

NMV - MU5IN453 : Git, un gestionnaire de versions décentralisé.

Version 20.02

Julien Sopena¹

¹julien.sopena@lip6.fr
Équipe Delys - INRIA Paris
LIP6 - Sorbonne Université / CNRS

Master SAR 2^{ème} année - 2022/2023

Grandes lignes du cours

Git c'est quoi ?

Architecture interne de git

Création d'un dépôt

Les objets manipulé par git

Les "Blobs" (fichier)

Les "Trees"

Les "Commits"

Les "Tags"

Structure générale

Gestion des versions dans le dépôt local.

Les commits

Les branches

Les merges

Les rebases

Les remords

Utilisation de l'historique

Synchronisation avec les dépôts distants.

Principe des DVCS :

Gestionnaire de versions décentralisé.

Les dépôts distants.

Modèles de travail coopératif

Les outils graphiques

git-gui et gitk

Exemple de gitg

Conclusion

Outline

Git c'est quoi ?

Architecture interne de git

Gestion des versions dans le dépôt local.

Synchronisation avec les dépôts distants.

Les outils graphiques

Conclusion

Principe de base des gestionnaires de version

Un gestionnaire de versions (**VCS**) doit permettre de :

- ▶ conserver toutes les versions de tous les fichiers ;
- ▶ conserver toutes les arborescences de fichiers ;
- ▶ permettre d'identifier une arborescence de versions de fichiers ;
- ▶ fournir des outils pour gérer le tout.

Un **DVCS** ("*Distributed Version Control*") offre les mêmes services sur une architecture décentralisée.

Tous deux reposent sur deux mécanismes :

1. un mécanisme permettant de calculer la différence entre deux versions : **diff/patch** ;
2. un gestionnaire d'**historique** des diff.

diff & patch

diff : Comparaison de fichiers ligne par ligne

- ▶ indique les lignes ajoutées ou supprimées ;
- ▶ peut ignorer les casses, les tabulations, les espaces ;
- ▶ option **-u** pour créer des patchs unifiés, avec plus d'informations.

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

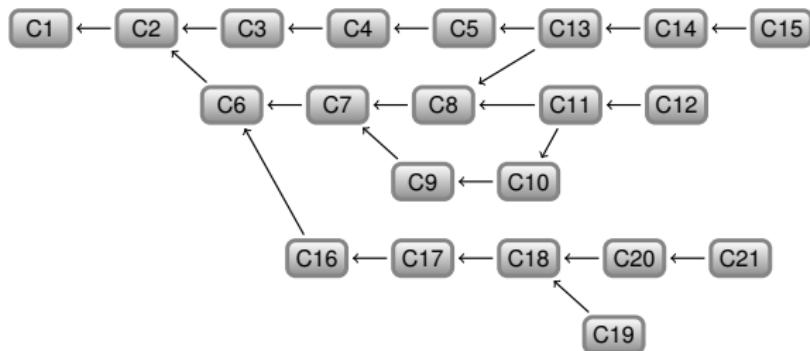
Exemple

```
$ diff toto.c toto-orig.c > correction.patch  
$ bzip2 correction.patch  
$ bzcat correction.patch.bz2 | patch -p 0 toto.c
```

La notion d'historique

Definition

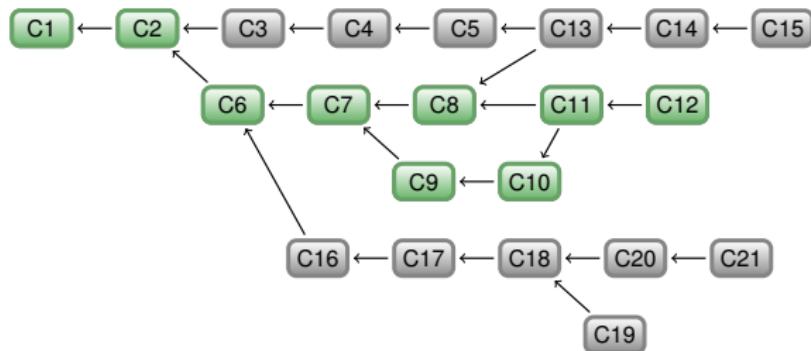
On appelle **historique** un graphe orienté acyclique composé d'un ensemble de versions pouvant être recalculées à partir des versions adjacentes en appliquant les patchs modélisés par les arcs sortants.



Historique : Les branches

Definition

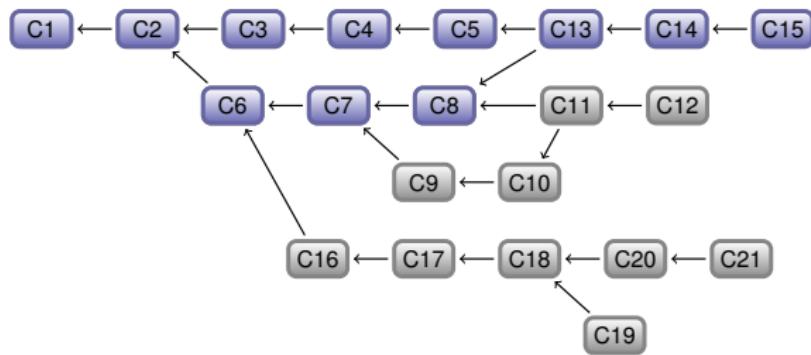
La **branche** de la version v_i d'un historique est le sous-graphe composé de l'ensemble des versions accessibles depuis v_i dans le graphe de l'historique.



Historique : Le tronc

Definition

Le **tronc** ou **branche principale** de l'historique est la branche issue de la dernière version stable.



#balanceTonDepot

Depuis le 1er octobre 2020, la branche principale des dépôts créés sur GitHub est nommée **main** au lieu de **master**...

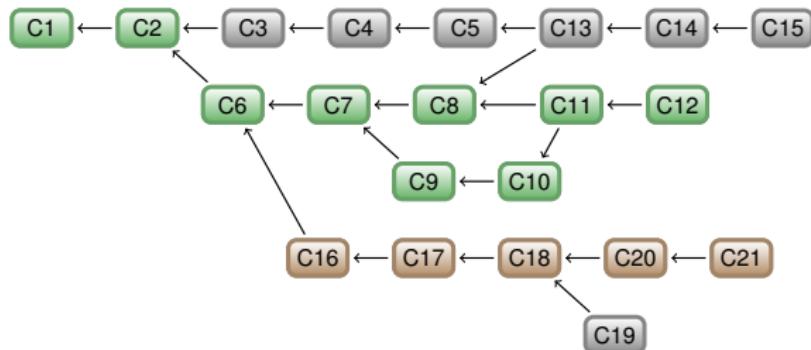
Ce changement s'inscrit dans un mouvement visant à "*supprimer les références inutiles à l'esclavage et les remplacer par des termes plus inclusifs*".

```
$ git init
astuce: Utilisation de 'master' comme nom de la branche initiale.
astuce: Pour configurer le nom de la branche initiale pour tous les
astuce: nouveaux dépôts, et supprimer cet avertissement, lancez :
astuce:
astuce: git config --global init.defaultBranch <nom>
astuce:
astuce: Les noms les plus utilisés à la place de 'master' sont
astuce: 'main', 'trunk' et 'development'. La branche nouvellement
astuce: créée peut être renommée avec :
astuce:
astuce: git branch -m <nom>
Dépôt Git vide initialisé dans /tmp/depot
```

Historique : Sous branches

Definition

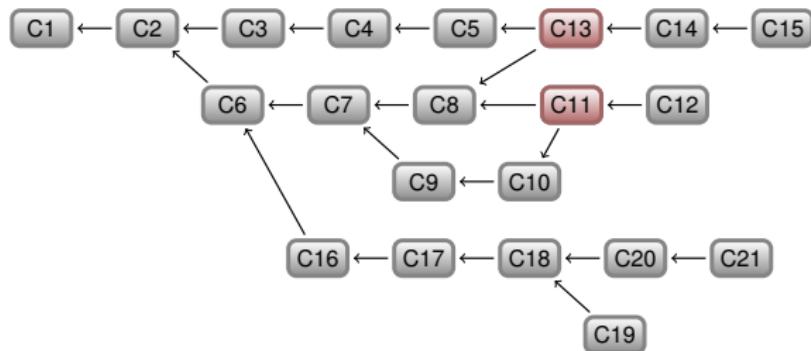
Une **sous-branche** SB_1 d'une branche B_2 est le sous graphe composé de l'ensemble des versions d'une branche B_1 n'appartenant pas à la branche B_2 , l'intersection de B_1 et de B_2 devant être non vide.



Historique : Les merges.

Definition

On appelle **merge** toute version ayant un degré sortant strictement supérieur à 1. Cette version correspond alors à la fusion des **patches** de plusieurs branches.



Historique

2001 : Linux est développé sur **CVS**.

2002 à 2005 : Linux est développé sur **Bitkeeper** :

- ▶ *Bitkeeper* est décentralisé.
- ▶ Plus *Bitkeeper* progresse et plus le développement de Linux devient efficace.

6 avril 2005 : *Bitkeeper* quitte le libre :

- ▶ Création de Git par Linus Thorvalds.

18 avril 2005 : Git peut faire un *Merge*

16 juin 2005 : Linux est développé officiellement sur **Git**.

14 février 2007 : Sortie de la version 1.5.0

- ▶ Git devient vraiment utilisable par tous !

Outline

Git c'est quoi ?

Architecture interne de git

Création d'un dépôt

Les objets manipulé par git

Les "Blobs" (fichier)

Les "Trees"

Les "Commits"

Les "Tags"

Structure générale

Gestion des versions dans le dépôt local.

Synchronisation avec les dépôts distants.

Les outils graphiques

Conclusion

Outline

Git c'est quoi ?

Architecture interne de git

Création d'un dépôt

Les objets manipulé par git

Les "Blobs" (fichier)

Les "Trees"

Les "Commits"

Les "Tags"

Structure générale

Gestion des versions dans le dépôt local.

Synchronisation avec les dépôts distants.

Les outils graphiques

Conclusion

Création d'un nouveau dépôt

La création d'un dépôt se fait avec la commande **git init**, toutes les données seront stockées dans un répertoire **.git/** à la racine du projet.

```
$ cd monProjet
$ ls -a
  Makefile README main.c
$ git init
  defaulting to local storage area
$ ls -a
  .git/ Makefile README main.c
$ ls -a .git/
  HEAD branches/ config description hooks/ info/ objects/ refs/
```

Pour arrêter la gestion de version et détruire le dépôt, *i.e.*, supprimer ses données et ses métadonnées, il suffit d'effacer le répertoire **git**.

```
$ rm -rf .git/
$ ls -a
  Makefile README main.c
```

Création d'un nouveau dépôt "serveur"

Un dépôt sur un serveur n'est là que pour la gestion des versions :
il n'a donc besoin que de l'historique.

Y maintenir une version des fichiers est une source de problèmes :
on ne peut pas push dans un dépôt qui contient des fichiers modifiés

Il est donc préférable d'utiliser l'option **--bare** :

```
git init --bare ou git clone --bare <URL>
```

Ce dépôt n'a :

- ▶ pas de fichier juste l'historique
- ▶ pas de répertoire xxx/.git mais directement un xxx.git/
- ▶ pas de "remote" (origin)

```
$ git clone --bare <URL>
$ ls monProjet.git/
  HEAD branches/ config description hooks/ info/ objects/ refs/
$ cat monProjet.git/config
[core]
  bare = true
```

Outline

Git c'est quoi ?

Architecture interne de git

Création d'un dépôt

Les objets manipulé par git

Les "Blobs" (fichier)

Les "Trees"

Les "Commits"

Les "Tags"

Structure générale

Gestion des versions dans le dépôt local.

Synchronisation avec les dépôts distants.

Les outils graphiques

Conclusion

Les objets : tout est Blob

Git a été conçu comme un : **système de fichiers versionnés**.

Linus Torvalds a repris les concepts de Linux :

- ▶ "tout est fichier"
→ "tout est blob"
- ▶ "les fichiers sont identifiés par leur numéro d'inode"
→ "les blobs sont identifiés par le **SHA-1** de leur contenu"

L'identification par *hash* n'est possible que si le contenu est fixe :

⇒ les blobs sont des objets immuables"

Cela réduit naturellement la taille des dépôts :

⇒ la déduplication niveau fichier est gratuite

Secure Hash Algorithm : SHA-1

SHA-1 est une fonction de hachage cryptographique :

- ▶ elle a été conçue par la NSA
- ▶ elle est "limitée" à des fichiers de 2^{64} bits (soit 2 exaoctets)
- ▶ elle retourne un hash de 160 bits
- ▶ son résultat se note avec 40 caractères en notation hexadécimale

```
$ echo "une petite phrase" > unFichier
$ shasum unFichier
  a3bdd9c7cb436cf85a3b68ac3a653f3e9c8a40c unFichier
$ echo "Une petite phrase" > unFichier
$ shasum unFichier
  e62bea749297a6f7ad5cb5744b7946dcc9036f20 unFichier
```

On peut vérifier l'intégrité d'un dépôt en recalculant tous les SHA-1

```
$ git fsck
Vérification des répertoires d'objet: 100% (256/256), fait.
```

Conséquence d'une collision de hash

Quelle que soit la fonction de hash les collisions sont inévitables :

- ▶ avec SHA-1, il y a fichiers en collision sur chaque hash

Conséquence d'une collision de hash

Quelle que soit la fonction de hash les collisions sont inévitables :

- ▶ avec SHA-1, il y a $2^{2^{64}-160}$ fichiers en collision sur chaque hash

Conséquence d'une collision de hash

Quelle que soit la fonction de hash les collisions sont inévitables :

- ▶ avec SHA-1, il y a $2^{2^{64}-160}$ fichiers en collision sur chaque hash

Mais dans le cas de SHA1 c'est très improbable que cela arrive :

- ▶ deux hash aléatoires ont 1 chance sur d'être en collision

Conséquence d'une collision de hash

Quelle que soit la fonction de hash les collisions sont inévitables :

- ▶ avec SHA-1, il y a $2^{2^{64}-160}$ fichiers en collision sur chaque hash

Mais dans le cas de SHA1 c'est très improbable que cela arrive :

- ▶ deux hash aléatoires ont 1 chance sur 2^{160} d'être en collision

Conséquence d'une collision de hash

Quelle que soit la fonction de hash les collisions sont inévitables :

- ▶ avec SHA-1, il y a $2^{2^{64}-160}$ fichiers en collision sur chaque hash

Mais dans le cas de SHA1 c'est très improbable que cela arrive :

- ▶ deux hash aléatoires ont 1 chance sur 2^{160} d'être en collision

Dans la réflexion sur le choix de la fonction de hachage, il faut distinguer :

- ▶ les conséquences d'une collision résultant de la malchance
- ▶ les risques liés à une attaque basée sur une collision de hash

Conséquence d'une collision de hash

Quelle que soit la fonction de hash les collisions sont inévitables :

- ▶ avec SHA-1, il y a $2^{2^{64}-160}$ fichiers en collision sur chaque hash

Mais dans le cas de SHA1 c'est très improbable que cela arrive :

- ▶ deux hash aléatoires ont 1 chance sur 2^{160} d'être en collision

Dans la réflexion sur le choix de la fonction de hachage, il faut distinguer :

- ▶ les conséquences d'une collision résultant de la malchance
- ▶ les risques liés à une attaque basée sur une collision de hash

Nous reviendrons sur cela une fois tous les mécanismes bien étudiés :-)

Les 4 types d'objets

Definition

Le **Blob** est l'élément de base pour le stockage des données. Il est :

- ▶ identifié par le SHA-1 de son contenu noté en hexadécimal :
SHA-1 (blob) = XXYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
- ▶ stocké dans le dépôt sous forme d'un fichier :
.git/objects/XX/YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
- ▶ formé de "**<type> <taille du contenu>\0<contenu>**"
- ▶ compressé à l'aide de la bibliothèque **zlib**

Git utilise quatre types d'objets :

Blob : stocke le contenu des fichiers (*ambiguïté du Blob*)

Tree : stocke l'arborescence

Commit : stocke les versions du dépôt

Tag : identifie certaines versions du dépôt

Outline

Git c'est quoi ?

Architecture interne de git

Création d'un dépôt

Les objets manipulé par git

Les "Blobs" (fichier)

Les "Trees"

Les "Commits"

Les "Tags"

Structure générale

Gestion des versions dans le dépôt local.

Synchronisation avec les dépôts distants.

Les outils graphiques

Conclusion

Les "Blobs" (fichiers)

Definition

On appelle aussi **Blob**, un blob qui stocke le contenu d'un fichier :

- ▶ ils sont composés de :
"blob <taille du contenu>\0<contenu>"
- ▶ à chaque révision du fichier correspond un nouveau *Blob*.
- ▶ le *Blob* ne dépend pas du nom ou de l'emplacement :
 - ▶ Si un fichier est renommé, pas de nouveau *Blob*
 - ▶ Si un fichier est déplacé, pas de nouveau *Blob*

Les "Blobs" (fichiers) : exemple

33f3a..

Blob	Size
Mouches d'automne Comme il est facile De vous écrabouiller! <i>Shiki (1866 - 1902)</i>	

a35e1..

