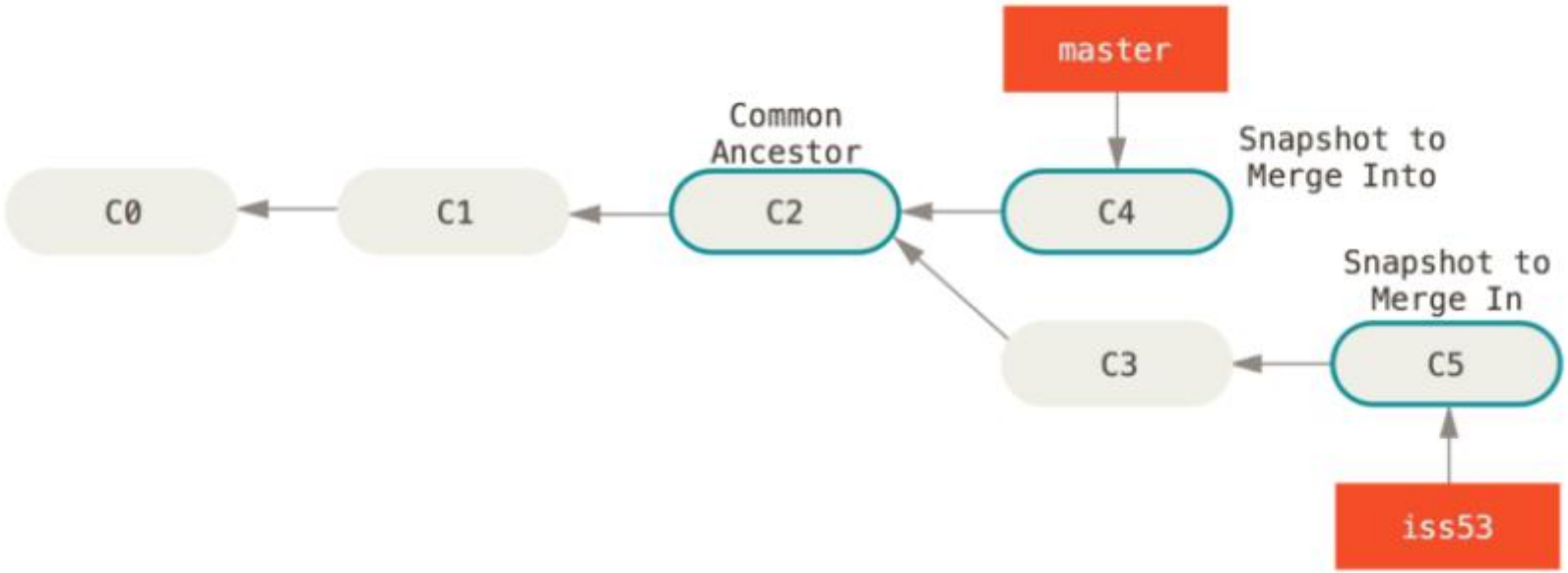
Basculer sur la nouvelle branche

Branches – les merges

- $git checkout brancheDestination
- $git merge brangeSource

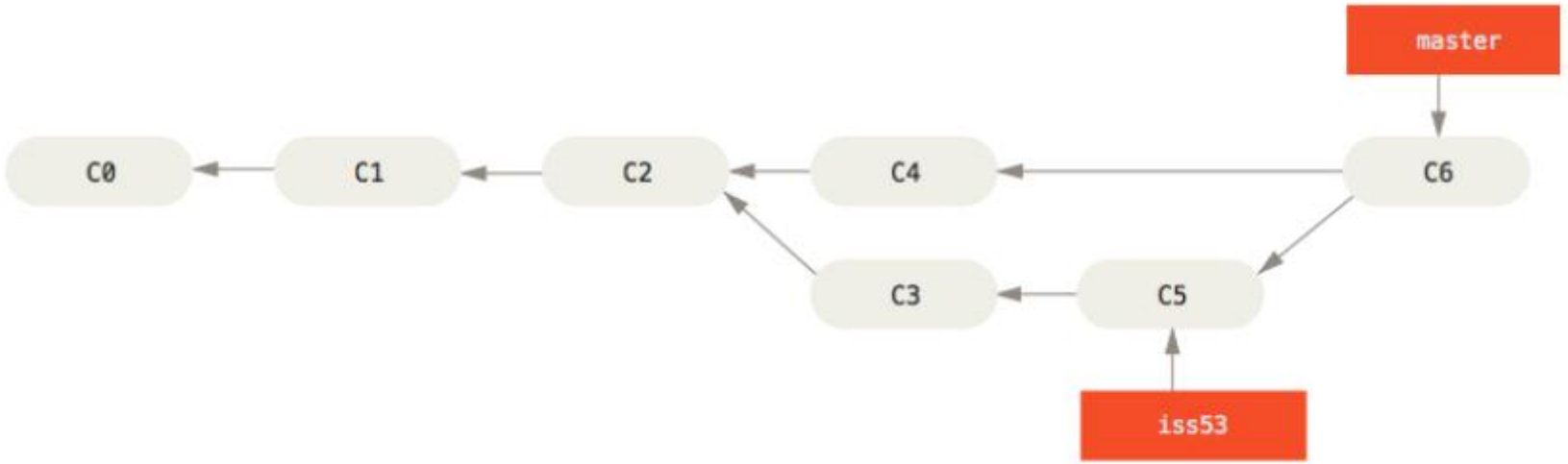
- Créé un commit qui a pour parent les deux branches
- La branche courante avance à ce commit
- La source ne bouge pas, mais devient un fils du nouveau commit