Blob	Size
Mante religieuse Piteuse escaladeuse De ce melon <i>Shiki (1866 - 1902)</i>	

Les "Blobs" sont compressés avec la zlib

```
$ cat foo.txt
Mouches d'automne
Comme il est facile
De vous écrabouiller !

Shiki ( 1866 - 1902 )
```

Les "Blobs" sont compressés avec la zlib

```
$ cat foo.txt  
Mouches d'automne  
Comme il est facile  
De vous écrabouiller !
```

Shiki (1866 - 1902)

```
$ git cat-file -p 6ded47ba437cb9cc986e47bafc146c75e50be03c  
Mouches d'automne  
Comme il est facile  
De vous écrabouiller !
```

Shiki (1866 - 1902)

Les "Blobs" sont compressés avec la zlib

```
$ cat foo.txt  
Mouches d'automne  
Comme il est facile  
De vous écrabouiller !
```

Shiki (1866 - 1902)

```
$ git cat-file -p 6ded47ba437cb9cc986e47bafc146c75e50be03c  
Mouches d'automne  
Comme il est facile  
De vous écrabouiller !
```

Shiki (1866 - 1902)

```
$ cat ..git/objects/6d/ed47ba437cb9cc986e47bafc146c75e50be03c  
xKOR0e/MH-VHQO,-KrMUQH-.QHKLIrIU(/-V82(1)43'''HA+8#3;SACLAWAHA
```

Les "Blobs" sont compressés avec la zlib

```
$ cat foo.txt  
Mouches d'automne  
Comme il est facile  
De vous écrabouiller !
```

Shiki (1866 - 1902)

```
$ git cat-file -p 6ded47ba437cb9cc986e47bafc146c75e50be03c  
Mouches d'automne  
Comme il est facile  
De vous écrabouiller !
```

Shiki (1866 - 1902)

```
$ cat ..git/objects/6d/ed47ba437cb9cc986e47bafc146c75e50be03c  
xKOR0e/MH-VHQO,-KrMUQH-.QHKLIrIU(/-V82(1)43'''HA+8#3;SACLAWHA
```

```
$ pigz --decompress --zlib < .git/objects/6d/ed47ba437cb9cc9*  
blob 85Mouches d'automne  
Comme il est facile  
De vous écrabouiller !
```

Shiki (1866 - 1902)

Le SHA-1 est calculé sur le blob pas sur le contenu

```
$ shasum foo.txt  
df8ac3abf89ac0545a35783ca23c138408a0c476 foo.txt
```

Le SHA-1 est calculé sur le blob pas sur le contenu

```
$ shasum foo.txt  
df8ac3abf89ac0545a35783ca23c138408a0c476 foo.txt  
  
$ ls .git/objects/df/*  
/usr/bin/ls: impossible d'accéder à '.git/objects/df/*'
```

Le SHA-1 est calculé sur le blob pas sur le contenu

```
$ shasum foo.txt  
df8ac3abf89ac0545a35783ca23c138408a0c476 foo.txt  
  
$ ls .git/objects/df/*  
/usr/bin/ls: impossible d'accéder à '.git/objects/df/*'  
  
$ echo -en "blob $(wc -c < foo.txt)\0" | cat - foo.txt | shasum  
6ded47ba437cb9cc986e47bafc146c75e50be03c -
```

Le SHA-1 est calculé sur le blob pas sur le contenu

```
$ shasum foo.txt  
df8ac3abf89ac0545a35783ca23c138408a0c476 foo.txt  
  
$ ls .git/objects/df/*  
/usr/bin/ls: impossible d'accéder à '.git/objects/df/*'  
  
$ echo -en "blob $(wc -c < foo.txt)\0" | cat - foo.txt | shasum  
6ded47ba437cb9cc986e47bafc146c75e50be03c -  
  
$ ls .git/objects/6d/*  
.git/objects/6d/ed47ba437cb9cc986e47bafc146c75e50be03c
```

Le SHA-1 est calculé sur le blob pas sur le contenu

```
$ shasum foo.txt
df8ac3abf89ac0545a35783ca23c138408a0c476 foo.txt

$ ls .git/objects/df/*
/usr/bin/ls: impossible d'accéder à '.git/objects/df/*'

$ echo -en "blob $(wc -c < foo.txt)\0" | cat - foo.txt | shasum
6ded47ba437cb9cc986e47bafc146c75e50be03c -

$ ls .git/objects/6d/*
.git/objects/6d/ed47ba437cb9cc986e47bafc146c75e50be03c

$ git hash-object foo.txt
6ded47ba437cb9cc986e47bafc146c75e50be03c
```

Outline

Git c'est quoi ?

Architecture interne de git

Création d'un dépôt

Les objets manipulé par git

Les "Blobs" (fichier)

Les "Trees"

Les "Commits"

Les "Tags"

Structure générale

Gestion des versions dans le dépôt local.

Synchronisation avec les dépôts distants.

Les outils graphiques

Conclusion

Les "Trees" : définitions

Definition

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 (resp. repertoire) à chacun des pointeurs de **Blobs** (resp. **Trees**).
- ▶ Un ensemble de **Tree** permet de décrire l'état d'une hiérarchie de dossiers à un moment donné.

Les "Tree" : exemple

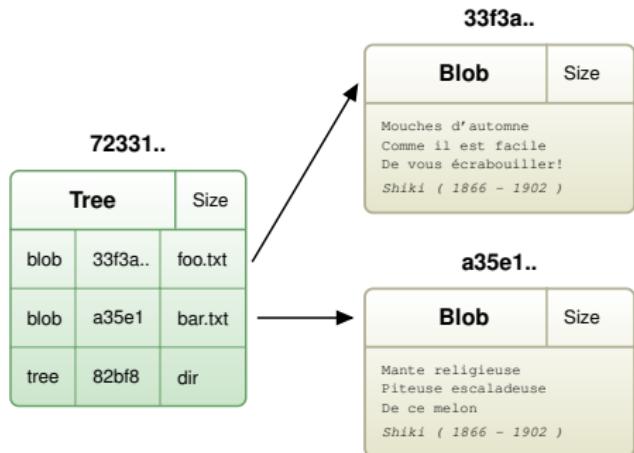
33f3a..

Blob	Size
Mouches d'automne Comme il est facile De vous écrabouiller! <i>Shiki (1866 - 1902)</i>	

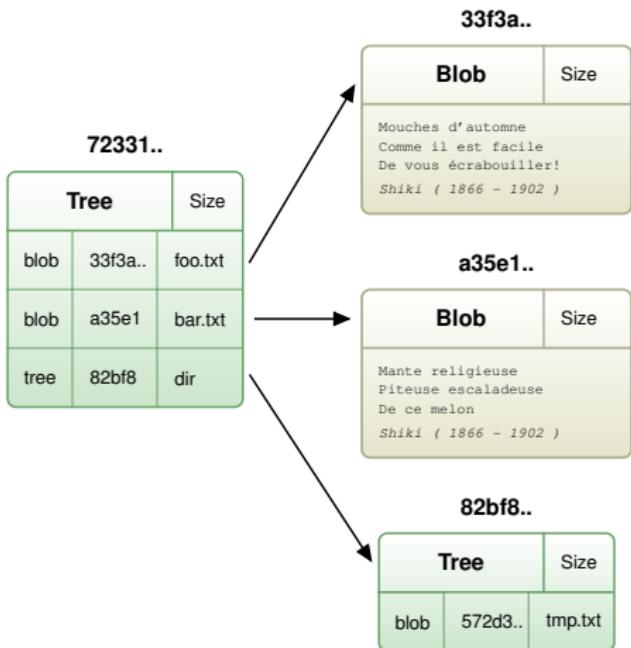
a35e1..

Blob	Size
Mante religieuse Piteuse escaladeuse De ce melon <i>Shiki (1866 - 1902)</i>	

Les "Tree" : exemple



Les "Tree" : exemple



Stockage de la hiérarchie

```
$ ls -R
.:
bar.txt dir/ foo.txt

./dir:
tmp.txt
```

Stockage de la hierarchie

```
$ ls -R
.:
bar.txt dir/ foo.txt

./dir:
tmp.txt

$ git cat-file -p e7f3b24b01115fd0ea58d72099a24af383f627ba
100644 blob b13fb47577e28a3069183f9ad91fb0ee44d3871f bar.txt
040000 tree 96a31457a25444a576b1fb0b8eafee735a706693 dir
100644 blob 6ded47ba437cb9cc986e47bafc146c75e50be03c foo.txt
```

Stockage de la hierarchie

```
$ ls -R
.:
bar.txt dir/ foo.txt

./dir:
tmp.txt

$ git cat-file -p e7f3b24b01115fd0ea58d72099a24af383f627ba
100644 blob b13fb47577e28a3069183f9ad91fb0ee44d3871f bar.txt
040000 tree 96a31457a25444a576b1fb0b8eafee735a706693 dir
100644 blob 6ded47ba437cb9cc986e47bafc146c75e50be03c foo.txt

$ git cat-file -p 96a31
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391 tmp.txt
```

Outline

Git c'est quoi ?

Architecture interne de git

Création d'un dépôt

Les objets manipulé par git

Les "Blobs" (fichier)

Les "Trees"

Les "Commits"

Les "Tags"

Structure générale

Gestion des versions dans le dépôt local.

Synchronisation avec les dépôts distants.

Les outils graphiques

Conclusion

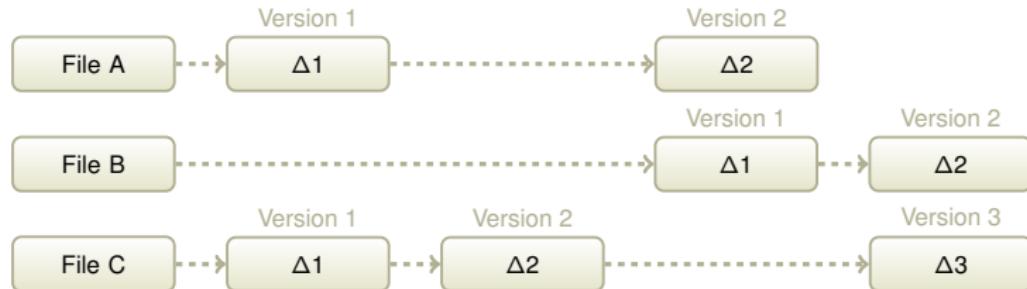
Commit : définition.

Definition

Committer un fichier signifie : enregistrer une version de ce dernier (le plus souvent la version actuelle, mais pas toujours) dans un gestionnaire de versions. Par extension, si le fichier est déjà versionné, on dit que l'on **commit** une modification du fichier.

La numérotation des commits dans CVS.

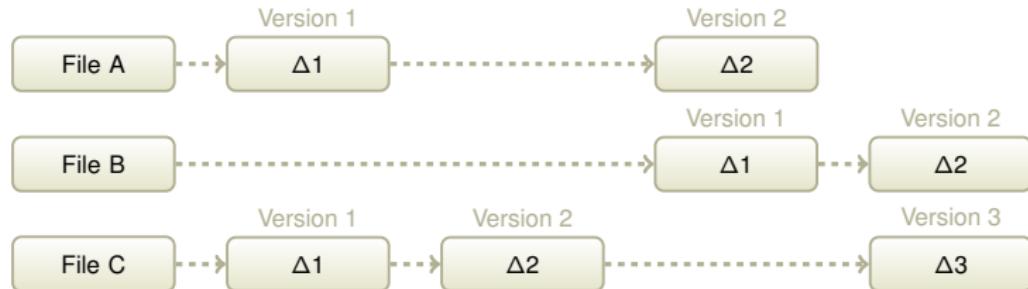
CVS utilise une numérotation sur des patchs, fichier par fichier :



La numérotation des commits dans CVS.

CVS utilise une numérotation sur des patchs, fichier par fichier :

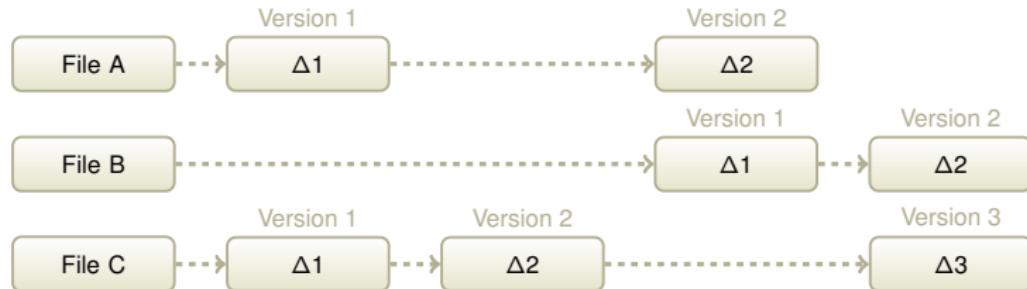
=> difficile de trouver un état cohérent du système (ajout de tag).



La numérotation des commits dans CVS.

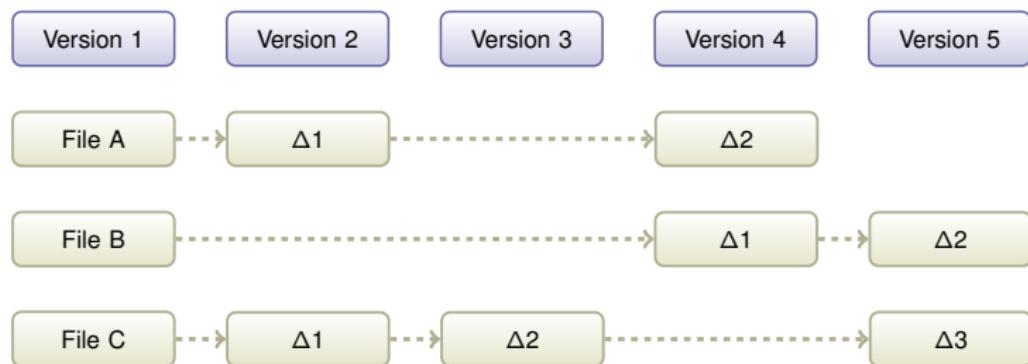
CVS utilise une numérotation sur des patchs, fichier par fichier :

- => difficile de trouver un état cohérent du système (ajout de tag).
- => l'accès à une version nécessite de ré-appliquer les patchs.



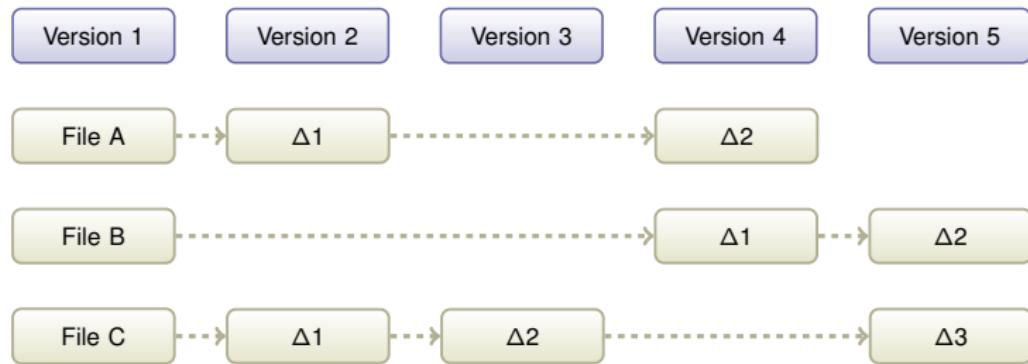
La numérotation des commits dans SVN.

SVN utilise une numérotation globale sur l'ensemble des patchs :



La numérotation des commits dans SVN.

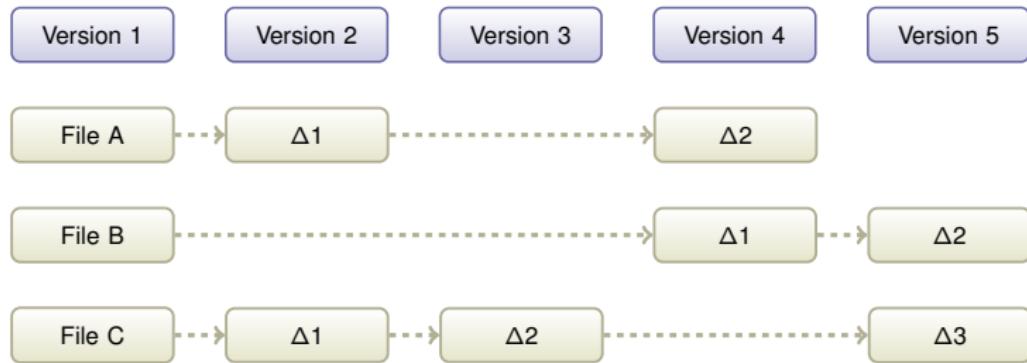
SVN utilise une numérotation globale sur l'ensemble des patchs :
=> chaque numéro de version correspond à un état cohérent ;



La numérotation des commits dans SVN.

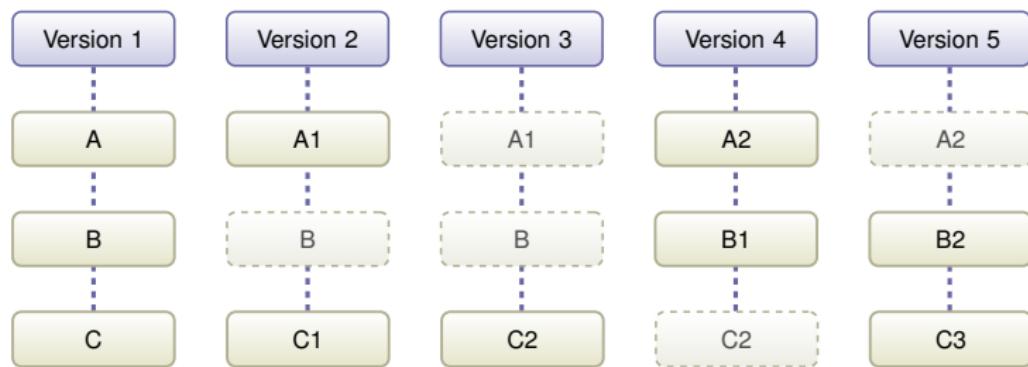
SVN utilise une numérotation globale sur l'ensemble des patchs :

- => chaque numéro de version correspond à un état cohérent ;
- => l'accès à une version nécessite de ré-appliquer les patchs.



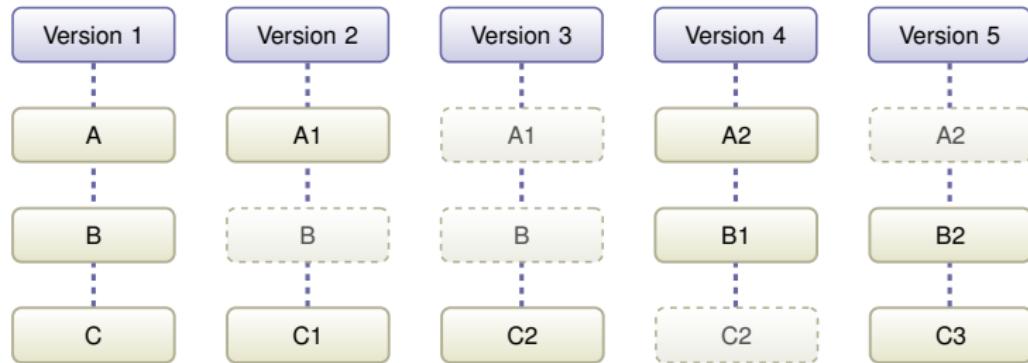
La numérotation des commits dans *Git*.

Git utilise une numérotation globale des versions des fichiers :



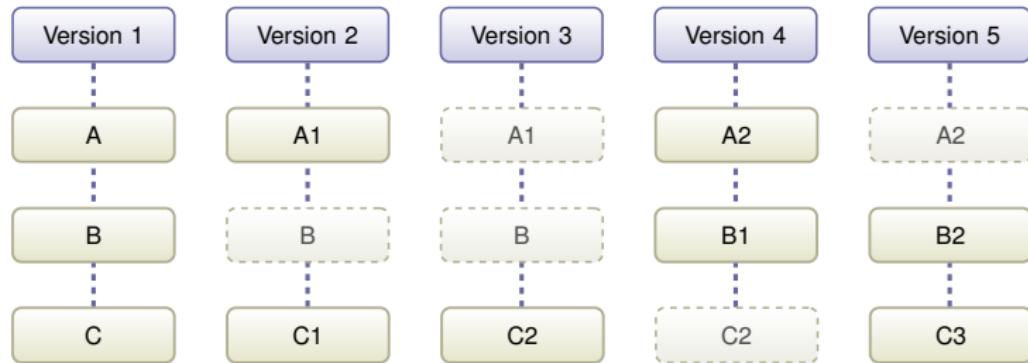
La numérotation des commits dans *Git*.