Branches – les merges



Branches avant fusion

Branches – les merges

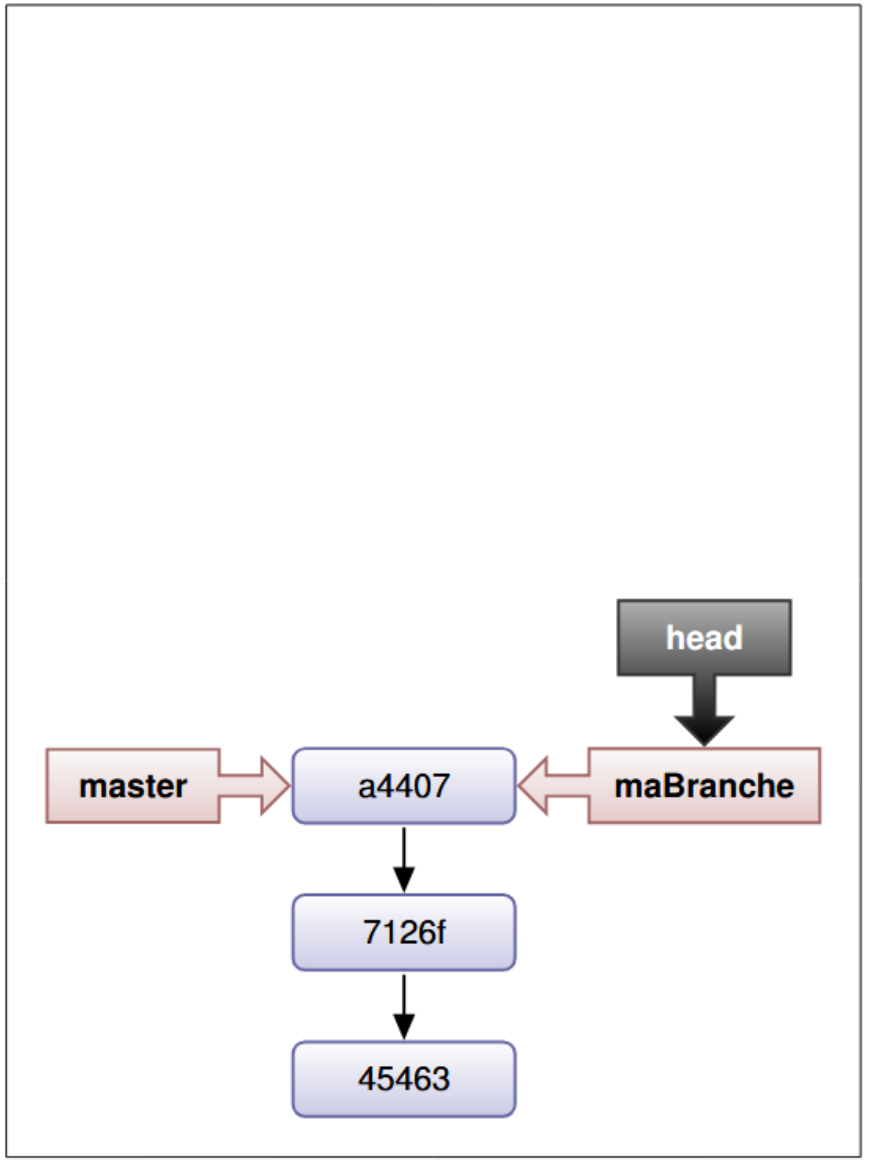


Branches après fusion

Branches

Basculer sur la branche maBranche

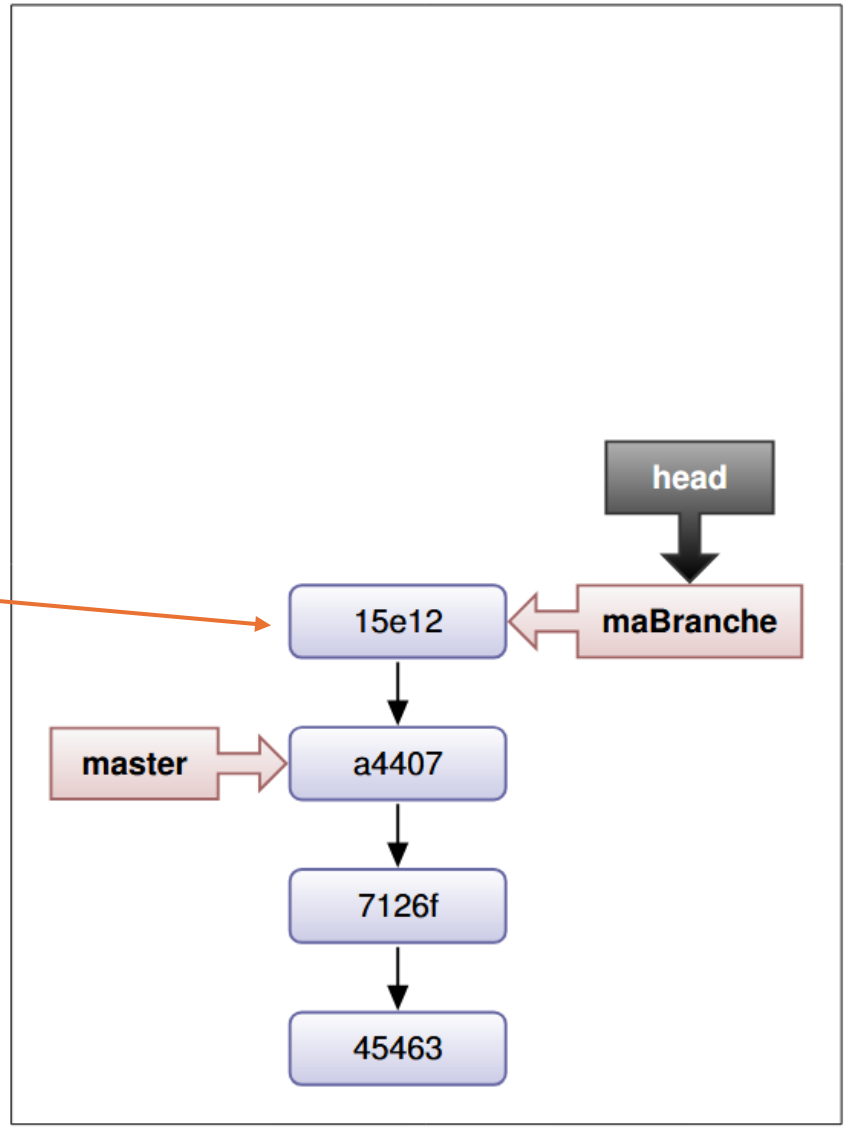
```
ls
 foo.txt dir

git branch maBranche

git checkout maBranche
```



Branches

Ajout du fichier fichier1.txt dans la branche maBranche

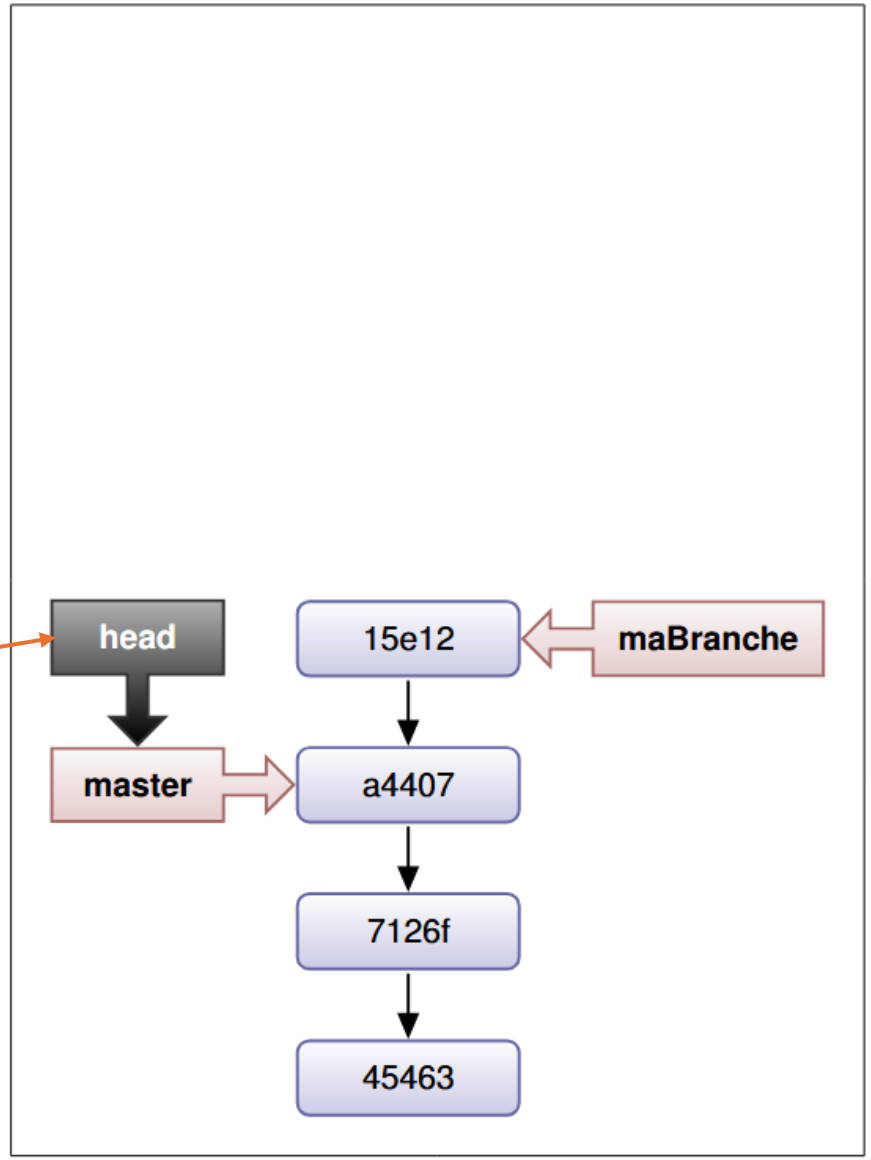
```
ls
 foo.txt dir

git branch maBranche

git checkout maBranche

touch fichier1.txt
ls
 dir fichier1.txt foo.txt
git add fichier1.txt
git commit -m "Add fichier1.txt"
```



Branches

Suppression de la branche master

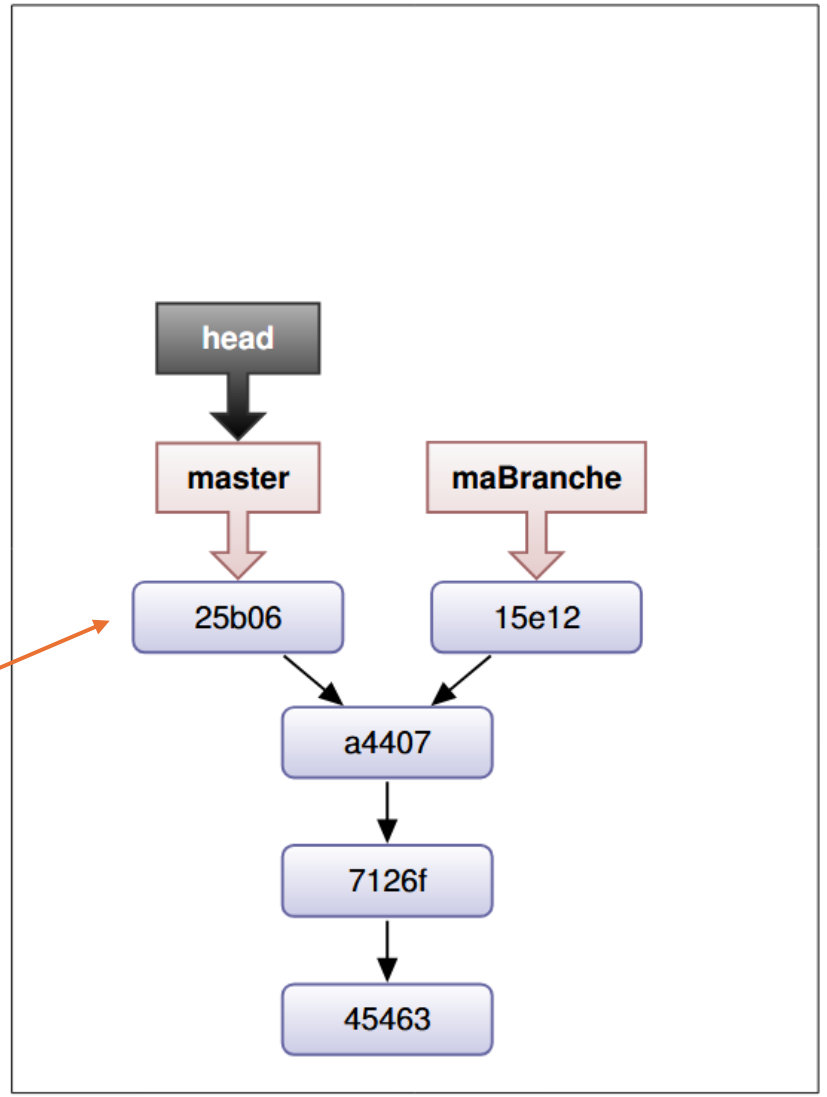
```
ls
 foo.txt dir

git branch maBranche

git checkout maBranche

touch fichier1.txt
ls
 dir fichier1.txt foo.txt
git add fichier1.txt
git commit -m "Add fichier1.txt"

git checkout master
ls
 dir foo.txt
```