Git utilise une numérotation globale des versions des fichiers :
=> chaque numéro de version correspond à un état cohérent ;



La numérotation des commits dans *Git*.

Git utilise une numérotation globale des versions des fichiers :
=> chaque numéro de version correspond à un état cohérent ;
=> accès direct aux versions du système.



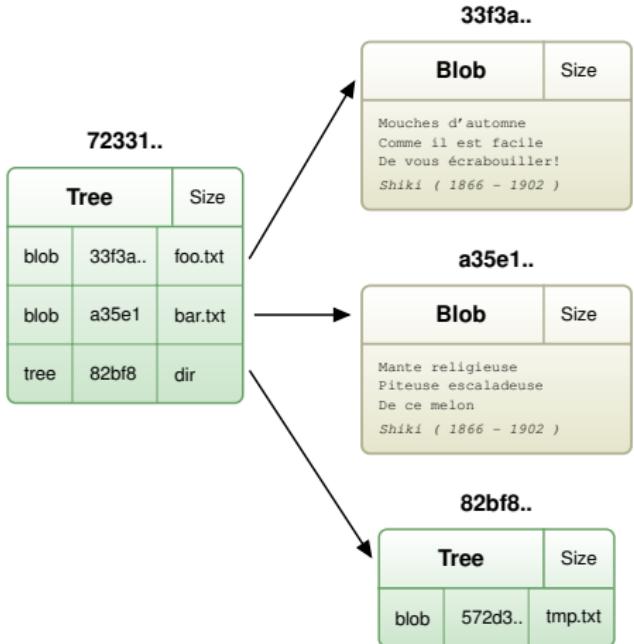
Les "Commits" : définitions

Definition

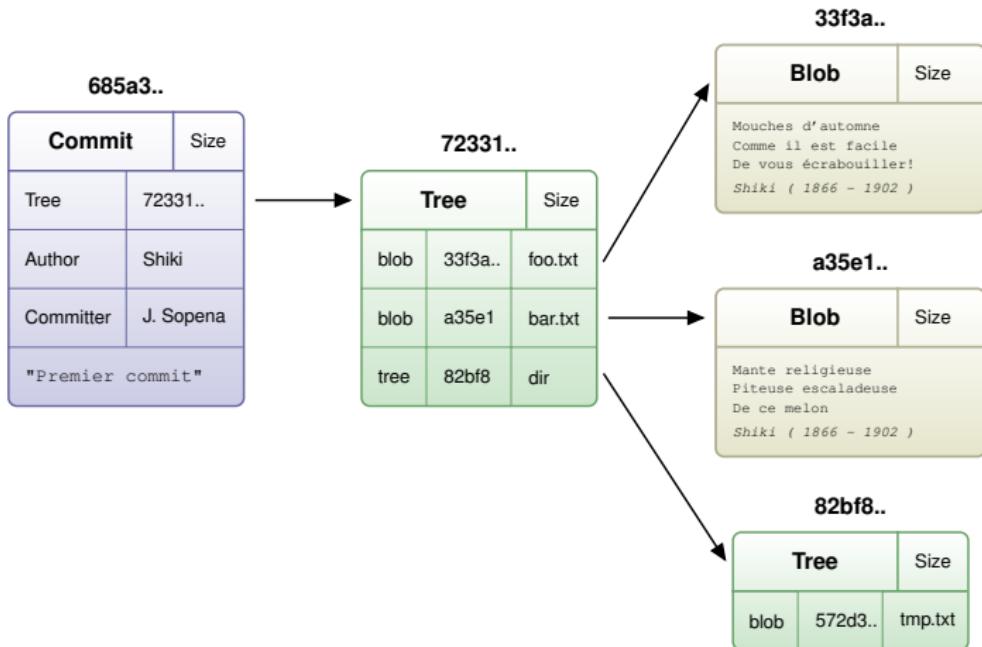
Un **Commit** stocke l'état d'une partie du dépôt à un instant donné. Il contient :

- ▶ un pointeur vers un **Tree** dont on souhaite sauver l'état.
- ▶ un pointeur vers un ou plusieurs autres **Commits** pour constituer un historique.
- ▶ le nom d'un **auteur** et d'un **commiteur**.
- ▶ une description sous forme d'une chaîne de caractères.

Les "Commits" : exemple



Les "Commits" : exemple



Contenu des blobs commit

Attention : les blobs *commit* contiennent des références à la date (en notation Epoch). Contrairement aux autres blobs, vous n'obtiendrez pas les même hash si vous refaites l'expérience.

```
$ pigz --decompress --zlib < .git/objects/10/5a55d3153a65563393*  
commit 175tree e7f3b24b01115fd0ea58d72099a24af383f627ba  
author Shiki <shiki@haikus.jp> 1633967030 +0200  
committer Julien SOPENA <julien.sopena@lip6.fr> 1633967030 +0200
```

Premier commit

Contenu des blobs commit

Attention : les blobs *commit* contiennent des références à la date (en notation Epoch). Contrairement aux autres blobs, vous n'obtiendrez pas les même hash si vous refaites l'expérience.

```
$ pigz --decompress --zlib < .git/objects/10/5a55d3153a65563393*  
commit 175tree e7f3b24b01115fd0ea58d72099a24af383f627ba  
author Shiki <shiki@haikus.jp> 1633967030 +0200  
committer Julien SOPENA <julien.sopena@lip6.fr> 1633967030 +0200
```

Premier commit

```
$ git cat-file -p e7f3b24b01115fd0ea58d72099a24af383f627ba  
100644 blob b13fb47577e28a3069183f9ad91fb0ee44d3871f bar.txt  
040000 tree 96a31457a25444a576b1fb0b8eafee735a706693 dir  
100644 blob 6ded47ba437cb9cc986e47bafc146c75e50be03c foo.txt
```

Contenu des blobs commit

Attention : les blobs *commit* contiennent des références à la date (en notation Epoch). Contrairement aux autres blobs, vous n'obtiendrez pas les même hash si vous refaites l'expérience.

```
$ pigz --decompress --zlib < .git/objects/10/5a55d3153a65563393*
commit 175tree e7f3b24b01115fd0ea58d72099a24af383f627ba
author Shiki <shiki@haikus.jp> 1633967030 +0200
committer Julien SOPENA <julien.sopena@lip6.fr> 1633967030 +0200
```

Premier commit

```
$ git cat-file -p e7f3b24b01115fd0ea58d72099a24af383f627ba
100644 blob b13fb47577e28a3069183f9ad91fb0ee44d3871f bar.txt
040000 tree 96a31457a25444a576b1fb0b8eafee735a706693 dir
100644 blob 6ded47ba437cb9cc986e47bafc146c75e50be03c foo.txt
```

La suite un peu plus tard

Outline

Git c'est quoi ?

Architecture interne de git

Création d'un dépôt

Les objets manipulé par git

Les "Blobs" (fichier)

Les "Trees"

Les "Commits"

Les "Tags"

Structure générale

Gestion des versions dans le dépôt local.

Synchronisation avec les dépôts distants.

Les outils graphiques

Conclusion

Les "Tags" : définitions

Definition

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**.
- ▶ une signature.

Commandes basiques du dépôt

init : initialisation d'un dépôt.

clone : copie d'un dépôt existant (local ou distant).

fsck-object : pour valider un dépôt.

repack : fait des paquets de **blobs** pour l'efficacité.

prune : supprime les **blobs** uniques mais existants dans un paquet.

Outline

Git c'est quoi ?

Architecture interne de git

Création d'un dépôt

Les objets manipulé par git

Les "Blobs" (fichier)

Les "Trees"

Les "Commits"

Les "Tags"

Structure générale

Gestion des versions dans le dépôt local.

Synchronisation avec les dépôts distants.

Les outils graphiques

Conclusion

Graphe acyclique des objets.

685a3..

Commit	Size
Tree	72331..
Author	Shiki
Committer	J. Sopena
"Premier commit"	

Graphe acyclique des objets.

685a3..

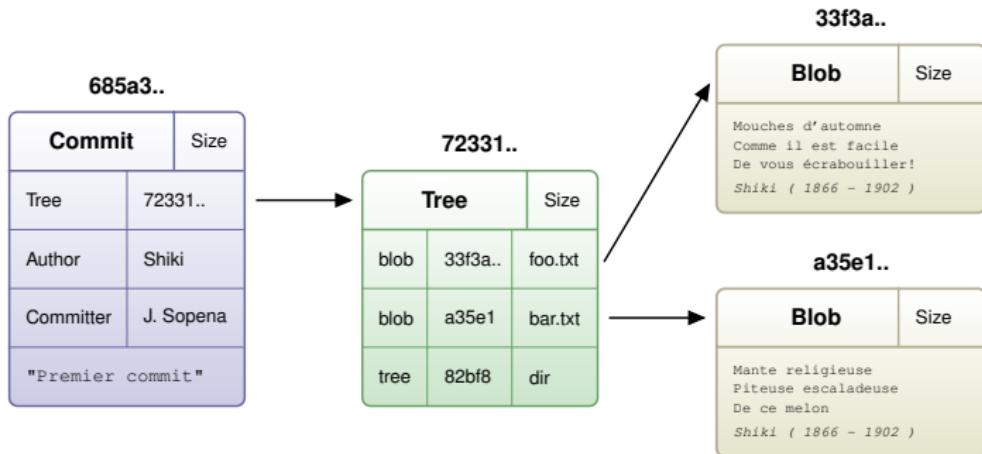
Commit		Size
Tree	72331..	
Author	Shiki	
Committer	J. Sopena	
"Premier commit"		

72331..

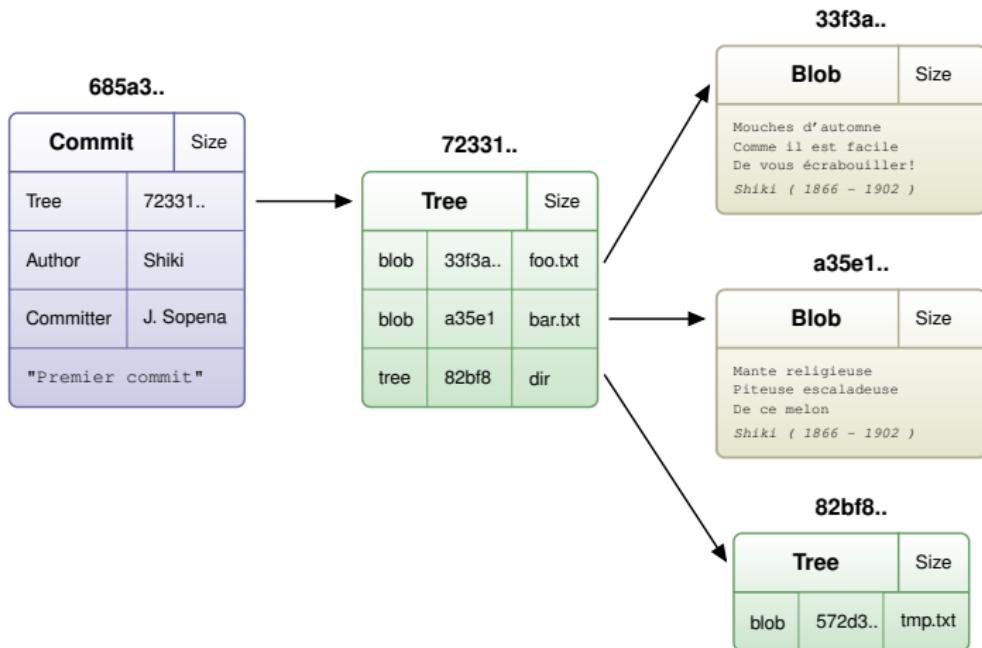
Tree		Size
blob	33f3a..	foo.txt
blob	a35e1	bar.txt
tree	82bf8	dir



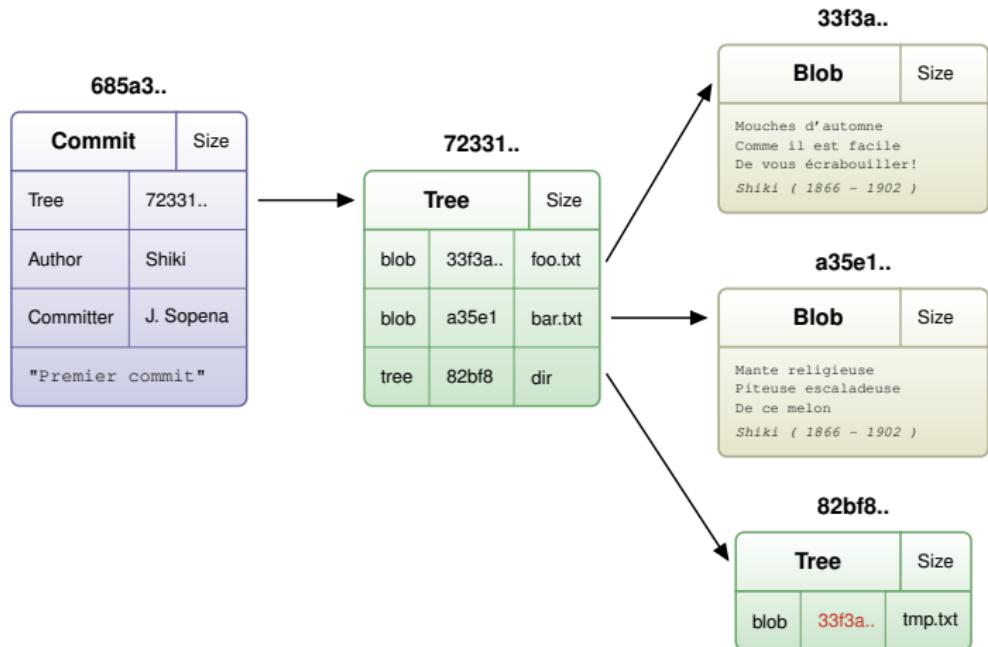
Graphe acyclique des objets.



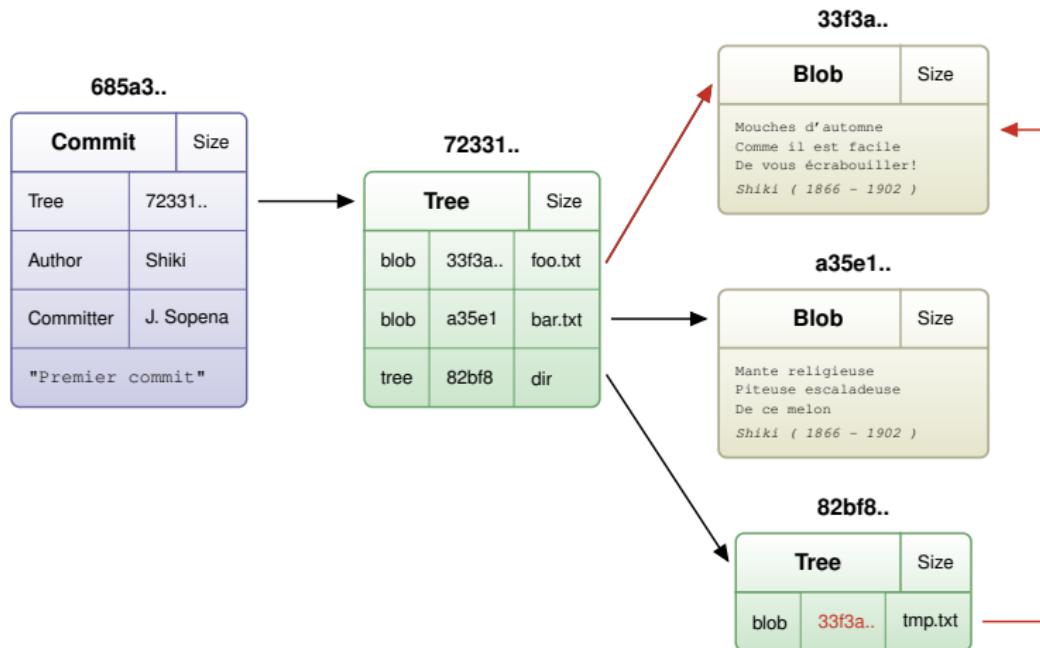
Graphe acyclique des objets.



Déduplication implicite



Déduplication implicite



Outline

Git c'est quoi ?

Architecture interne de git

Gestion des versions dans le dépôt local.

Les commits

Les branches

Les merges

Les rebases

Les remords

Utilisation de l'historique

Synchronisation avec les dépôts distants.

Les outils graphiques

Conclusion

Les commandes

Definition

Git est un ensemble de commandes indépendantes permettant d'archiver, de rechercher et de publier des ensembles d'objets représentant l'état global de l'espace de travail versionné à un instant donné.

Toutes ces commandes :

- ▶ portent un nom du type **git-<nom_de_la_commande>**.
- ▶ ont un équivalent sous la forme d'une option de la commande **git <nom_de_la_commande>**.

Exemple

```
git-add monFichier.txt ⇔ git add  
monFichier.txt
```

Git c'est un ensemble de 145 commandes.

add	daemon	index-pack	parse-remote	shortlog
add-	describe	init	patch-id	show
interactive	diff	instaweb	peek-remote	show-branch
am	diff-files	log	prune	show-index
annotate	diff-index	lost-found	prune-packed	show-ref
apply	diff-tree	ls-files	pull	stage
archimport	difftool	ls-remote	push	stash
archive	difftool-	ls-tree	quiltimport	status
bisect	helper	mailinfo	read-tree	stripspace
bisect-helper	fast-export	mailsplit	rebase	submodule
blame	fast-import	merge	rebase-	svn
branch	fetch	merge-base	interactive	symbolic-ref
bundle	fetch-tool	merge-file	receive-pack	tag
cat-file	fetch-pack	merge-index	reflog	tar-tree
check-attr	filter-branch	merge-octopus	relink	unpack-file
check-ref-	fmt-merge-msg	merge-one-file	remote	unpack-objects
format	for-each-ref	merge-ours	remote-curl	update-index
checkout	format-patch	merge-	repack	update-ref
checkout-index	fsck	recursive	replace	update-server-
cherry	fsck-objects	merge-resolve	repo-config	info
cherry-pick	gc	merge-subtree	request-pull	upload-archive
citool	get-tar-	merge-tree	rerere	upload-pack
clean	commit-id	mergetool	reset	var
clone	grep	mergetool-lib	rev-list	verify-pack
commit	gui (et gitk)	mktag	rev-parse	verify-tag
commit-tree	gui-askpass	mktree	revert	web-browse
config	hash-object	mv	rm	whatchanged
count-objects	help	name-rev	send-email	write-tree
cvsexportcommit	http-fetch	pack-objects	send-pack	
cvsimport	http-push	pack-redundant	sh-setup	
cvsserver	imap-send	pack-refs	shell	

Git c'est un ensemble de 145 commandes.

Les commandes permettant de gérer un dépôt local

```
add           init
             diff      log      prune
                           stash    status
branch        rebase
               tag
               checkout
               grep      reset
commit
config        mv       revert
               help     rm
```

Git c'est un ensemble de 145 commandes.

Les commandes permettant gérer des dépôts distants

Git c'est un ensemble de 145 commandes.

git-gui et gitk pour l'interface graphique.

add		init					
am	diff	log		prune			
				pull	stash		
				push	status		
			merge	rebase			
branch							
	fetch					tag	
checkout							
					reset		
clone	grep						
commit	gui (et gitk)						
					revert		
config		mv					
					rm		
	help						

Git c'est un ensemble de 145 commandes.

Les commandes pour une utilisation avancée

```
add           init
am            diff          log
              prune
bisect
blame         merge         pull
branch        fetch          push
                  rebase
tag
checkout      repack
gc
clone         grep          rerere
commit        gui (et gitk)  reset
config        mv
help
revert
rm
```

Git c'est un ensemble de 145 commandes.

Les autres commandes au cas par cas

add	cvsserver	index-pack	pack-refs	shortlog
add-	daemon	init	parse-remote	show
interactive	describe	instaweb	patch-id	show-branch
am	diff	log	peek-remote	show-index
annotate	diff-files	lost-found	prune	show-ref
apply	diff-index	ls-files	prune-packed	stage
archimport	diff-tree	ls-remote	pull	stash
archive	difftool	ls-tree	push	status
bisect	difftool-	mailinfo	quiltimport	stripspace
bisect-helper	helper	mailsplit	read-tree	submodule
blame	fast-export	merge	rebase	svn
branch	fast-import	merge-base	rebase-	symbolic-ref
bundle	fetch	merge-file	interactive	tag
cat-file	fetch-tool	merge-index	receive-pack	tar-tree
check-attr	fetch-pack	merge-octopus	reflog	unpack-file
check-ref-	filter-branch	merge-one-file	relink	
format	fmt-merge-msg	merge-ours	remote	unpack-objects
checkout	for-each-ref	merge-	remote-curl	update-index
	format-patch	recursive	repack	update-ref
checkout-index	fsck	merge-resolve	replace	update-
cherry	fsck-objects	merge-subtree	repo-config	server-info
cherry-pick	gc	merge-tree	request-pull	
citool	get-tar-	mergetool	rerere	upload-archive
clean	commit-id	mergetool-lib	reset	upload-pack
clone	grep	mktag	rev-list	var
commit	gui (et gitk)	mktree	rev-parse	verify-pack
commit-tree	gui-askpass	mv	revert	verify-tag
config	hash-object	name-rev	rm	web-browse
count-objects	help	pack-objects	send-email	whatchanged
cvsexportcom-	http-fetch	pack-redundant	send-pack	write-tree
mit	http-push	imap-send	sh-setup	
cvimport			shell	

Outline

Git c'est quoi ?

Architecture interne de git

Gestion des versions dans le dépôt local.

Les commits

Les branches

Les merges

Les rebases

Les remords

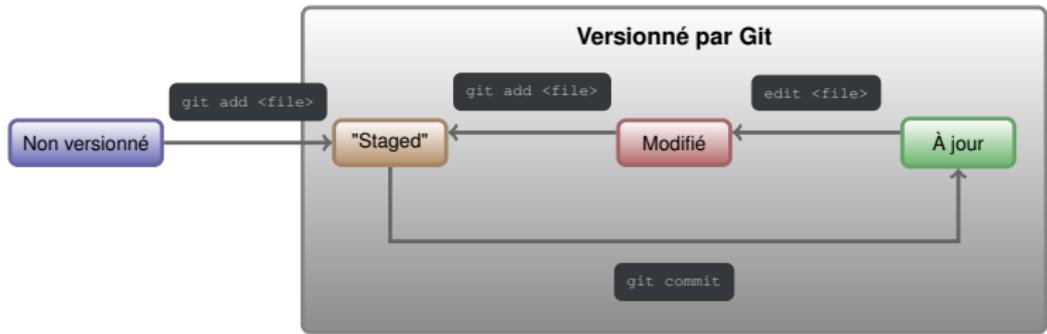
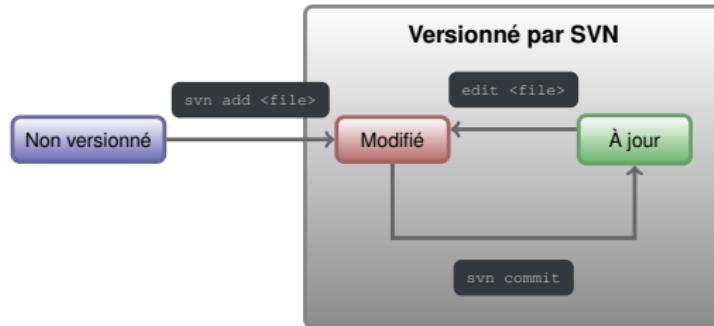
Utilisation de l'historique

Synchronisation avec les dépôts distants.

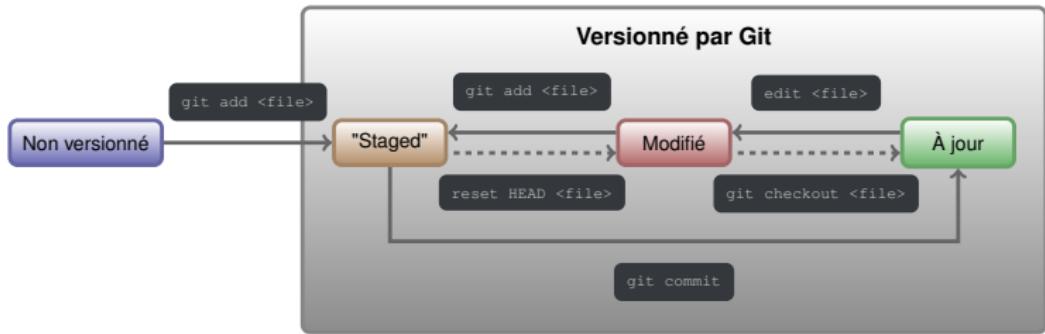
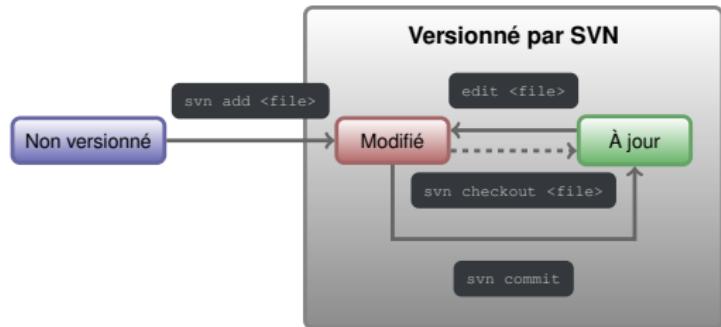
Les outils graphiques

Conclusion

Commit : différence avec cvs/svn.



Commit : différence avec cvs/svn.



Commit : les commandes.

add : ajoute dans l'index un fichier à commiter **dans son état actuel**.

commit : enregistre dans le dépôt local les modifications qui ont été **ajoutées dans l'index** par une commande **add**.

reset HEAD : supprime la référence d'un fichier de l'index ajouté par une commande **add**.

Exemple

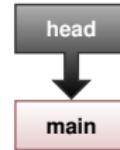
```
$ echo "Premier fichier" > foo.txt  
$ git add foo.txt  
$ git commit -m "Description de ce commit"
```

Étude des objets générés par un exemple simple.

```
$ mkdir project  
$ cd project  
$ git init
```

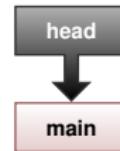
Étude des objets générés par un exemple simple.

```
$ mkdir project  
$ cd project  
$ git init
```



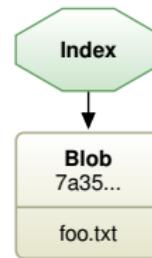
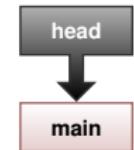
Étude des objets générés par un exemple simple.

```
$ mkdir project  
$ cd project  
$ git init  
  
$ echo "toto" > foo.txt  
$ git add foo.txt
```



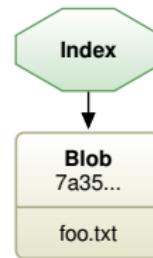
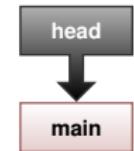
Étude des objets générés par un exemple simple.

```
$ mkdir project  
$ cd project  
$ git init  
  
$ echo "toto" > foo.txt  
$ git add foo.txt
```



Étude des objets générés par un exemple simple.

```
$ mkdir project  
$ cd project  
$ git init  
  
$ echo "toto" > foo.txt  
$ git add foo.txt  
  
$ git commit -m "Add foo.txt"
```

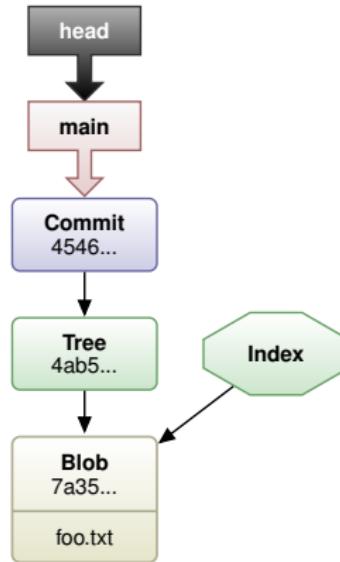


Étude des objets générés par un exemple simple.

```
$ mkdir project
$ cd project
$ git init

$ echo "toto" > foo.txt
$ git add foo.txt

$ git commit -m "Add foo.txt"
```



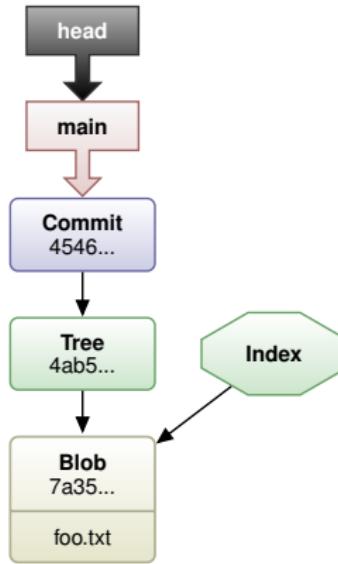
Étude des objets générés par un exemple simple.

```
$ mkdir project
$ cd project
$ git init

$ echo "toto" > foo.txt
$ git add foo.txt

$ git commit -m "Add foo.txt"

$ mkdir dir
$ echo "titi" > dir/bar.txt
$ git add dir/bar.txt
```



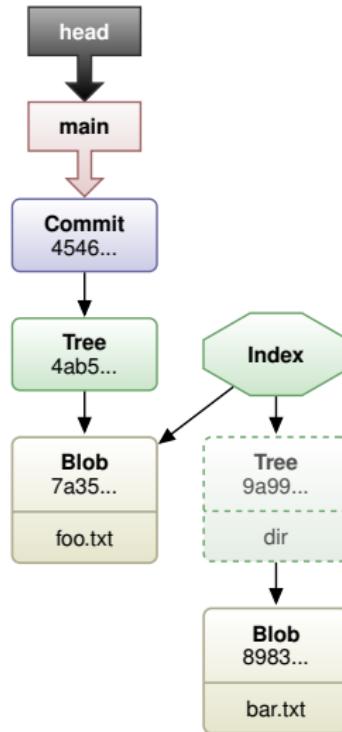
Étude des objets générés par un exemple simple.

```
$ mkdir project
$ cd project
$ git init

$ echo "toto" > foo.txt
$ git add foo.txt

$ git commit -m "Add foo.txt"

$ mkdir dir
$ echo "titi" > dir/bar.txt
$ git add dir/bar.txt
```



Étude des objets générés par un exemple simple.

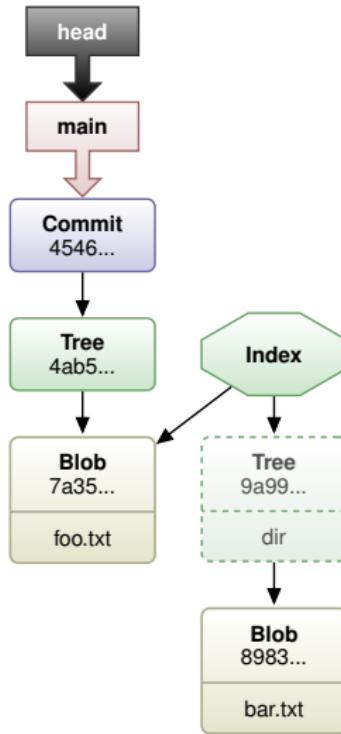
```
$ mkdir project
$ cd project
$ git init

$ echo "toto" > foo.txt
$ git add foo.txt

$ git commit -m "Add foo.txt"

$ mkdir dir
$ echo "titi" > dir/bar.txt
$ git add dir/bar.txt

$ git commit -m "Add dir/bar.txt"
```



Étude des objets générés par un exemple simple.

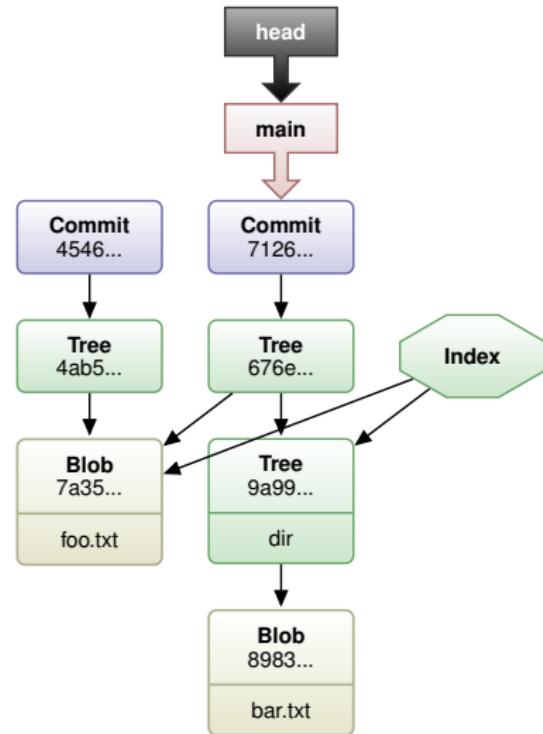
```
$ mkdir project
$ cd project
$ git init

$ echo "toto" > foo.txt
$ git add foo.txt

$ git commit -m "Add foo.txt"

$ mkdir dir
$ echo "titi" > dir/bar.txt
$ git add dir/bar.txt

$ git commit -m "Add dir/bar.txt"
```



Étude des objets générés par un exemple simple.

```
$ mkdir project
$ cd project
$ git init

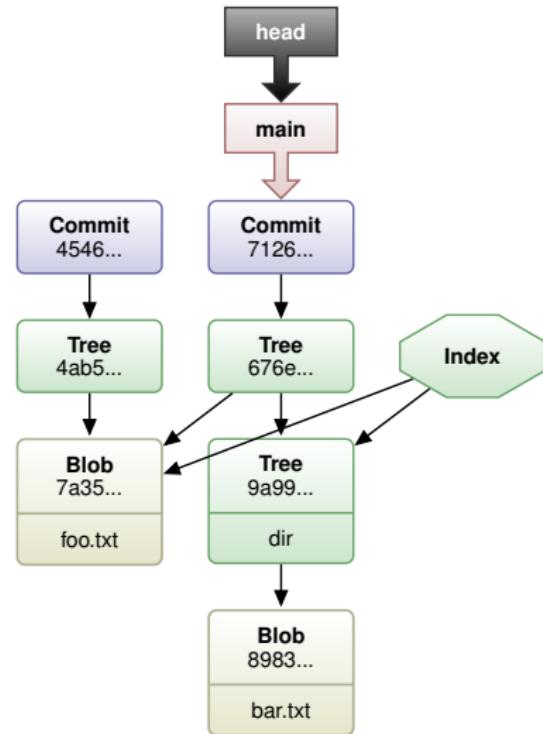
$ echo "toto" > foo.txt
$ git add foo.txt

$ git commit -m "Add foo.txt"

$ mkdir dir
$ echo "titi" > dir/bar.txt
$ git add dir/bar.txt

$ git commit -m "Add dir/bar.txt"

$ echo "tutu" >> dir/bar.txt
$ git add dir/bar.txt
```



Étude des objets générés par un exemple simple.

```
$ mkdir project
$ cd project
$ git init

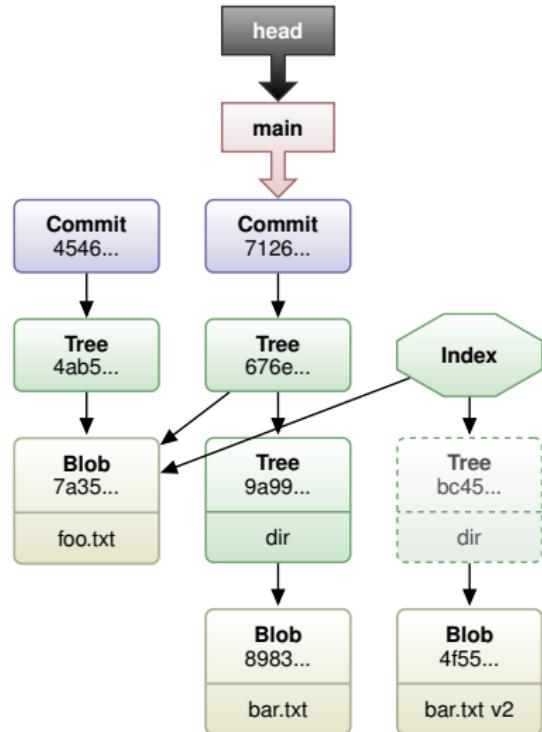
$ echo "toto" > foo.txt
$ git add foo.txt

$ git commit -m "Add foo.txt"

$ mkdir dir
$ echo "titi" > dir/bar.txt
$ git add dir/bar.txt

$ git commit -m "Add dir/bar.txt"

$ echo "tutu" >> dir/bar.txt
$ git add dir/bar.txt
```