Branches

Ajout du fichier fichier2.txt dans la branche master

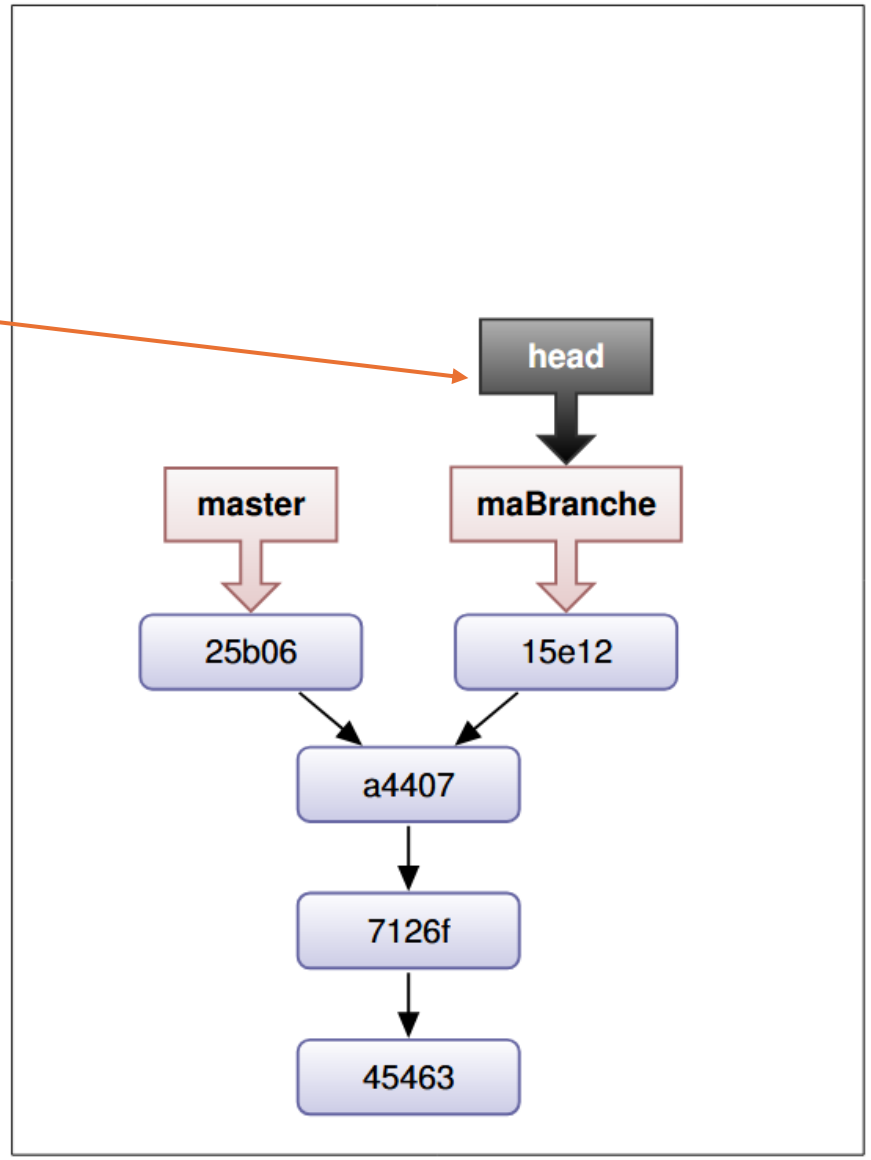
```
ls
 foo.txt dir

git branch maBranche

git checkout maBranche

touch fichier1.txt
ls
 dir fichier1.txt foo.txt
git add fichier1.txt
git commit -m "Add fichier1.txt"

git checkout master
ls
 dir foo.txt

touch fichier2.txt
git add fichier2.txt
git commit -m "Add fichier2.txt"
```



Branches

Suppression de la branche maBranche

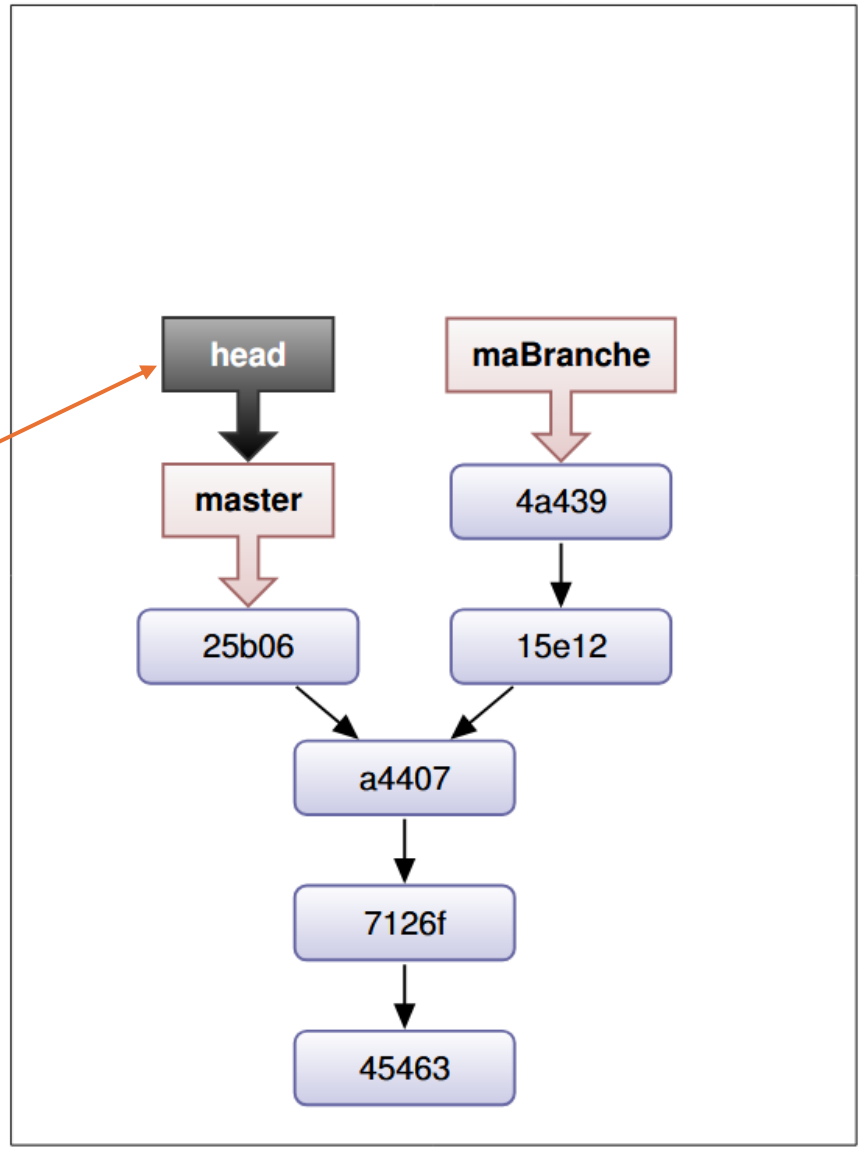
```
ls
 dir fichier2.txt foo.txt

git checkout maBranche
ls
 dir fichier1.txt foo.txt
```



Branches

Basculer sur la branche master

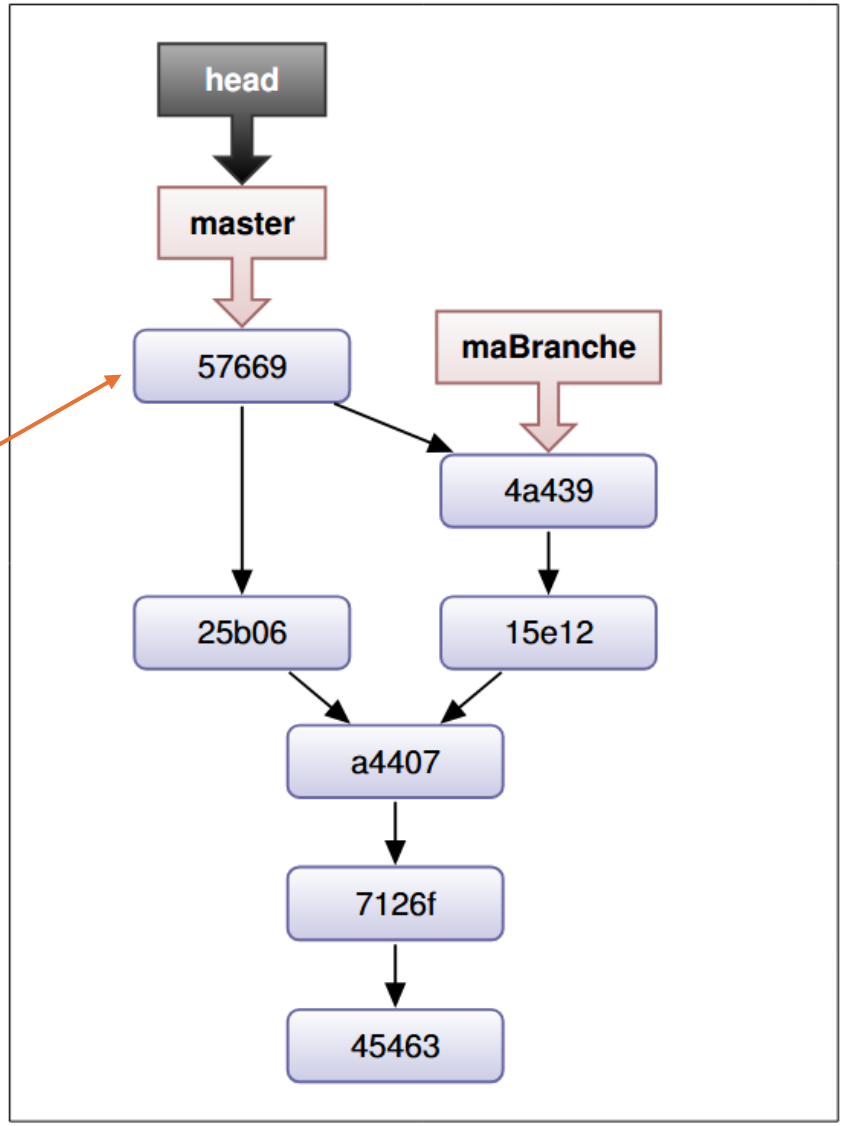
```
ls
 dir fichier2.txt foo.txt

git checkout maBranche
ls
 dir fichier1.txt foo.txt

echo "titi" > fichier1.txt
git commit -am "Modif
fichier1.txt"

git checkout master
```