Étude des objets générés par un exemple simple.

```
$ mkdir project
$ cd project
$ git init

$ echo "toto" > foo.txt
$ git add foo.txt

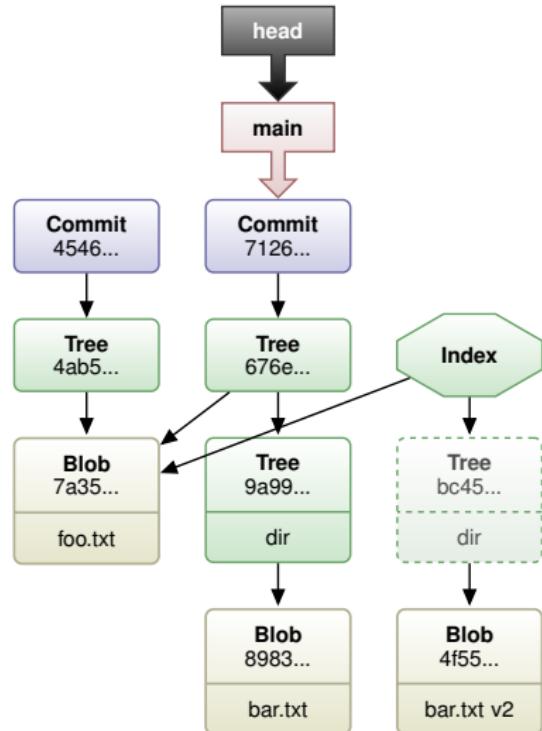
$ git commit -m "Add foo.txt"

$ mkdir dir
$ echo "titi" > dir/bar.txt
$ git add dir/bar.txt

$ git commit -m "Add dir/bar.txt"

$ echo "tutu" >> dir/bar.txt
$ git add dir/bar.txt

$ git commit -m "Modif dir/bar.txt"
```



Étude des objets générés par un exemple simple.

```
$ mkdir project
$ cd project
$ git init

$ echo "toto" > foo.txt
$ git add foo.txt

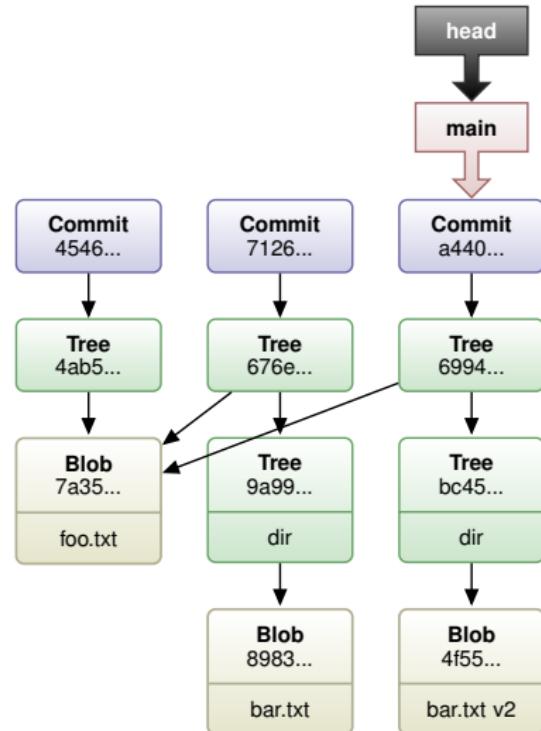
$ git commit -m "Add foo.txt"

$ mkdir dir
$ echo "titi" > dir/bar.txt
$ git add dir/bar.txt

$ git commit -m "Add dir/bar.txt"

$ echo "tutu" >> dir/bar.txt
$ git add dir/bar.txt

$ git commit -m "Modif dir/bar.txt"
```



Étude des objets générés par un exemple simple.

```
$ mkdir project
$ cd project
$ git init

$ echo "toto" > foo.txt
$ git add foo.txt

$ git commit -m "Add foo.txt"

$ mkdir dir
$ echo "titi" > dir/bar.txt
$ git add dir/bar.txt

$ git commit -m "Add dir/bar.txt"

$ echo "tutu" >> dir/bar.txt
$ git add dir/bar.txt

$ git commit -m "Modif dir/bar.txt"
```



Étude des objets générés par un exemple simple.

```
$ mkdir project
$ cd project
$ git init

$ echo "toto" > foo.txt
$ git add foo.txt

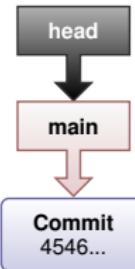
$ git commit -m "Add foo.txt"

$ mkdir dir
$ echo "titi" > dir/bar.txt
$ git add dir/bar.txt

$ git commit -m "Add dir/bar.txt"

$ echo "tutu" >> dir/bar.txt
$ git add dir/bar.txt

$ git commit -m "Modif dir/bar.txt"
```



Étude des objets générés par un exemple simple.

```
$ mkdir project
$ cd project
$ git init

$ echo "toto" > foo.txt
$ git add foo.txt

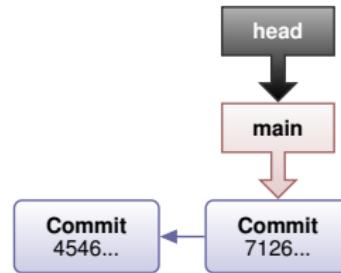
$ git commit -m "Add foo.txt"

$ mkdir dir
$ echo "titi" > dir/bar.txt
$ git add dir/bar.txt

$ git commit -m "Add dir/bar.txt"

$ echo "tutu" >> dir/bar.txt
$ git add dir/bar.txt

$ git commit -m "Modif dir/bar.txt"
```



Étude des objets générés par un exemple simple.

```
$ mkdir project
$ cd project
$ git init

$ echo "toto" > foo.txt
$ git add foo.txt

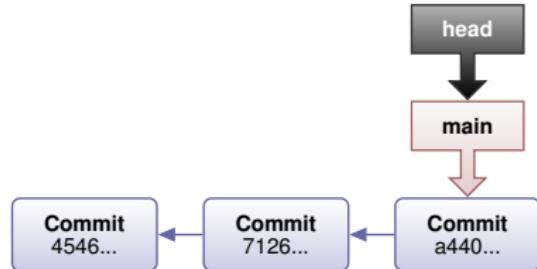
$ git commit -m "Add foo.txt"

$ mkdir dir
$ echo "titi" > dir/bar.txt
$ git add dir/bar.txt

$ git commit -m "Add dir/bar.txt"

$ echo "tutu" >> dir/bar.txt
$ git add dir/bar.txt

$ git commit -m "Modif dir/bar.txt"
```



Étude des objets générés par un exemple simple.

```
$ mkdir project
$ cd project
$ git init

$ echo "toto" > foo.txt
$ git add foo.txt

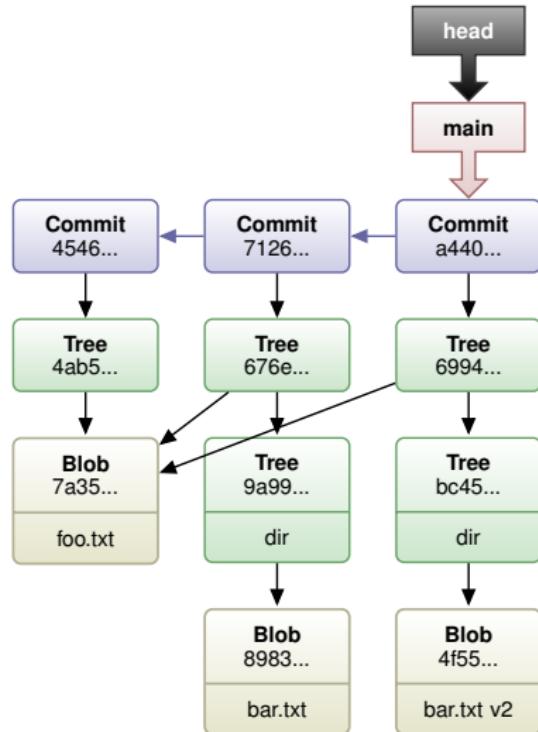
$ git commit -m "Add foo.txt"

$ mkdir dir
$ echo "titi" > dir/bar.txt
$ git add dir/bar.txt

$ git commit -m "Add dir/bar.txt"

$ echo "tutu" >> dir/bar.txt
$ git add dir/bar.txt

$ git commit -m "Modif dir/bar.txt"
```



Chaîne de commit dans les Blobs

```
$ pigz --decompress --zlib < .git/objects/10/5a55d3153a65563393*  
commit 175tree e7f3b24b01115fd0ea58d72099a24af383f627ba  
author Shiki <shiki@haikus.jp> 1633967030 +0200  
committer Julien SOPENA <julien.sopena@lip6.fr> 1633967030 +0200
```

Premier commit

Chaîne de commit dans les Blobs

```
$ pigz --decompress --zlib < .git/objects/10/5a55d3153a65563393*
commit 175tree e7f3b24b01115fd0ea58d72099a24af383f627ba
author Shiki <shiki@haikus.jp> 1633967030 +0200
committer Julien SOPENA <julien.sopena@lip6.fr> 1633967030 +0200
```

Premier commit

```
$ TODO
```

Outline

Git c'est quoi ?

Architecture interne de git

Gestion des versions dans le dépôt local.

Les commits

Les branches

Les merges

Les rebases

Les remords

Utilisation de l'historique

Synchronisation avec les dépôts distants.

Les outils graphiques

Conclusion

Branche : les commandes.

branch : liste les branches avec une * pour la branche active.

branch <nom> : crée une nouvelle branche <nom>;

branch -m : permet de renommer une branche;

branch -d : permet de supprimer une branche;

switch : change de branche active;

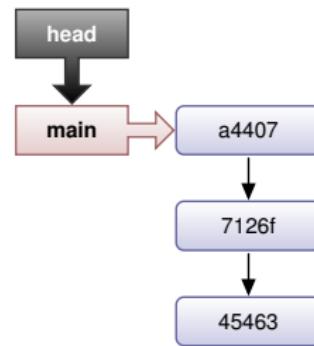
switch -c : crée une branche et l'active;

show-branch : affiche les branches et leurs commits.

```
$ git branch
  * main
$ git branch maBranche
$ git branch
  maBranche
  * main
$ git switch maBranche
$ git branch
  * maBranche
    main
```

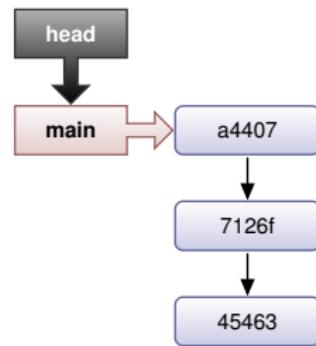
Branche : isoler provisoirement un développement

```
$ ls  
foo.txt dir
```



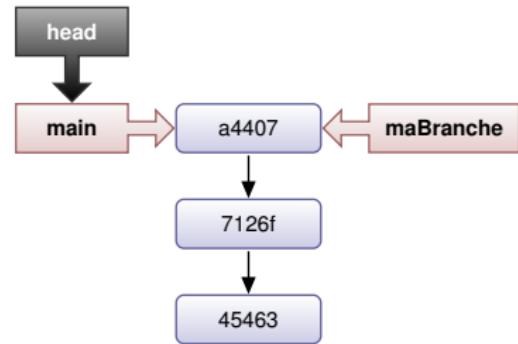
Branche : isoler provisoirement un développement

```
$ ls  
foo.txt dir  
  
$ git branch maBranche
```



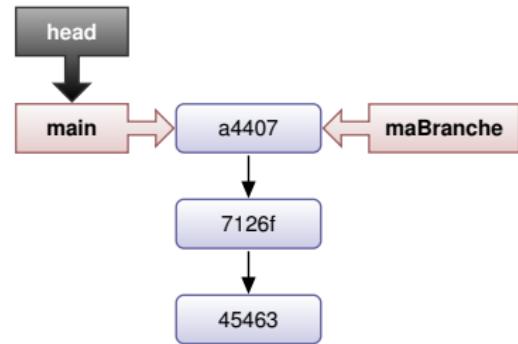
Branche : isoler provisoirement un développement

```
$ ls  
foo.txt dir  
  
$ git branch maBranche
```



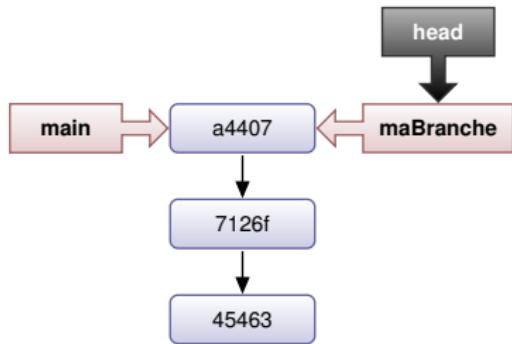
Branche : isoler provisoirement un développement

```
$ ls  
foo.txt dir  
  
$ git branch maBranche  
  
$ git switch maBranche
```



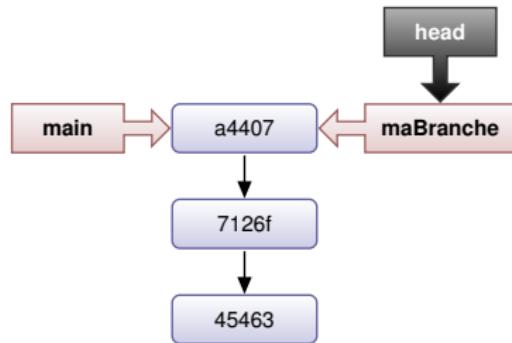
Branche : isoler provisoirement un développement

```
$ ls  
foo.txt dir  
  
$ git branch maBranche  
  
$ git switch maBranche
```



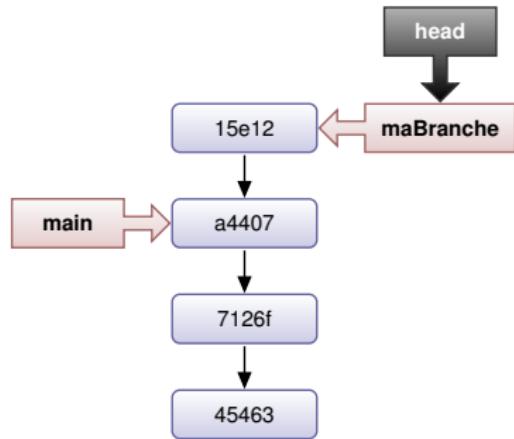
Branche : isoler provisoirement un développement

```
$ ls  
foo.txt dir  
  
$ git branch maBranche  
  
$ git switch maBranche  
  
$ touch fichier1.txt  
$ ls  
dir fichier1.txt foo.txt  
$ git add fichier1.txt  
$ git commit -m "Add fichier1.txt"
```



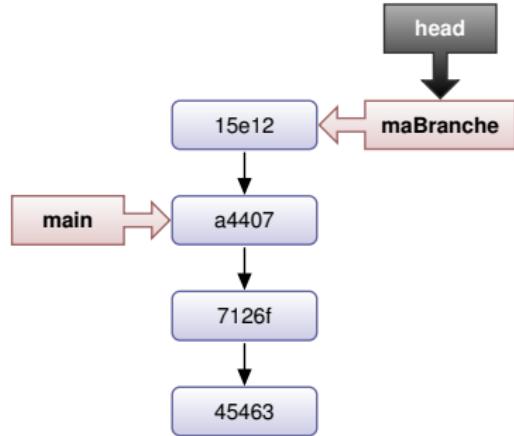
Branche : isoler provisoirement un développement

```
$ ls  
foo.txt dir  
  
$ git branch maBranche  
  
$ git switch maBranche  
  
$ touch fichier1.txt  
$ ls  
dir fichier1.txt foo.txt  
$ git add fichier1.txt  
$ git commit -m "Add fichier1.txt"
```



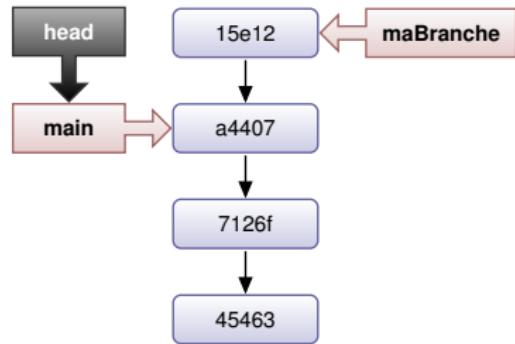
Branche : isoler provisoirement un développement

```
$ ls  
foo.txt dir  
  
$ git branch maBranche  
  
$ git switch maBranche  
  
$ touch fichier1.txt  
$ ls  
dir fichier1.txt foo.txt  
$ git add fichier1.txt  
$ git commit -m "Add fichier1.txt"  
  
$ git switch main  
$ ls  
dir foo.txt
```



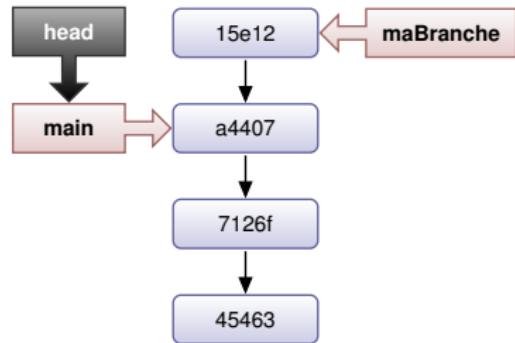
Branche : isoler provisoirement un développement

```
$ ls  
foo.txt dir  
  
$ git branch maBranche  
  
$ git switch maBranche  
  
$ touch fichier1.txt  
$ ls  
dir fichier1.txt foo.txt  
$ git add fichier1.txt  
$ git commit -m "Add fichier1.txt"  
  
$ git switch main  
$ ls  
dir foo.txt
```



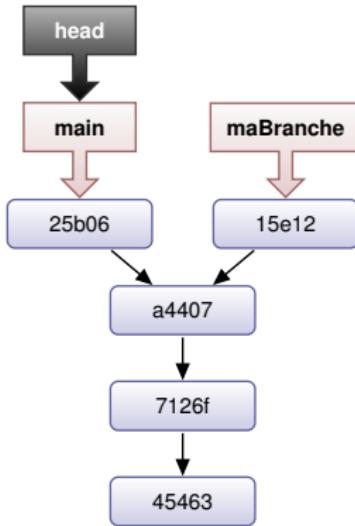
Branche : isoler provisoirement un développement

```
$ ls  
foo.txt dir  
  
$ git branch maBranche  
  
$ git switch maBranche  
  
$ touch fichier1.txt  
$ ls  
dir fichier1.txt foo.txt  
$ git add fichier1.txt  
$ git commit -m "Add fichier1.txt"  
  
$ git switch main  
$ ls  
dir foo.txt  
  
$ touch fichier2.txt  
$ git add fichier2.txt  
$ git commit -m "Add fichier2.txt"
```



Branche : isoler provisoirement un développement

```
$ ls  
foo.txt dir  
  
$ git branch maBranche  
  
$ git switch maBranche  
  
$ touch fichier1.txt  
$ ls  
dir fichier1.txt foo.txt  
$ git add fichier1.txt  
$ git commit -m "Add fichier1.txt"  
  
$ git switch main  
$ ls  
dir foo.txt  
  
$ touch fichier2.txt  
$ git add fichier2.txt  
$ git commit -m "Add fichier2.txt"
```



Outline

Git c'est quoi ?

Architecture interne de git

Gestion des versions dans le dépôt local.

Les commits

Les branches

Les merges

Les rebases

Les remords

Utilisation de l'historique

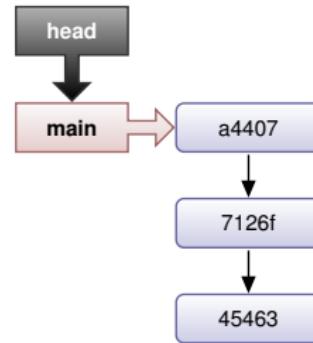
Synchronisation avec les dépôts distants.

Les outils graphiques

Conclusion

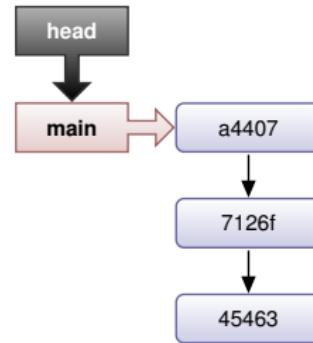
fast-forward : merge sans création de commit

```
$ ls  
foo.txt dir
```



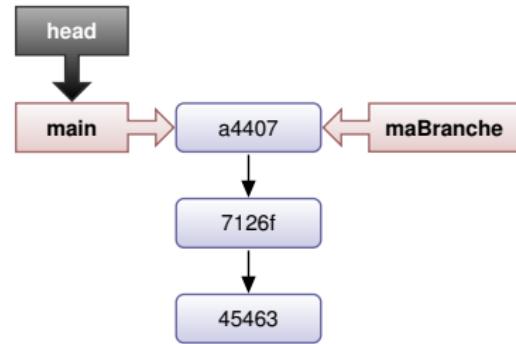
fast-forward : merge sans création de commit

```
$ ls  
foo.txt dir  
  
$ git switch -c maBranche
```



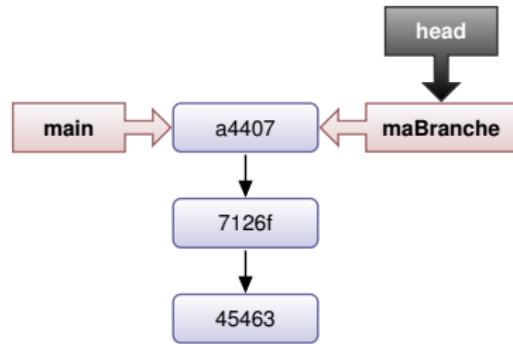
fast-forward : merge sans création de commit

```
$ ls  
foo.txt dir  
  
$ git switch -c maBranche
```



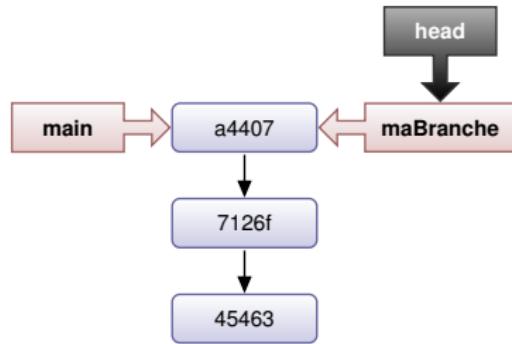
fast-forward : merge sans création de commit

```
$ ls  
foo.txt dir  
  
$ git switch -c maBranche
```



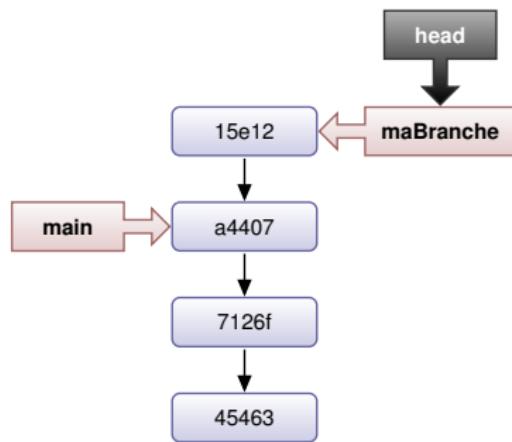
fast-forward : merge sans création de commit

```
$ ls  
foo.txt dir  
  
$ git switch -c maBranche  
  
$ echo "toto" > fichier1.txt  
$ git add fichier1.txt  
$ git commit -m "Add fichier1.txt"
```



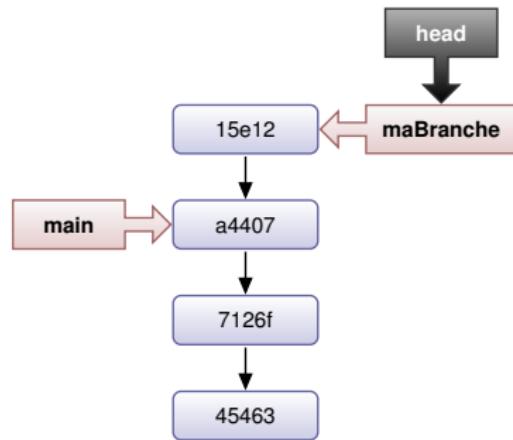
fast-forward : merge sans création de commit

```
$ ls  
foo.txt dir  
  
$ git switch -c maBranche  
  
$ echo "toto" > fichier1.txt  
$ git add fichier1.txt  
$ git commit -m "Add fichier1.txt"
```



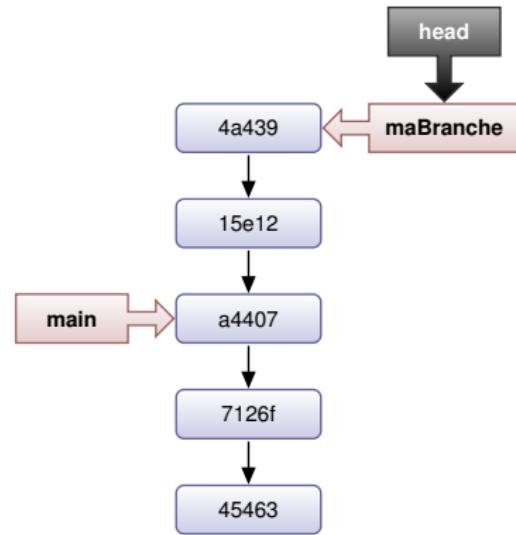
fast-forward : merge sans création de commit

```
$ ls  
foo.txt dir  
  
$ git switch -c maBranche  
  
$ echo "toto" > fichier1.txt  
$ git add fichier1.txt  
$ git commit -m "Add fichier1.txt"  
  
$ echo "titi" > fichier1.txt  
$ git commit -am "New fichier1.txt"
```



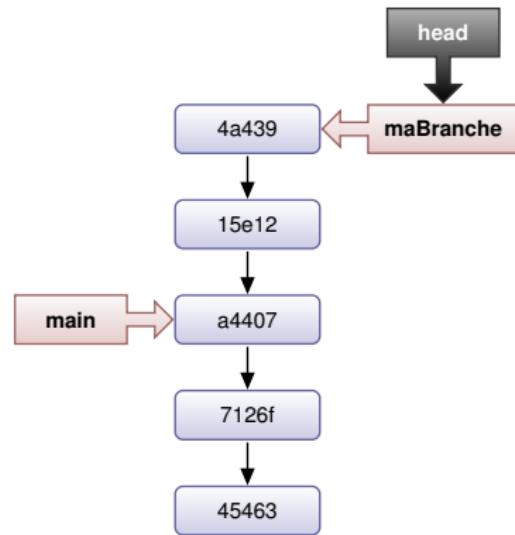
fast-forward : merge sans création de commit

```
$ ls  
foo.txt dir  
  
$ git switch -c maBranche  
  
$ echo "toto" > fichier1.txt  
$ git add fichier1.txt  
$ git commit -m "Add fichier1.txt"  
  
$ echo "titi" > fichier1.txt  
$ git commit -am "New fichier1.txt"
```



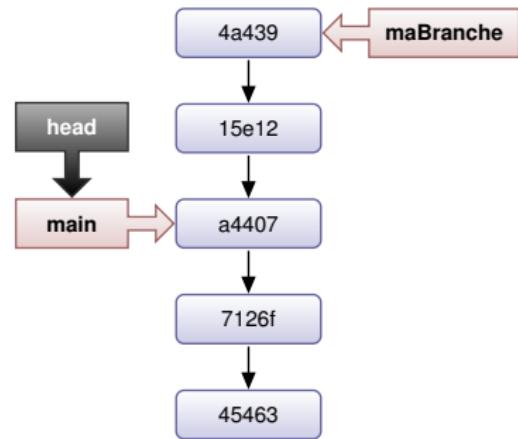
fast-forward : merge sans création de commit

```
$ ls  
foo.txt dir  
  
$ git switch -c maBranche  
  
$ echo "toto" > fichier1.txt  
$ git add fichier1.txt  
$ git commit -m "Add fichier1.txt"  
  
$ echo "titi" > fichier1.txt  
$ git commit -am "New fichier1.txt"  
  
$ git switch main  
$ ls  
dir foo.txt
```



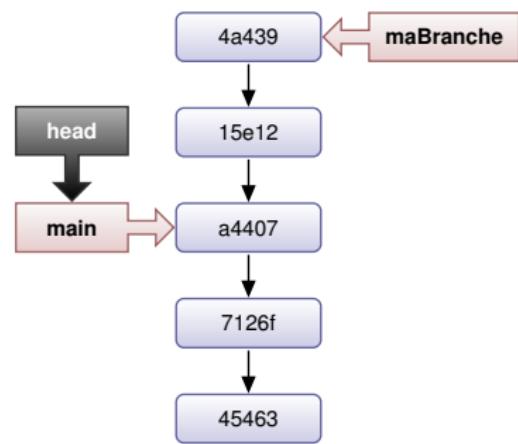
fast-forward : merge sans création de commit

```
$ ls  
foo.txt dir  
  
$ git switch -c maBranche  
  
$ echo "toto" > fichier1.txt  
$ git add fichier1.txt  
$ git commit -m "Add fichier1.txt"  
  
$ echo "titi" > fichier1.txt  
$ git commit -am "New fichier1.txt"  
  
$ git switch main  
$ ls  
dir foo.txt
```



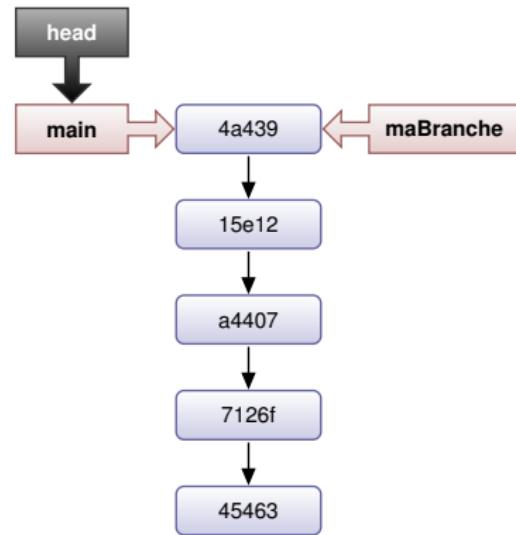
fast-forward : merge sans création de commit

```
$ ls  
foo.txt dir  
  
$ git switch -c maBranche  
  
$ echo "toto" > fichier1.txt  
$ git add fichier1.txt  
$ git commit -m "Add fichier1.txt"  
  
$ echo "titi" > fichier1.txt  
$ git commit -am "New fichier1.txt"  
  
$ git switch main  
$ ls  
dir foo.txt  
  
$ git merge maBranche  
$ cat fichier1.txt  
titi
```



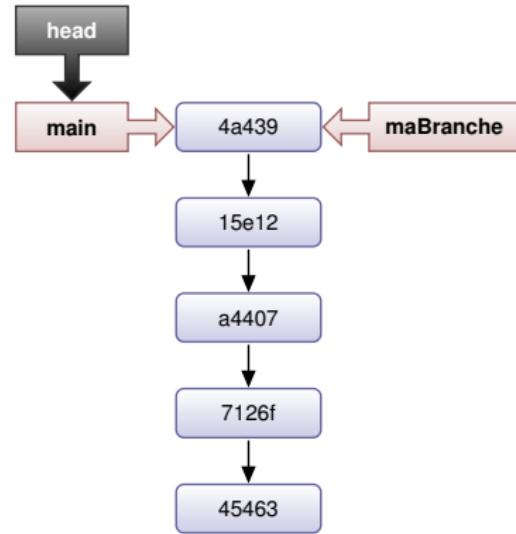
fast-forward : merge sans création de commit

```
$ ls  
foo.txt dir  
  
$ git switch -c maBranche  
  
$ echo "toto" > fichier1.txt  
$ git add fichier1.txt  
$ git commit -m "Add fichier1.txt"  
  
$ echo "titi" > fichier1.txt  
$ git commit -am "New fichier1.txt"  
  
$ git switch main  
$ ls  
dir foo.txt  
  
$ git merge maBranche  
$ cat fichier1.txt  
titi
```



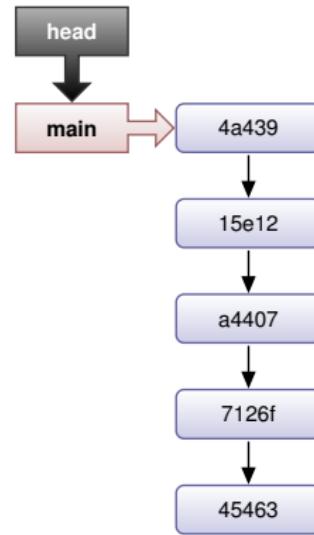
fast-forward : merge sans création de commit

```
$ ls  
foo.txt dir  
  
$ git switch -c maBranche  
  
$ echo "toto" > fichier1.txt  
$ git add fichier1.txt  
$ git commit -m "Add fichier1.txt"  
  
$ echo "titi" > fichier1.txt  
$ git commit -am "New fichier1.txt"  
  
$ git switch main  
$ ls  
dir foo.txt  
  
$ git merge maBranche  
$ cat fichier1.txt  
titi  
  
$ git branch -d maBranche
```



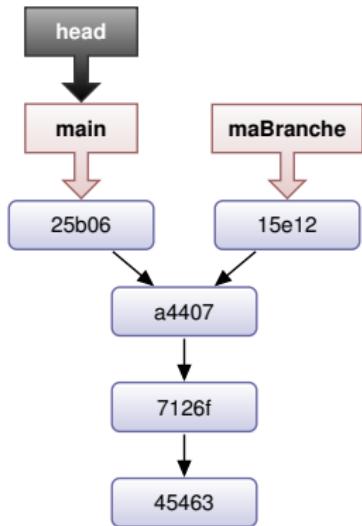
fast-forward : merge sans création de commit

```
$ ls  
foo.txt dir  
  
$ git switch -c maBranche  
  
$ echo "toto" > fichier1.txt  
$ git add fichier1.txt  
$ git commit -m "Add fichier1.txt"  
  
$ echo "titi" > fichier1.txt  
$ git commit -am "New fichier1.txt"  
  
$ git switch main  
$ ls  
dir foo.txt  
  
$ git merge maBranche  
$ cat fichier1.txt  
titi  
  
$ git branch -d maBranche
```



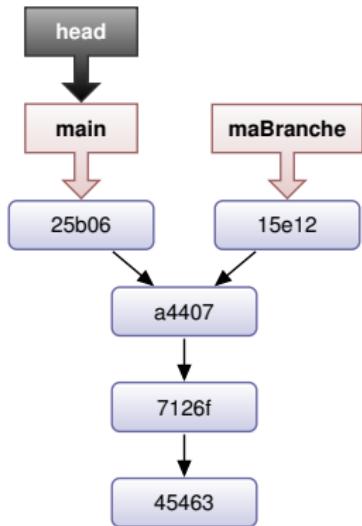
Merge : exemple sur deux branches distinctes.

```
$ ls  
dir fichier2.txt foo.txt
```



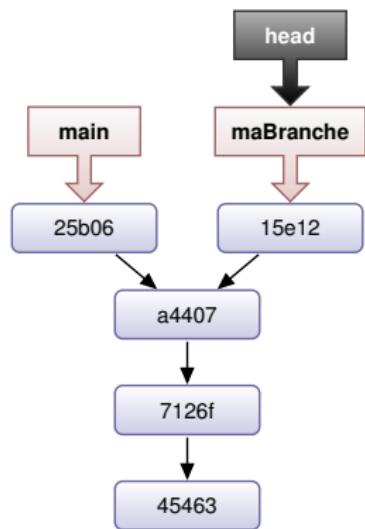
Merge : exemple sur deux branches distinctes.

```
$ ls  
dir fichier2.txt foo.txt  
  
$ git switch maBranche  
$ ls  
dir fichier1.txt foo.txt
```



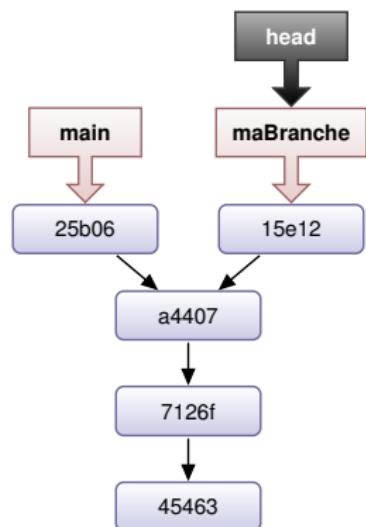
Merge : exemple sur deux branches distinctes.

```
$ ls  
dir fichier2.txt foo.txt  
  
$ git switch maBranche  
$ ls  
dir fichier1.txt foo.txt
```



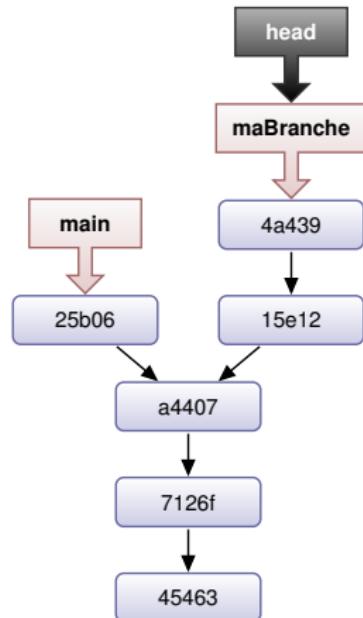
Merge : exemple sur deux branches distinctes.

```
$ ls  
dir fichier2.txt foo.txt  
  
$ git switch maBranche  
$ ls  
dir fichier1.txt foo.txt  
  
$ echo "titi" > fichier1.txt  
$ git commit -am "New fichier1.txt"
```