Branches

Merger la branche maBranche dans la branche master

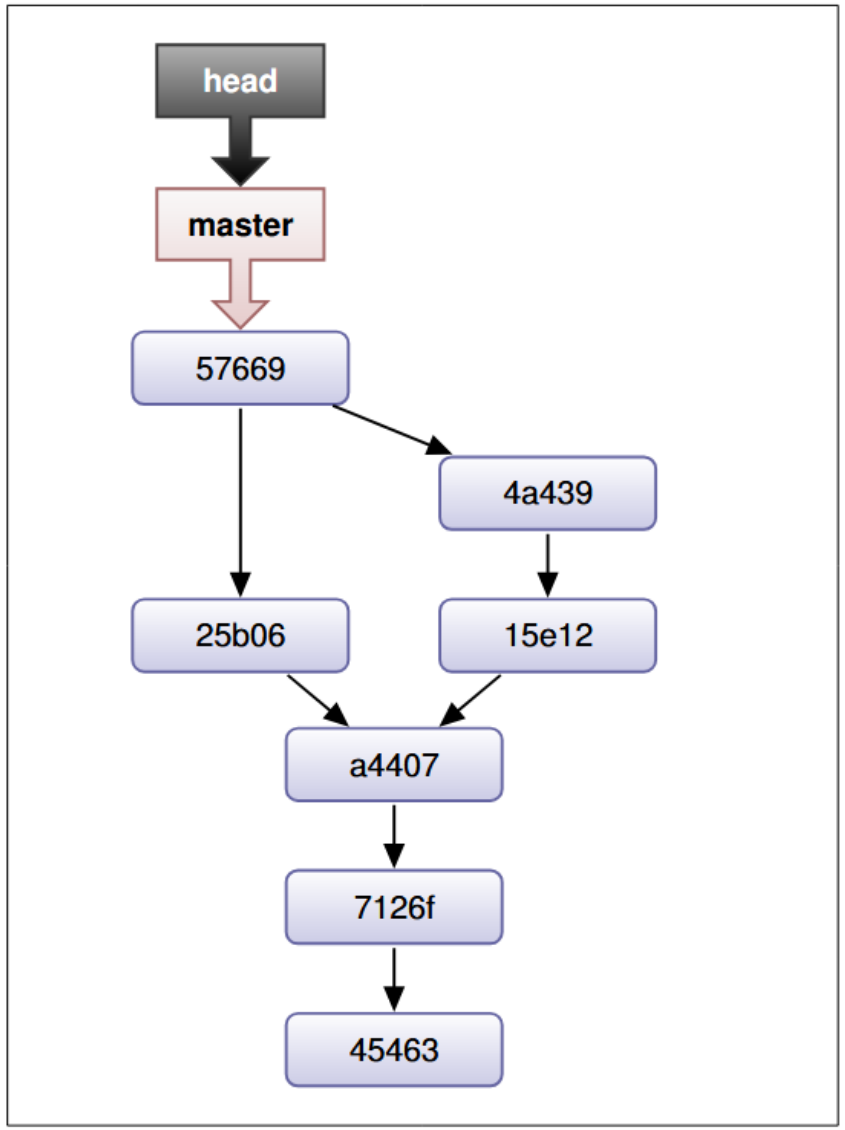
```
ls
 dir fichier2.txt foo.txt

git checkout maBranche
ls
 dir fichier1.txt foo.txt

echo "titi" > fichier1.txt
git commit -am "Modif
fichier1.txt"

git checkout master
git merge maBranche
ls
 dir fichier1.txt fichier2.txt
foo.txt
```



Branches

Suppression de la branche maBranche

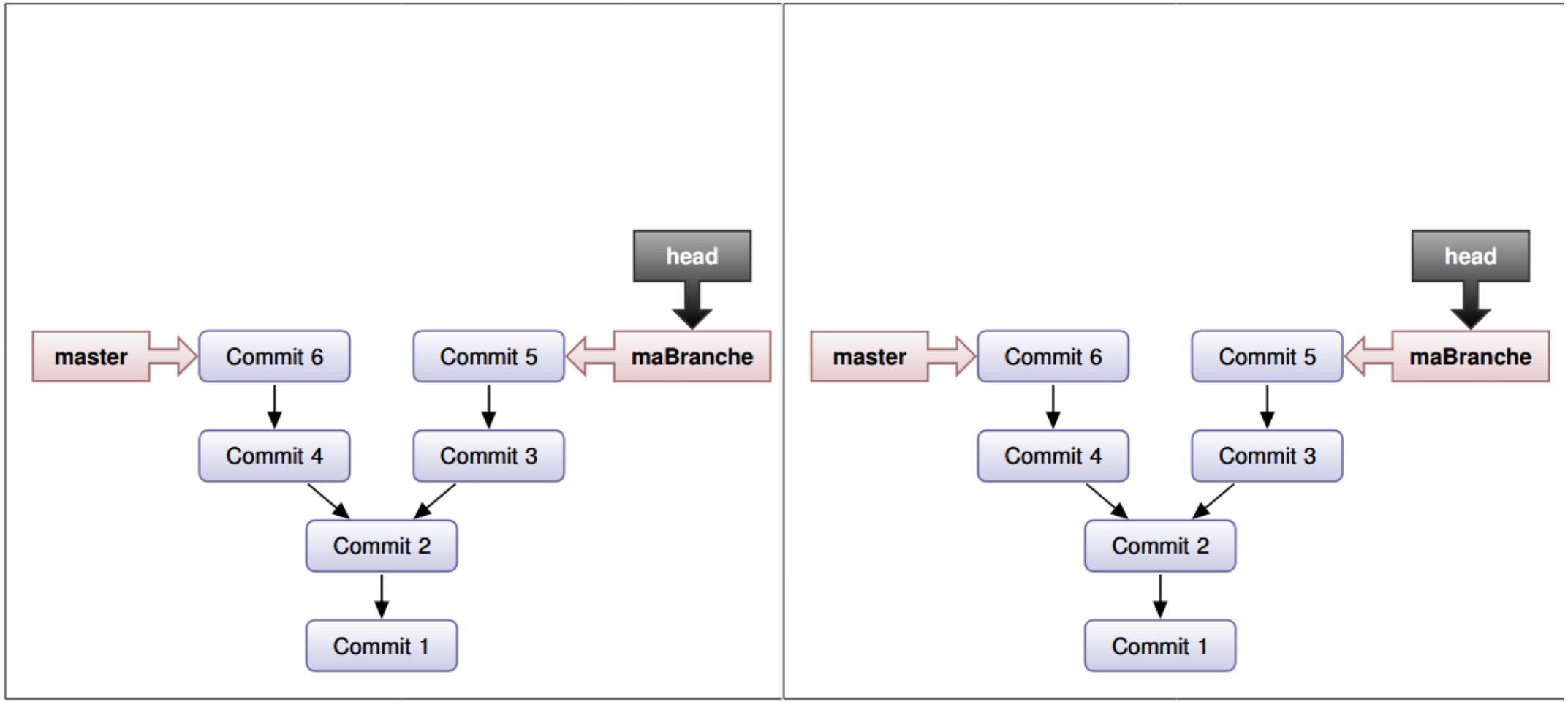
```
ls
 dir fichier2.txt foo.txt

git checkout maBranche
ls
 dir fichier1.txt foo.txt

echo "titi" > fichier1.txt
git commit -am "Modif
fichier1.txt"

git checkout master
git merge maBranche
ls
 dir fichier1.txt fichier2.txt
foo.txt

git branch -d maBranche
```