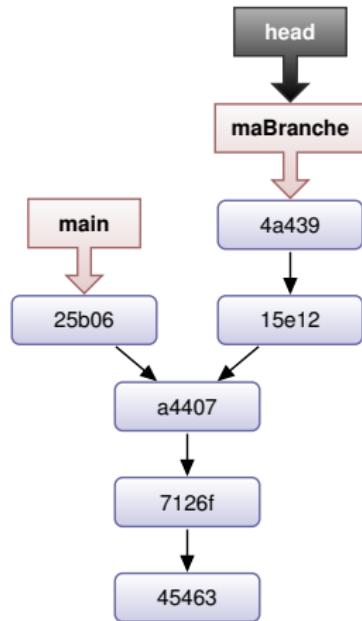
Merge : exemple sur deux branches distinctes.

```
$ ls  
dir fichier2.txt foo.txt  
  
$ git switch maBranche  
$ ls  
dir fichier1.txt foo.txt  
  
$ echo "titi" > fichier1.txt  
$ git commit -am "New fichier1.txt"
```



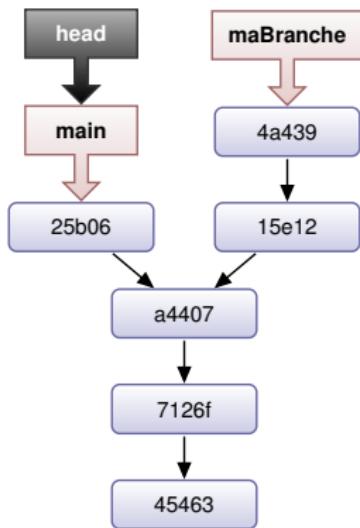
Merge : exemple sur deux branches distinctes.

```
$ ls  
dir fichier2.txt foo.txt  
  
$ git switch maBranche  
$ ls  
dir fichier1.txt foo.txt  
  
$ echo "titi" > fichier1.txt  
$ git commit -am "New fichier1.txt"  
  
$ git switch main
```



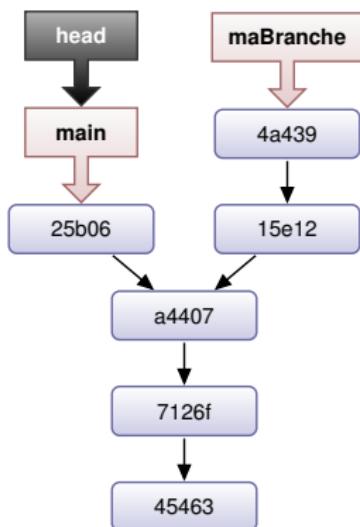
Merge : exemple sur deux branches distinctes.

```
$ ls  
dir fichier2.txt foo.txt  
  
$ git switch maBranche  
$ ls  
dir fichier1.txt foo.txt  
  
$ echo "titi" > fichier1.txt  
$ git commit -am "New fichier1.txt"  
  
$ git switch main
```



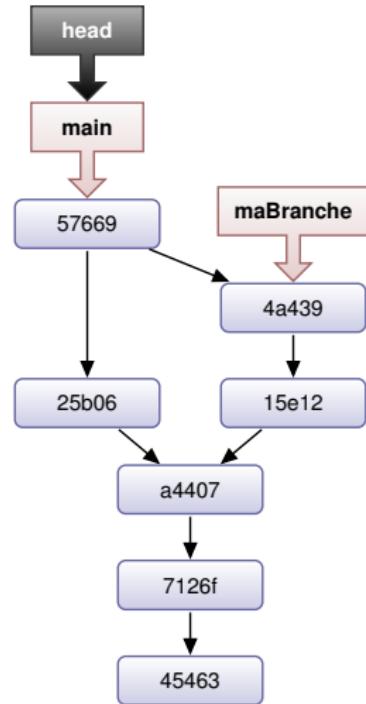
Merge : exemple sur deux branches distinctes.

```
$ ls  
dir fichier2.txt foo.txt  
  
$ git switch maBranche  
$ ls  
dir fichier1.txt foo.txt  
  
$ echo "titi" > fichier1.txt  
$ git commit -am "New fichier1.txt"  
  
$ git switch main  
  
$ git merge maBranche  
$ ls  
dir fichier1.txt fichier2.txt foo.txt
```



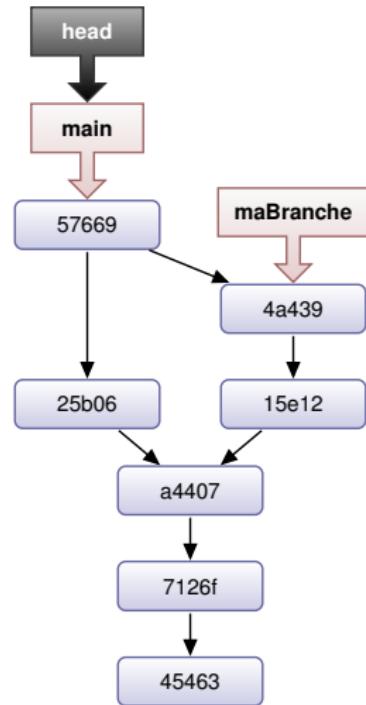
Merge : exemple sur deux branches distinctes.

```
$ ls  
dir fichier2.txt foo.txt  
  
$ git switch maBranche  
$ ls  
dir fichier1.txt foo.txt  
  
$ echo "titi" > fichier1.txt  
$ git commit -am "New fichier1.txt"  
  
$ git switch main  
  
$ git merge maBranche  
$ ls  
dir fichier1.txt fichier2.txt foo.txt
```



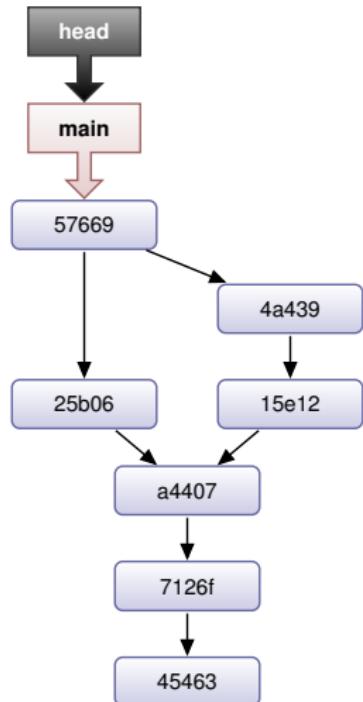
Merge : exemple sur deux branches distinctes.

```
$ ls  
dir fichier2.txt foo.txt  
  
$ git switch maBranche  
$ ls  
dir fichier1.txt foo.txt  
  
$ echo "titi" > fichier1.txt  
$ git commit -am "New fichier1.txt"  
  
$ git switch main  
  
$ git merge maBranche  
$ ls  
dir fichier1.txt fichier2.txt foo.txt  
  
$ git branch -d maBranche
```



Merge : exemple sur deux branches distinctes.

```
$ ls  
dir fichier2.txt foo.txt  
  
$ git switch maBranche  
$ ls  
dir fichier1.txt foo.txt  
  
$ echo "titi" > fichier1.txt  
$ git commit -am "New fichier1.txt"  
  
$ git switch main  
  
$ git merge maBranche  
$ ls  
dir fichier1.txt fichier2.txt foo.txt  
  
$ git branch -d maBranche
```



Outline

Git c'est quoi ?

Architecture interne de git

Gestion des versions dans le dépôt local.

Les commits

Les branches

Les merges

Les rebases

Les remords

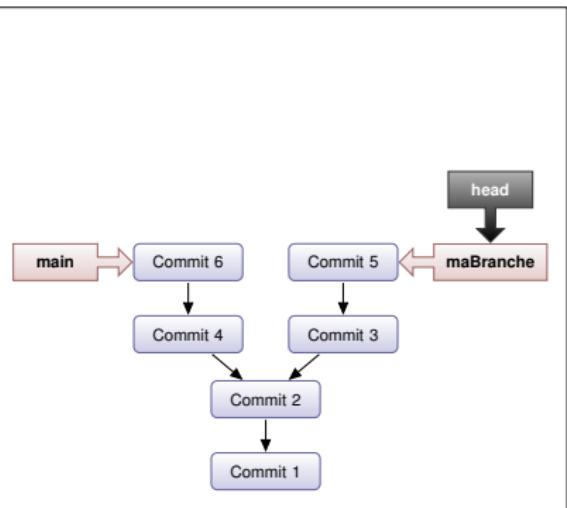
Utilisation de l'historique

Synchronisation avec les dépôts distants.

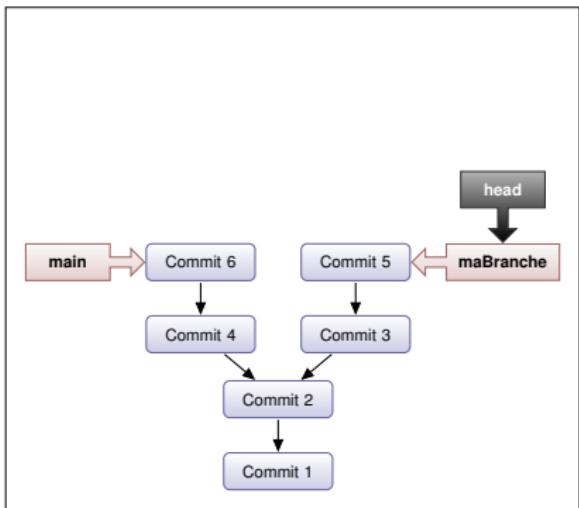
Les outils graphiques

Conclusion

Rebase vs Merge.

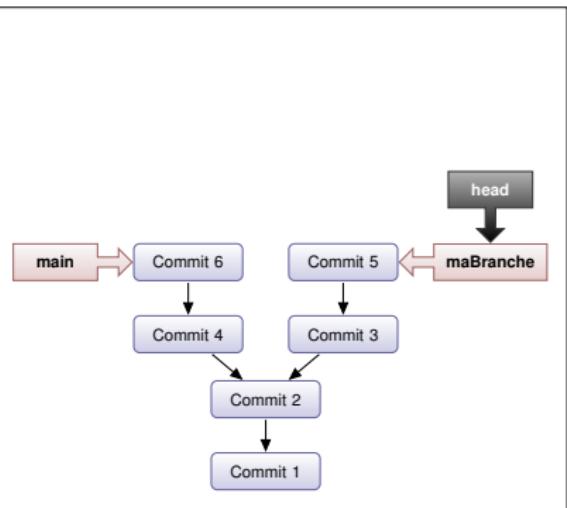


```
$ git switch maBranche  
$ git merge main
```

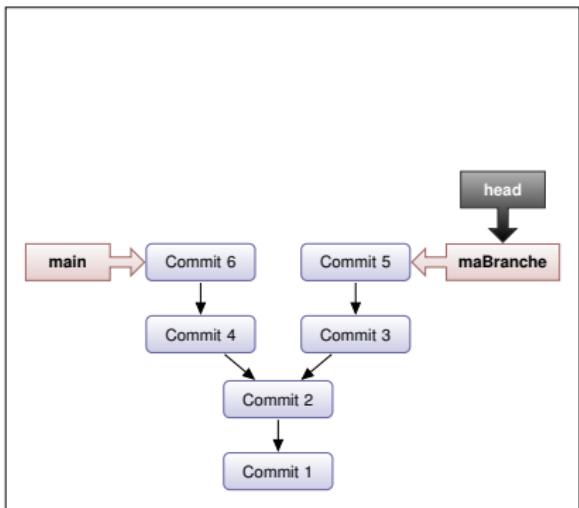


```
$ git switch maBranche  
$ git rebase main
```

Rebase vs Merge.

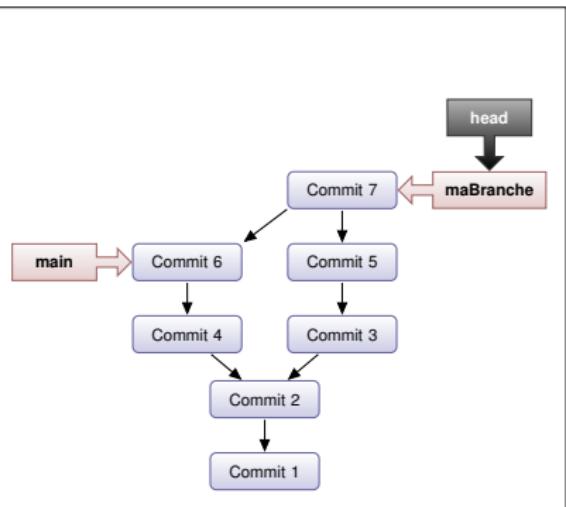


```
$ git switch maBranche  
$ git merge main
```

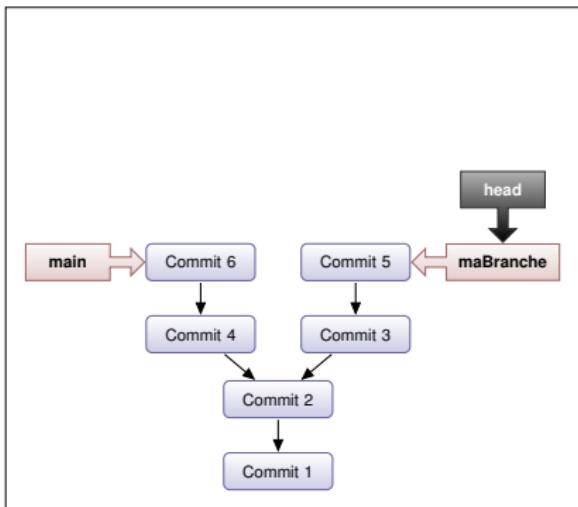


```
$ git switch maBranche  
$ git rebase main
```

Rebase vs Merge.

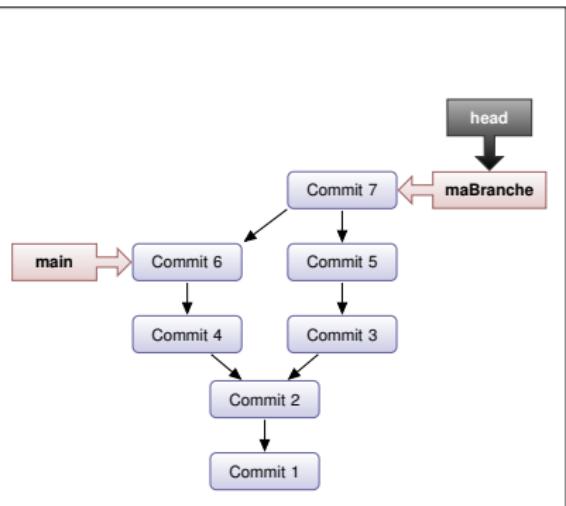


```
$ git switch maBranche  
$ git merge main
```

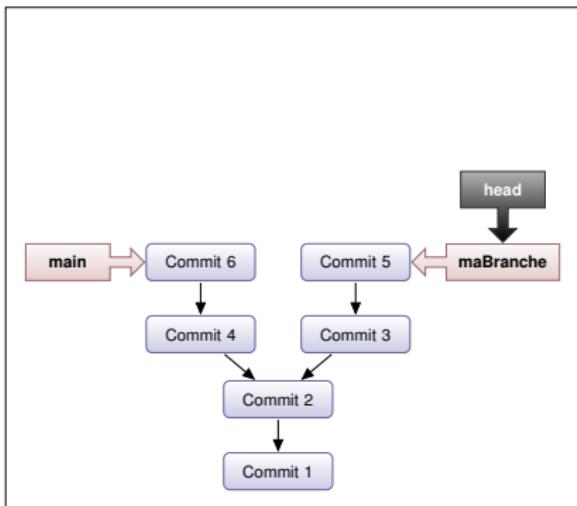


```
$ git switch maBranche  
$ git rebase main
```

Rebase vs Merge.

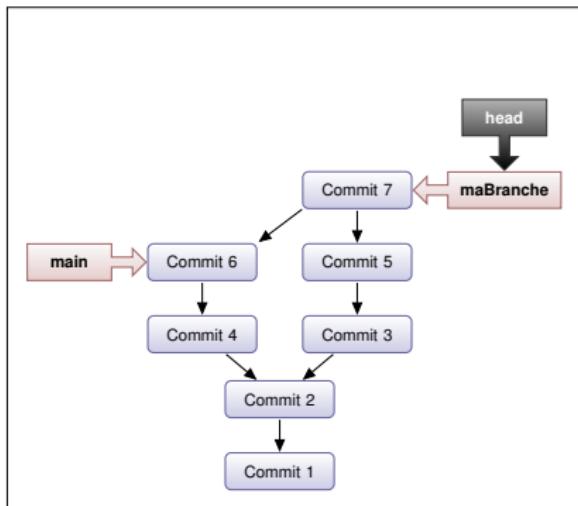


```
$ git switch maBranche  
$ git merge main
```

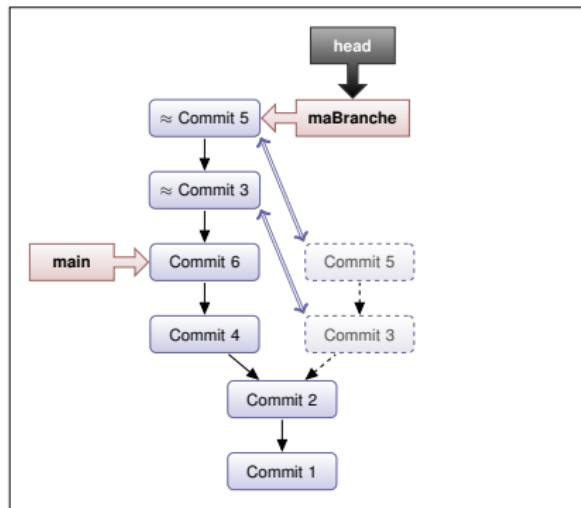


```
$ git switch maBranche  
$ git rebase main
```

Rebase vs Merge.