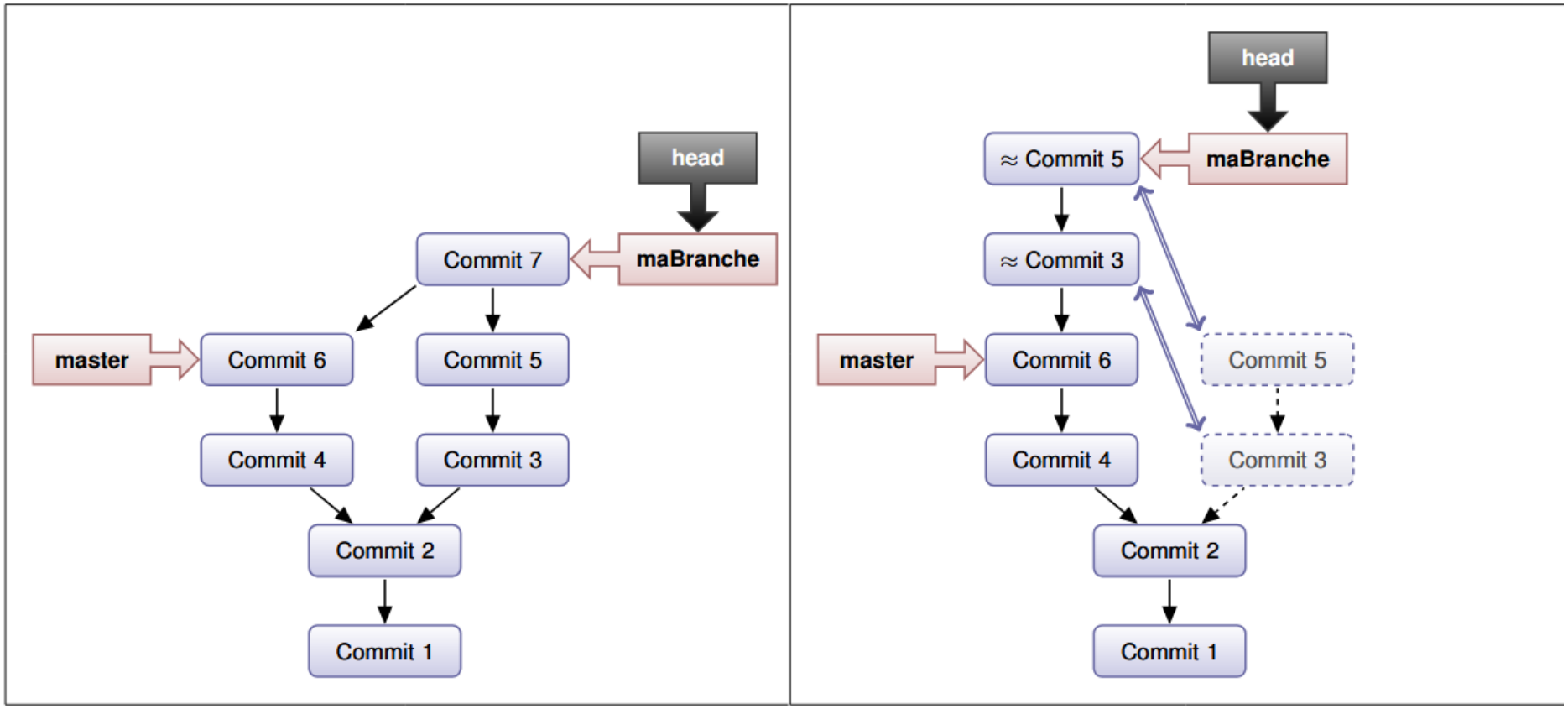
Branches – Rebase

- Autre manière de fusionner 2 branches
- Fusionne entièrement la branche source dans la branche destination
- Permet de simplifier l'historique
- Ne jamais rebaser des commits qui ont déjà été poussés sur un dépôt public

Branches – Merge VS Rebase



Branches – Merge VS Rebase




```
git checkout maBranche
git merge master
```

```
git checkout maBranche
git rebase master
```

Branches – Gestion des conflits

Dus à des modifications différentes au même endroit dans un même fichier

Git entre en mode résolution des conflits. Deux manières d'en sortir :

- Ne pas fusionner : $git merge –abort
- Résoudre les conflits :
  - Editer les fichiers sources de conflits
  - Terminer la résolution par git commit

Branches – Revenir en arrière

Cas de modifications non commitées

- Restaurer mon fichier dans la dernière version de l'index:

  $git checkout -- monfichier

  - Utilisation de "--": spécifie que monfichier désigne un fichier et pas une branche.
- Restaurer mon fichier dans la dernière version commitée:

  $git checkout HEAD monfichier
- Restaure tous les fichiers du répertoire courant:

  $git checkout .

Branches – git checkout

Une commande qui peut prêter à confusion

- Permet de naviguer dans les branches
- Permet de modifier le contenu de fichiers

Nouvelles commandes depuis git 2.23

- $git switch pour changer de branche
- $git restore pour modifier le contenu d'un ficher
- A utiliser en remplacement de git checkout

Branches – Revenir en arrière

Cas de modifications commitées

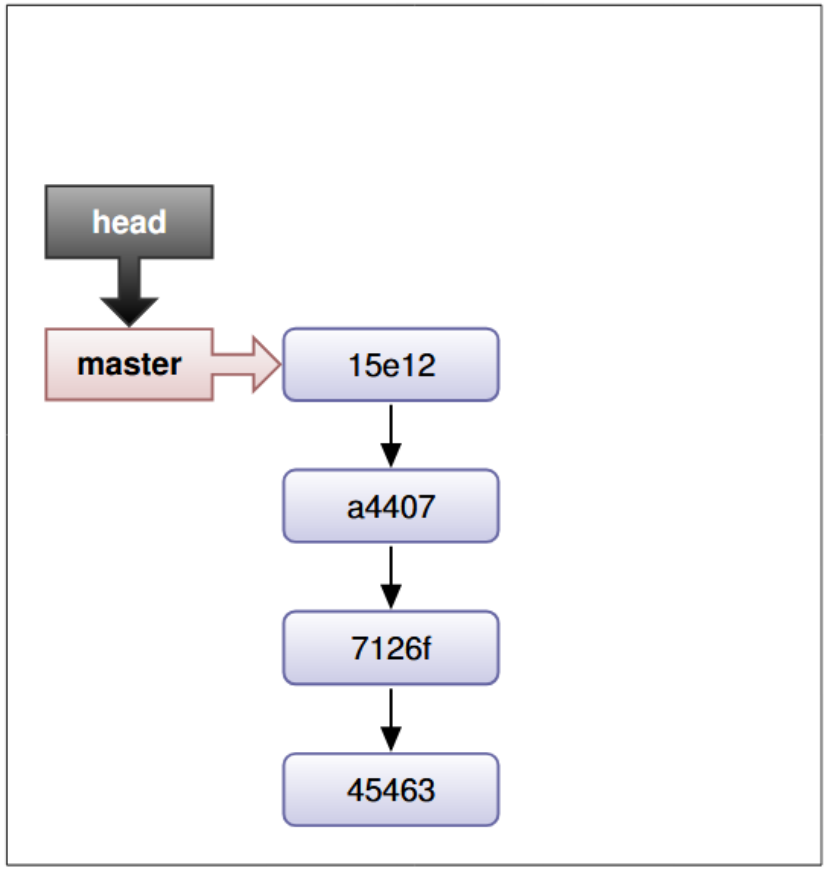
Trois commandes disponibles:

- $git amend : modifier le dernier commit
  - Ajoute des fichiers au commit
  - Changer le message de commit
- $git revert : annuler un commit par un autre commit
- $git reset : rétablir la situation d'un ancien commit

Si l'erreur a été rendue publique, la seule bonne pratique est revert.

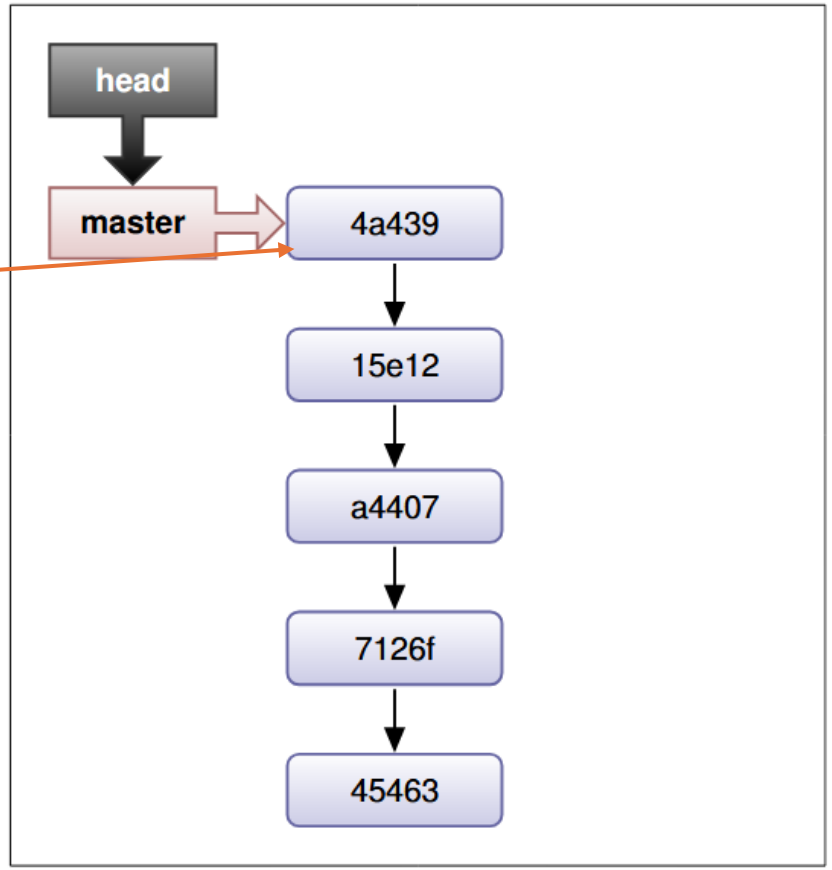
Branches – Amend modification du dernier commit

```
ls
  foo.txt dir
```



Branches – Amend modification du dernier commit

```
ls
 foo.txt dir

touch bar.txt
git commit -m "Ajou d'un fichier."
```

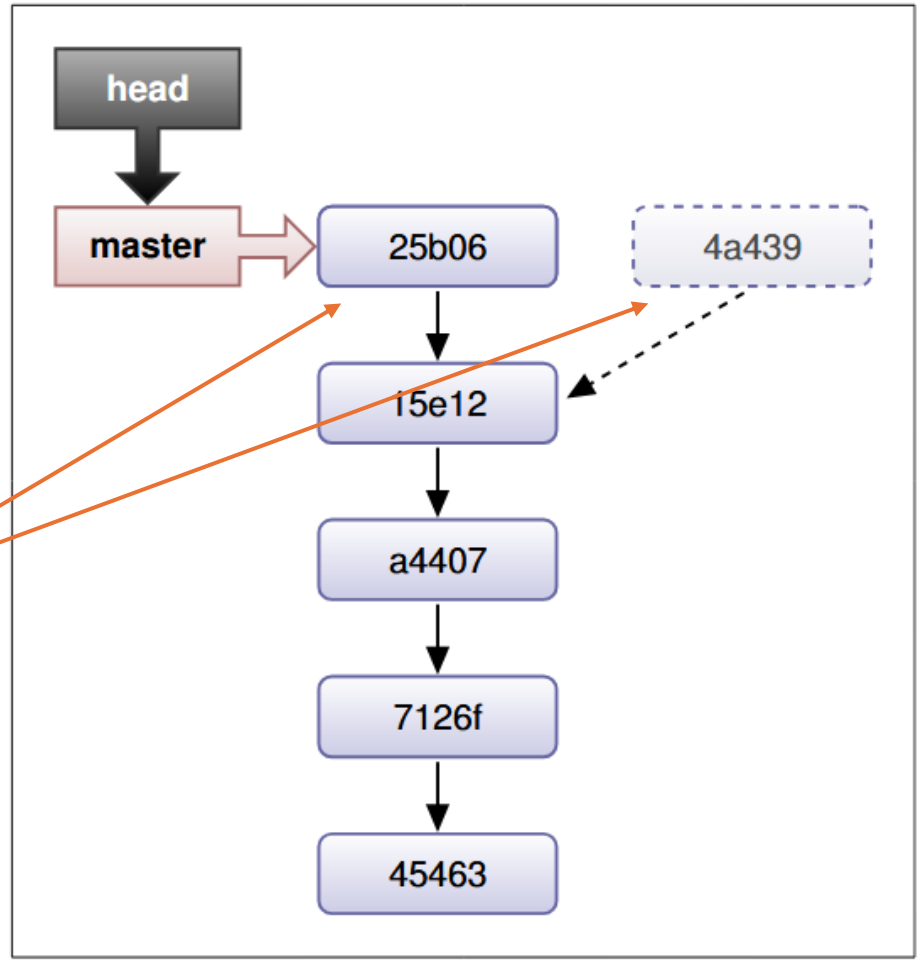


Ajout du fichier bar.txt

Branches – Amend modification du dernier commit

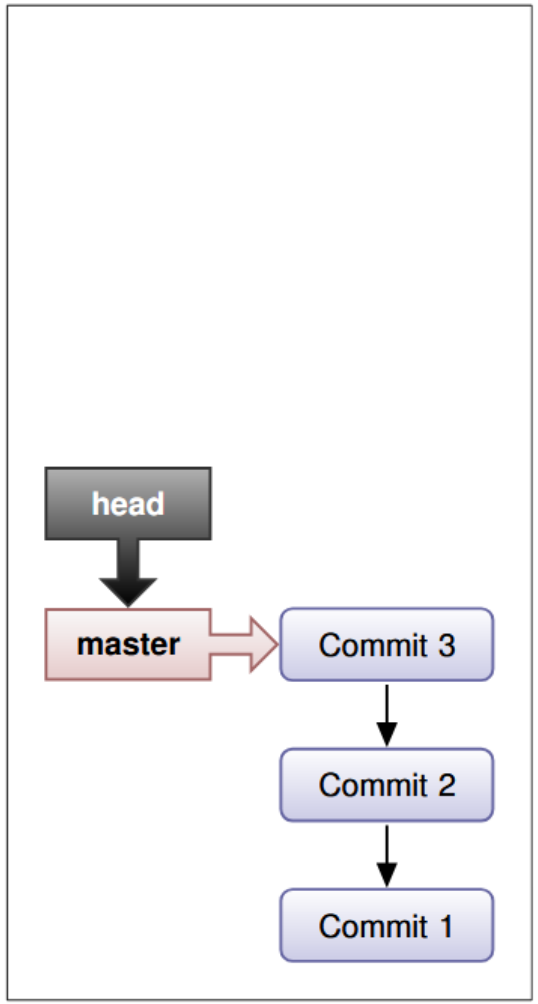
```
ls
 foo.txt dir

touch bar.txt
git commit -m "Ajou d'un fichier."

git add bar.txt

git commit --amend -m "Ajout d'un
fichier."
```

Modification du fichier bar.txt



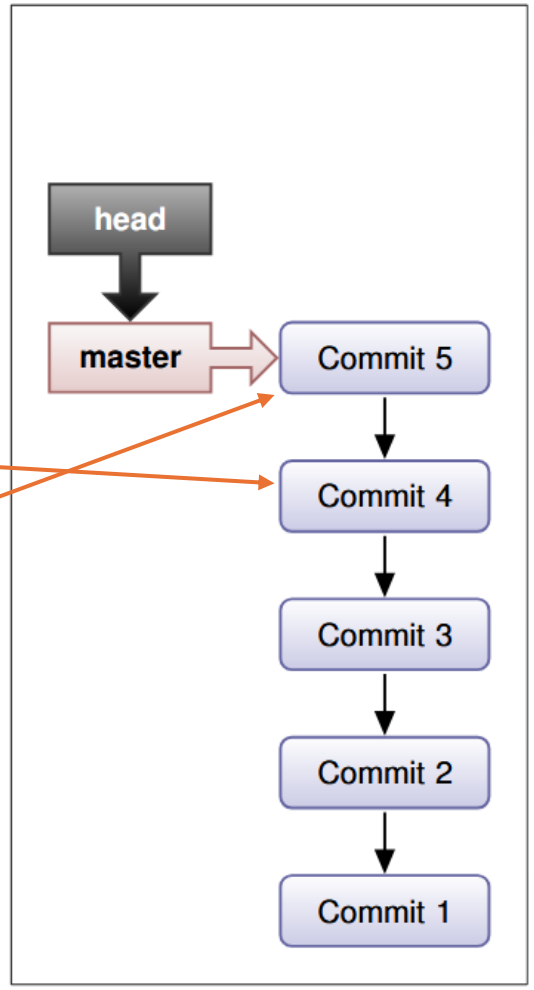
Branches – Git revert annulation par commit

```
git branch master
cat fichier1.txt
 Premiere version de F1
cat fichier2.txt
 Premiere version de F2
```



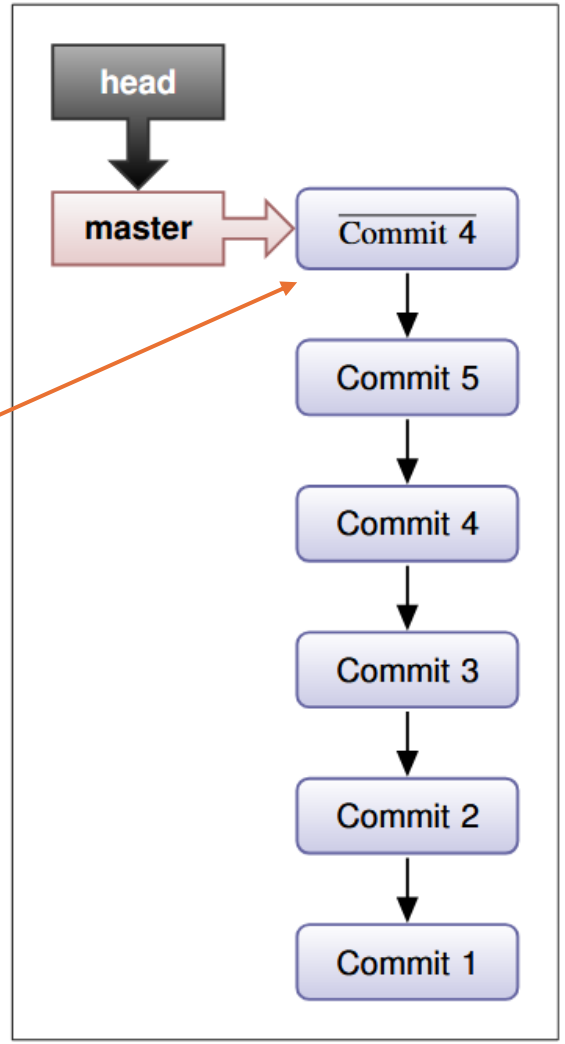
Branches – Git revert annulation par commit

```
git branch master
cat fichier1.txt
 Premiere version de F1
cat fichier2.txt
 Premiere version de F2

echo "Deuxieme version de F1" > fichier1.txt
git add fichier1.txt
git commit -m "Add fichier1.txt"

echo "Deuxieme version de F2" > fichier2.txt
git add fichier2.txt
git commit -m "Add fichier2.txt"
```



Branches – Git revert annulation par commit

```
git branch master
cat fichier1.txt
 Premiere version de F1
cat fichier2.txt
 Premiere version de F2

echo "Deuxieme version de F1" > fichier1.txt
git add fichier1.txt
git commit -m "Add fichier1.txt"

echo "Deuxieme version de F2" > fichier2.txt
git add fichier2.txt
git commit -m "Add fichier2.txt"

git revert HEAD^
cat fichier1.txt
 Premiere version de F1
cat fichier2.txt
 Deuxieme version de F2
```



Branches – Git Reset

- Annuler des ajouts dans l'index

  $git reset monfichier

- Restaurer un ancien commit (mais en conservant toutes les modifications des fichiers et l'index)

  $git reset --soft commitID

- Restaurer un ancien commit et l'index (mais en conservant toutes les modifications des fichiers)

  $git reset commitID

- Restaurer un ancien commit, l'index, et le contenu des fichiers correspondants

  $git reset --hard commitID

Plan du cours

I. Introduction

II. GIT

III. Utilisation de GIT

IV. Les bonnes pratiques

V. Synchronisation avec les dépôts distants

Bonnes pratiques

- Ne pas versionner de fichiers générés automatiquement (logs, pdf, exécutables, etc.) ou personnels
- Faire de petits commits réguliers et facile à intégrer, leur donner un nom explicite
- Utiliser les branches pour :
  - les développements à plusieurs
  - chaque développement conséquent d'une nouvelle fonctionnalité
- Ne pas développer sur la branche master à plusieurs pour éviter les conflits lors des pull
- Faire de petits commits locaux, et pusher des commits plus conséquents, toujours testés et fonctionnels !
- Faire des pull régulièrement

Bonnes pratiques .gitignore

- gitignore spécifie les fichiers non versionnés que git doit ignorer:
  - Les fichiers déjà versionnés ne sont pas affectés.
  - Définition à l'aide de patterns
  - Des fichiers .gitignore peuvent être placés n'importe où dans la hiérarchie.
    - Les règles des fichiers plus bas dans la hiérarchie se substituent aux règles définies plus haut.
    - Priorités: Même répertoire, parent, . . . , racine du dépôt
  - Ces fichiers .gitignore sont à ajouter au dépôt
- $GIT_DIR/info/exclude
  - Règles spécifiques à un utilisateur
- Fichier spécifié par l'option de configuration core.excludesFile dans (homedir)/.gitconfig

Bonnes pratiques .gitignore

- Définir des règles

```
# a comment - this is ignored
# no .a files
*.a
# but do track lib.a, even though you're ignoring .a files above
!lib.a
# only ignore the root TODO file, not subdir/TODO
/TODO
# ignore all files in the build/ directory
build/
# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt
```

Bonnes pratiques: Message de commit

- Messages de commit

Le plus important: Décrire quoi et pourquoi et pas comment

- Ne pas décrire les modifications qui sont faites (informations disponibles avec un diff)
- Décrire les fonctionnalités ajoutées

Exemple

- Bad: Modifie la fonction f pour tester la variable a
- Good: Vérifie les droits de l'utilisateur avant d'exécuter l'action X

Bonnes pratiques

Consulter l'historique des commits

- Affiche l'historique des commits en remontant à partir de commitID.

```
$git log commitID
```

- Par défaut, commitID est HEAD

Bonnes pratiques

Consulter des changements

- Afficher les détails sur un commit:

  $git show commitID

  $git show commitID -- monfichier monrepertoire

- Afficher les différences entre des versions:

  $git diff commitID1..commitID2 -- monfichier monrepertoire

- Savoir qui a modifié un fichier

  $git blame file.txt

  $git blame L80,+20 file.txt

Bonnes pratiques

Supprimer des fichiers

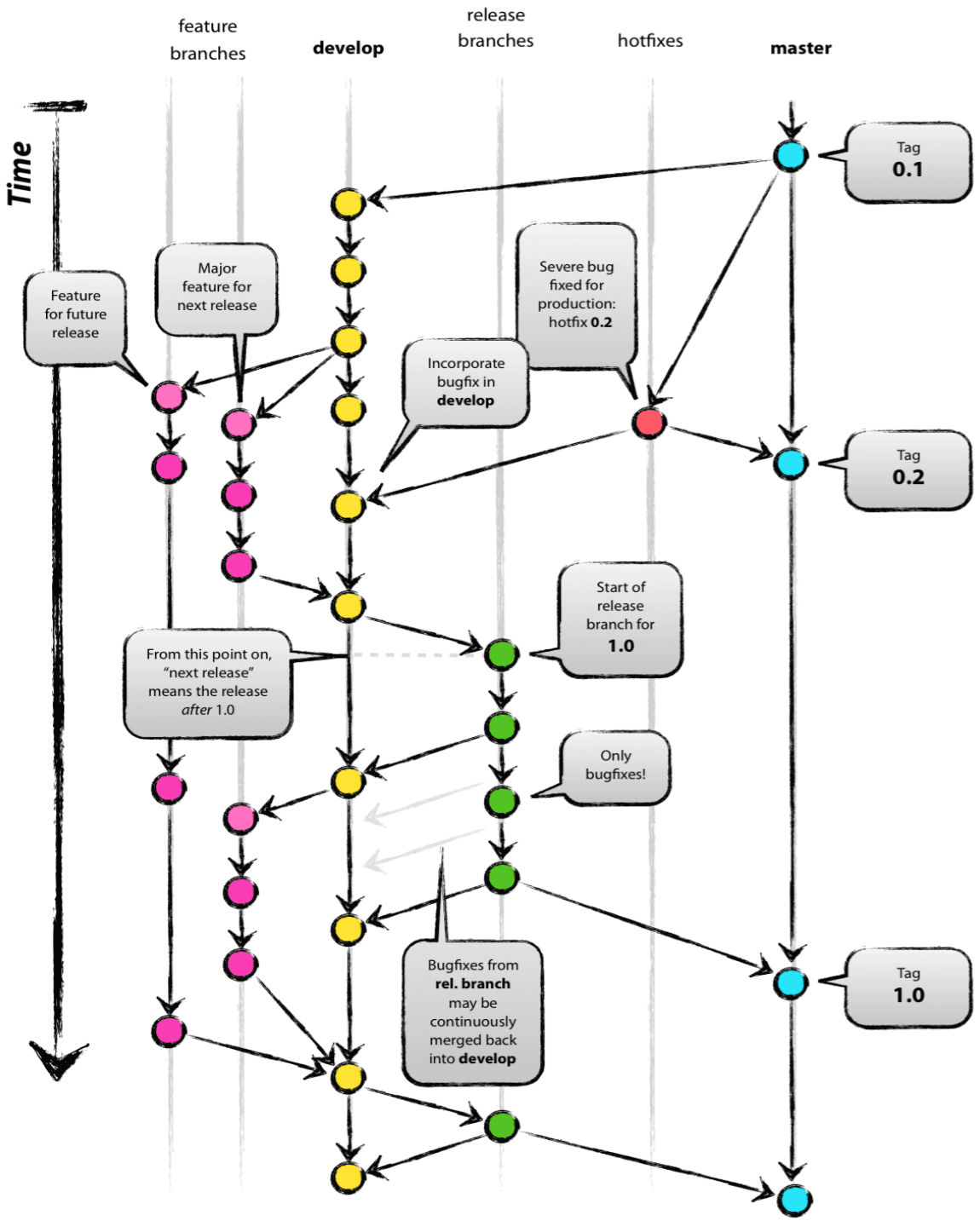
- Pour supprimer un fichier versionné:

  $git rm monfichier

Pourquoi le faire ainsi?

- On peut utiliser la commande unix rm
  - Requiert d'ajouter ensuite la modification à l'index (git add)
- La commande $git rm fait tout pour nous
  - Supprime le fichier du répertoire de travail
  - Met à jour l'index

Organisation du Travail - GitFlow



Plan du cours

I. Introduction

II. GIT

III. Utilisation de GIT

IV. Les bonnes pratiques

V. Synchronisation avec les dépôts distants

Centralisé vs Distribué

Modèle centralisé

- Un serveur gère l'intégralité des version (le dépôt)
- Les utilisateurs y ajoutent leurs modifications
- Les utilisateurs y récupèrent les modifications des autres

Modèle distribué

- Chaque utilisateur possède un dépôt entier
- Les dépôts peuvent s'échanger des modifications

Centralisé vs Distribué

Modèle distribué

- Chaque client a l'ensemble des fichiers dans son dépôt local
  - Travailler off-line
  - Changer de branche est rapide. On peut abuser des branches.
- Actions nécessitant un accès au dépôt distant:
  - Mise à jour du dépôt local depuis l'extérieur
  - L'envoi d'informations
- Le client peut versionner en local.

Branches distantes et branches locales

Un projet décentralisé possède deux types de branches :

- Définition

On appelle branche distante, une branche qui pointe sur des dépôts distants en lecture et/ou écriture. Ces dépôts distants peuvent être référencés par une ou plusieurs personnes.

- Définition

On appelle branche locale, une branche propre au dépôt local. Pour être envoyées, les données d'une telle branche doivent être fusionnées avec une branche distante.

Fetch, Merge et Pull

- fetch : Importe les commits d'un dépôt distant dans le dépôt local.
  - Utiliser merge pour importer les changements dans une branche locale
- pull : Fusionne les changements d'un dépôt distant directement dans une branche locale.
  - Equivalent d'un fetch suivi d'un merge.
  - Peut être configuré pour utiliser rebase au lieu de merge
- Fetch permet d'observer les changements avant de les intégrer dans sa branche de travail

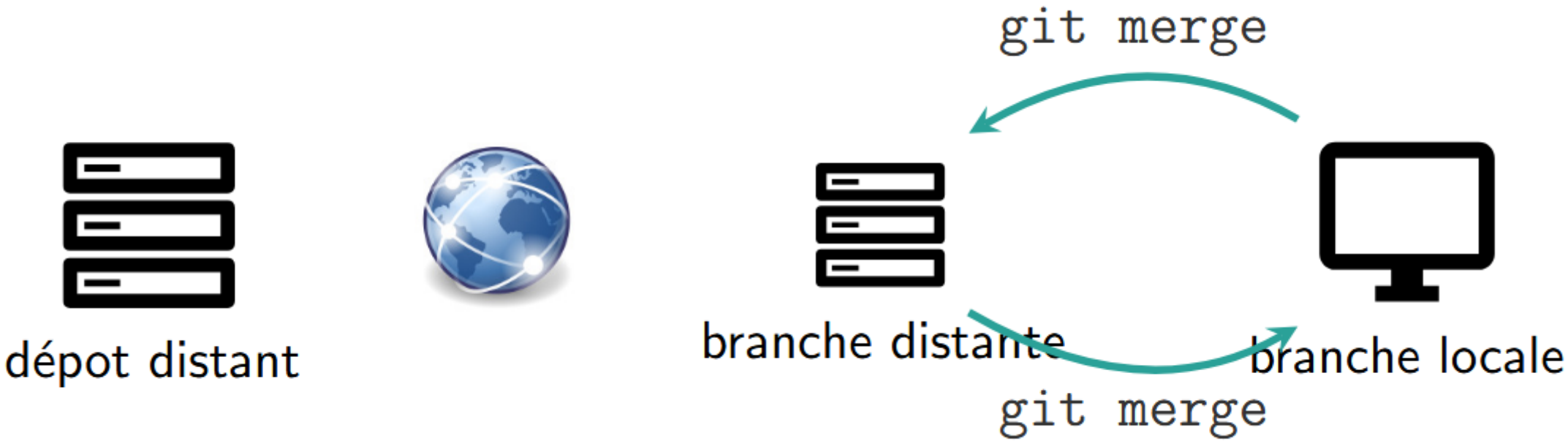
Modèle distribué

dépot distant

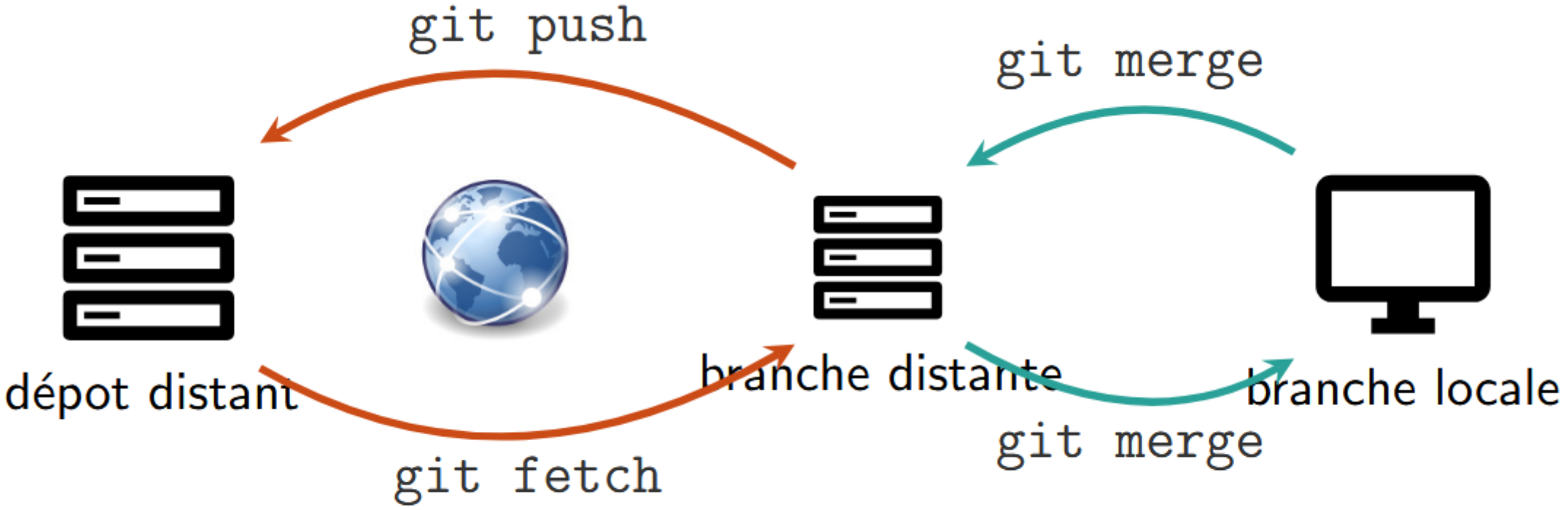
branche distante

branche locale

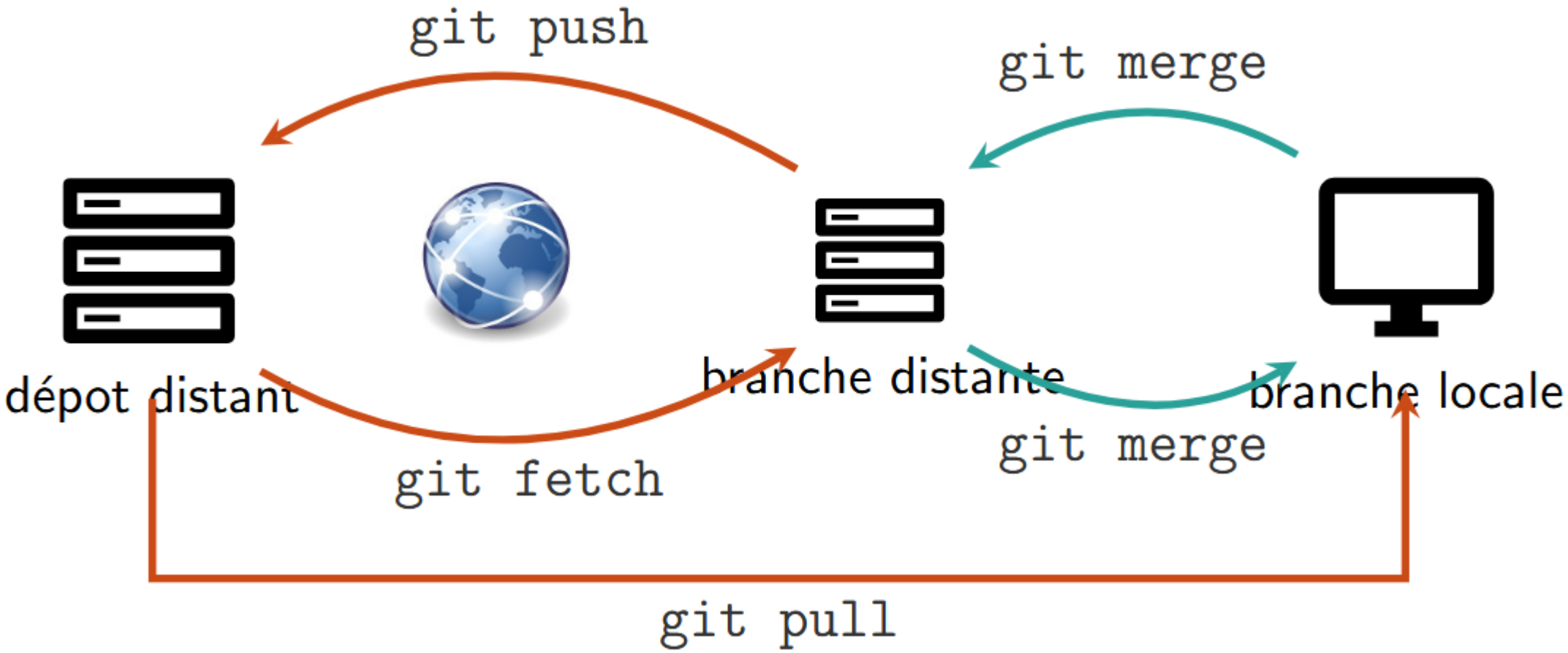
Modèle distribué



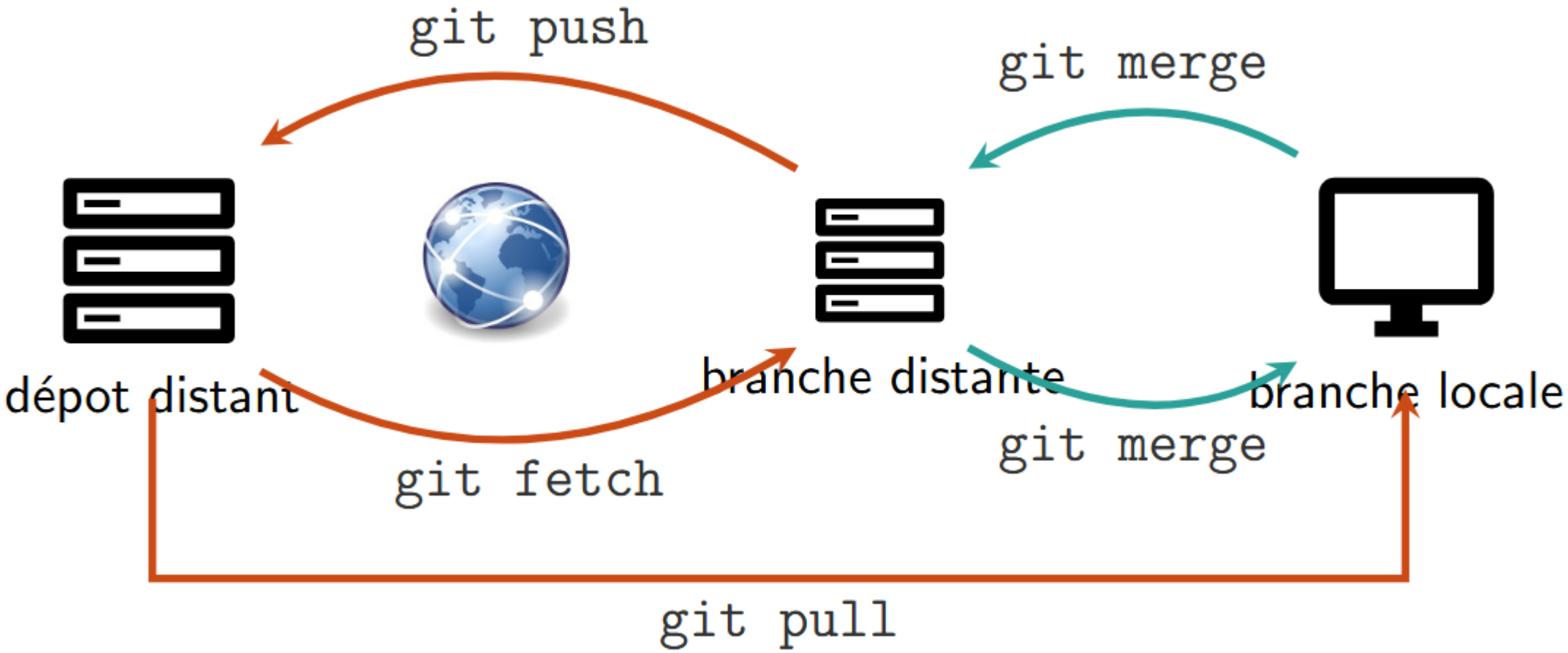
Modèle distribué



Modèle distribué



Modèle distribué



Merci