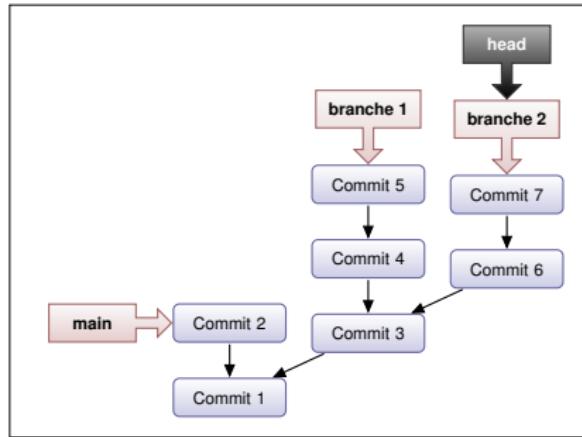


```
$ git switch maBranche  
$ git merge main
```



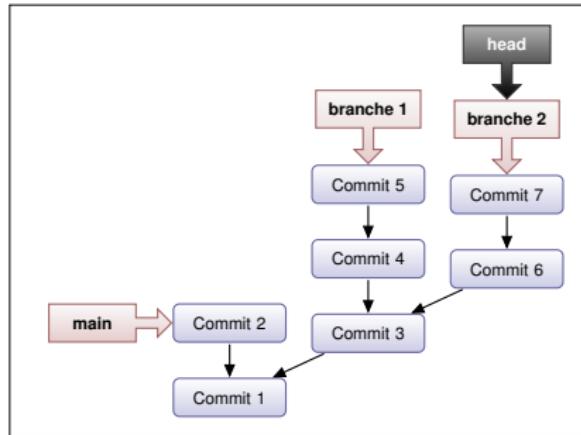
```
$ git switch maBranche  
$ git rebase main
```

Rebase vs Merge.



```
$ git switch branche2
```

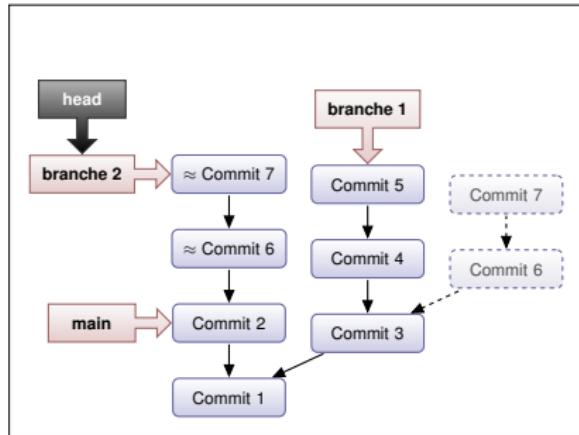
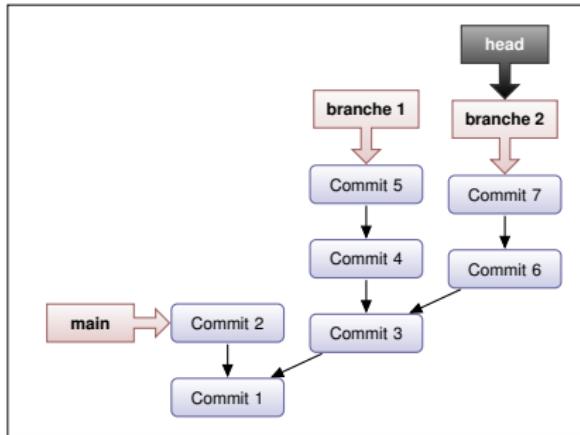
Rebase vs Merge.



```
$ git switch branche2
```

```
$ git rebase --onto main branchel branche2
```

Rebase vs Merge.



```
$ git switch branche2
```

```
$ git rebase --onto main branchel branche2
```

Outline

Git c'est quoi ?

Architecture interne de git

Gestion des versions dans le dépôt local.

Les commits

Les branches

Les merges

Les rebases

Les remords

Utilisation de l'historique

Synchronisation avec les dépôts distants.

Les outils graphiques

Conclusion

Les remords

En cas d'erreur sur un commit, git propose 3 types de correction :

revert : pour annuler un commit par un autre commit.

amend : modifier le dernier commit.

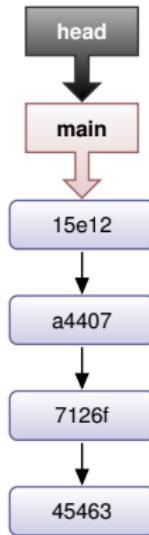
reset : pour rétablir la situation d'un ancien commit.

Attention

Si l'erreur a déjà été rendue publique, la seule bonne pratique est le **revert**. Les autres solutions peuvent conduire à des incohérences.

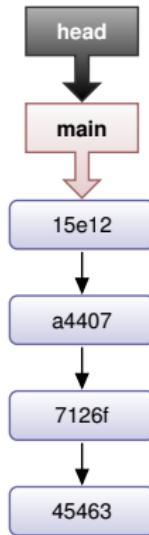
Amend : modification du dernier commit.

```
$ ls  
foo.txt dir
```



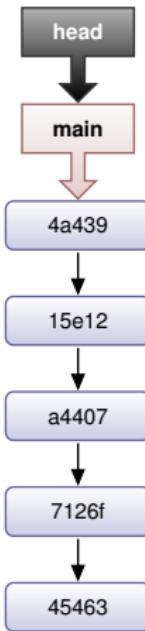
Amend : modification du dernier commit.

```
$ ls  
foo.txt dir  
  
$ touch bar.txt  
$ git commit -m "Ajout d'un fichier."
```



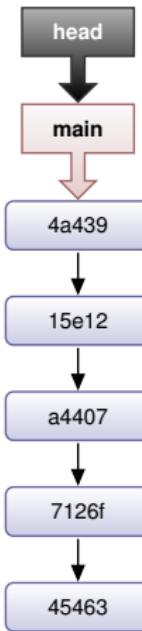
Amend : modification du dernier commit.

```
$ ls  
foo.txt dir  
  
$ touch bar.txt  
$ git commit -m "Ajout d'un fichier."
```



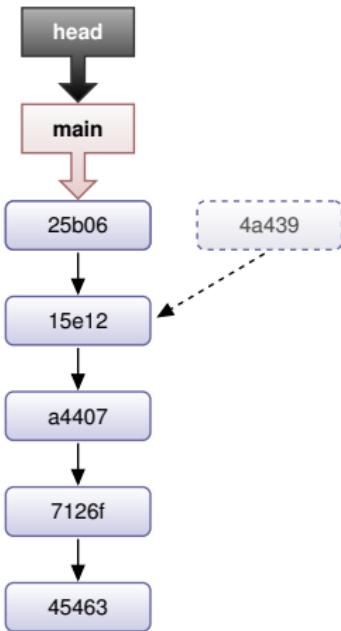
Amend : modification du dernier commit.

```
$ ls  
foo.txt dir  
  
$ touch bar.txt  
$ git commit -m "Ajout d'un fichier."  
  
$ git add bar.txt  
$ git commit --amend -m "Ajout d'un fichier."
```



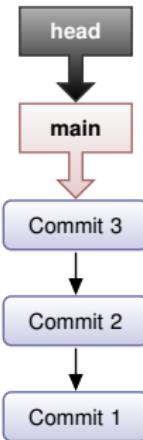
Amend : modification du dernier commit.

```
$ ls  
foo.txt dir  
  
$ touch bar.txt  
$ git commit -m "Ajout d'un fichier."  
  
$ git add bar.txt  
$ git commit --amend -m "Ajout d'un fichier."
```



Git revert : annulation par commit.

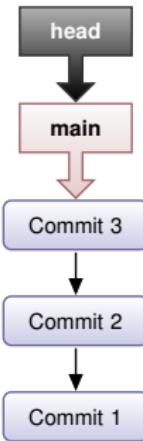
```
$ git branch main
$ cat fichier1.txt
Premiere version de F1
$ cat fichier2.txt
Premiere version de F2
```



Git revert : annulation par commit.

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

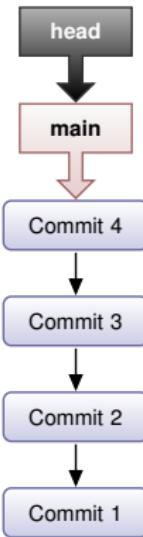
$ echo "Deuxieme version de F1" > fichier1.txt
$ git add fichier1.txt
$ git commit -m "Add fichier1.txt"
```



Git revert : annulation par commit.

```
$ git branch main
$ 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"
```

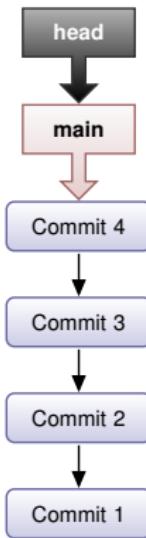


Git revert : annulation par commit.

```
$ git branch main
$ 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"
```

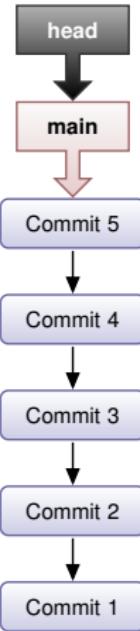


Git revert : annulation par commit.

```
$ git branch main
$ 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"
```



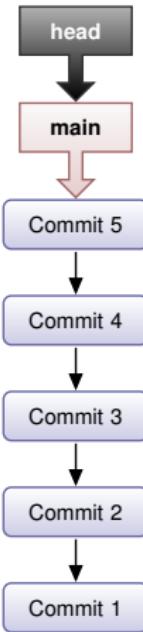
Git revert : annulation par commit.

```
$ git branch main
$ 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
```



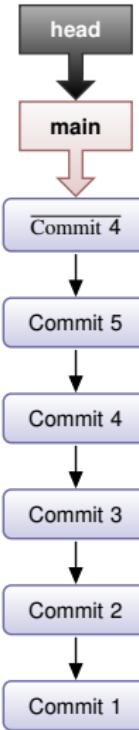
Git revert : annulation par commit.

```
$ git branch main
$ 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
```



Git reset : "suppression" de commit.

git reset permet de "supprimer" un ou plusieurs commits.

Dans les faits, les objets liés à ces commits ne seront vraiment effacés qu'après un appel à **git gc** et s'ils sont suffisamment vieux (15 jours par défaut).

Il existe trois types de reset :

1. **git reset --hard** :

- ▶ restore la référence du commit (de la branche active)
- ▶ restore l'index
- ▶ restore les données

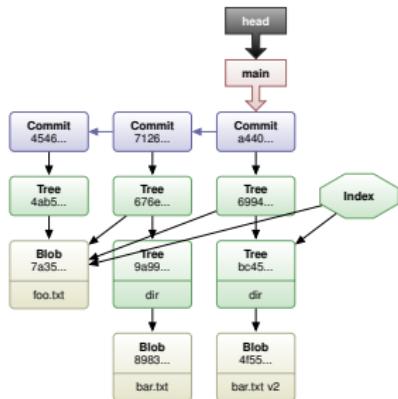
2. **git reset** :

- ▶ restore la référence du commit (de la branche active)
- ▶ restore l'index

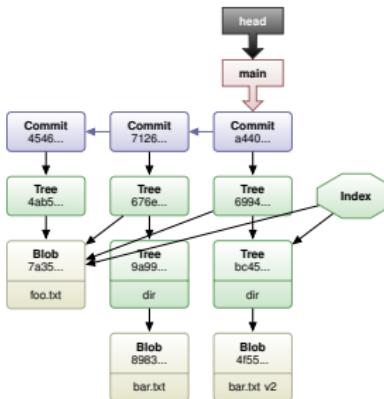
3. **git reset --soft** :

- ▶ restore la référence du commit (de la branche active)

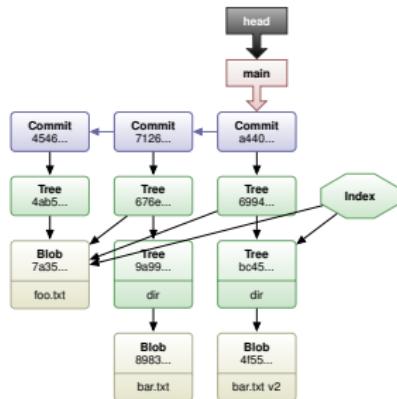
Les différents types de reset.



```
$ ls -R
.: foo.txt
dir: bar.txt
```



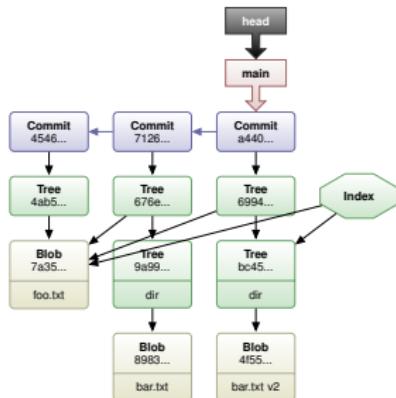
```
$ ls -R
.: foo.txt
dir: bar.txt
```



```
$ ls -R
.: foo.txt
dir: bar.txt
```

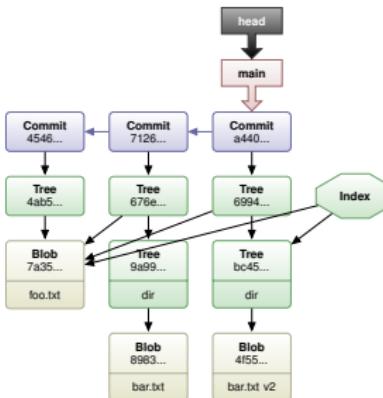
```
$ ls -R
.: foo.txt
dir: bar.txt
```

Les différents types de reset.

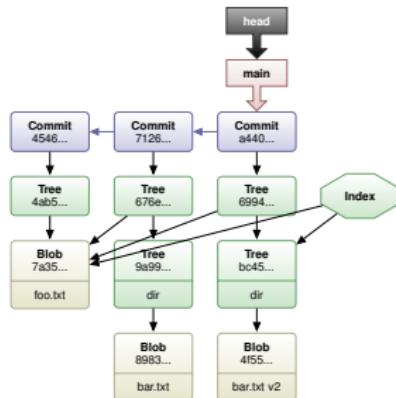


```
$ ls -R
.: foo.txt
dir: bar.txt

$ git reset --hard
4546
```



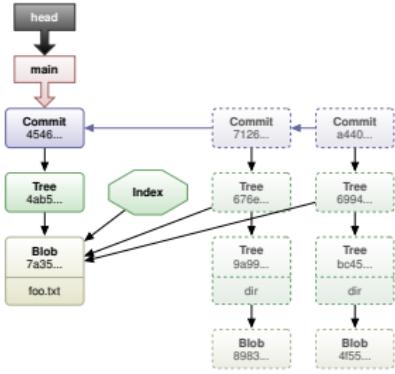
```
$ ls -R
.: foo.txt
dir: bar.txt
```



```
$ ls -R
.: foo.txt
dir: bar.txt
```

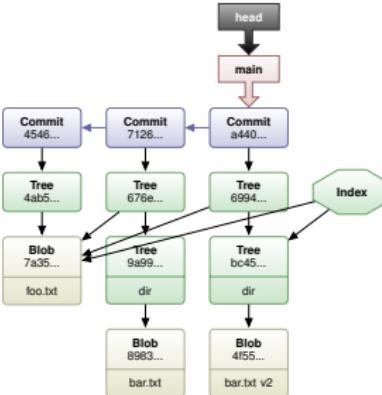
```
$ ls -R
.: foo.txt
dir: bar.txt
```

Les différents types de reset.

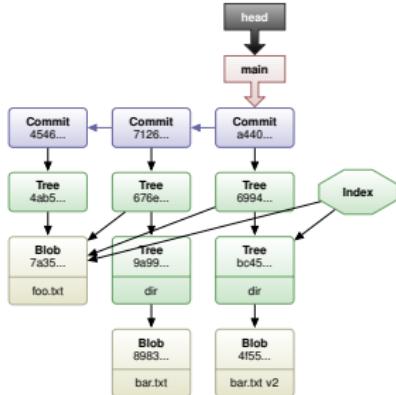


```
$ ls -R
.: foo.txt
dir: bar.txt

$ git reset --hard
4546
```

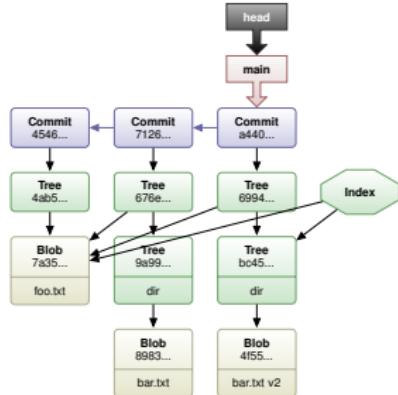
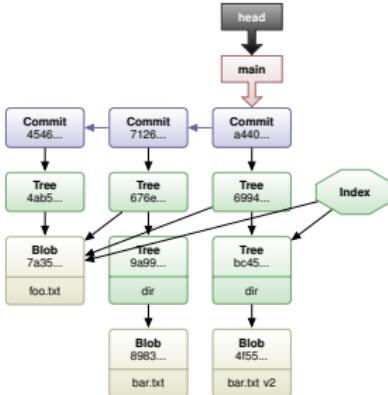
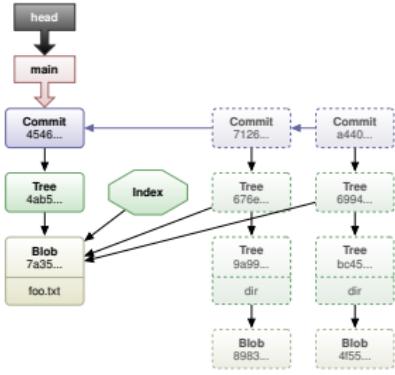


```
$ ls -R
.: foo.txt
dir: bar.txt
```



```
$ ls -R
.: foo.txt
dir: bar.txt
```

Les différents types de reset.



```
$ ls -R
.: foo.txt
dir: bar.txt

$ git reset --hard
4546

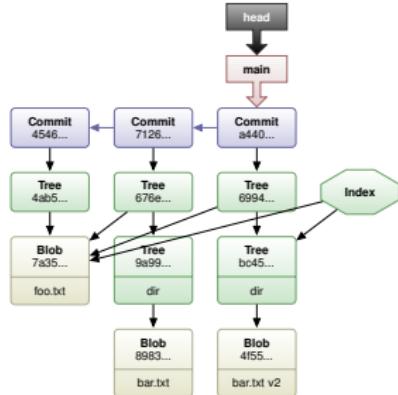
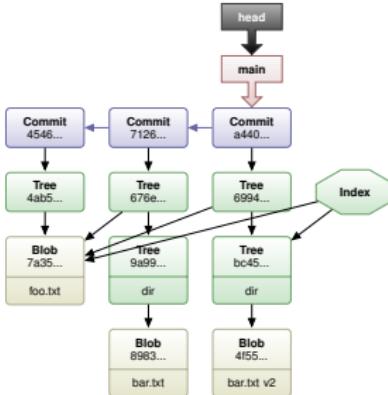
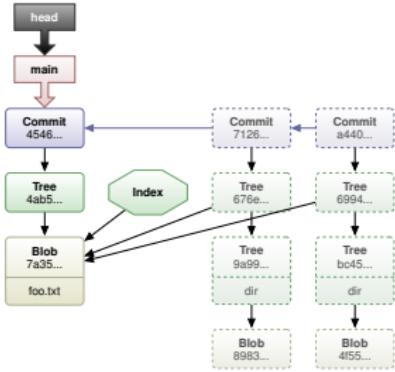
$ ls -R
.: foo.txt
```

```
$ ls -R
.: foo.txt
dir: bar.txt
```

```
$ ls -R
.: foo.txt
dir: bar.txt

$ ls -R
.: foo.txt
dir: bar.txt
```

Les différents types de reset.



```
$ ls -R
.: foo.txt
dir: bar.txt

$ git reset --hard
4546

$ ls -R
.: foo.txt
```

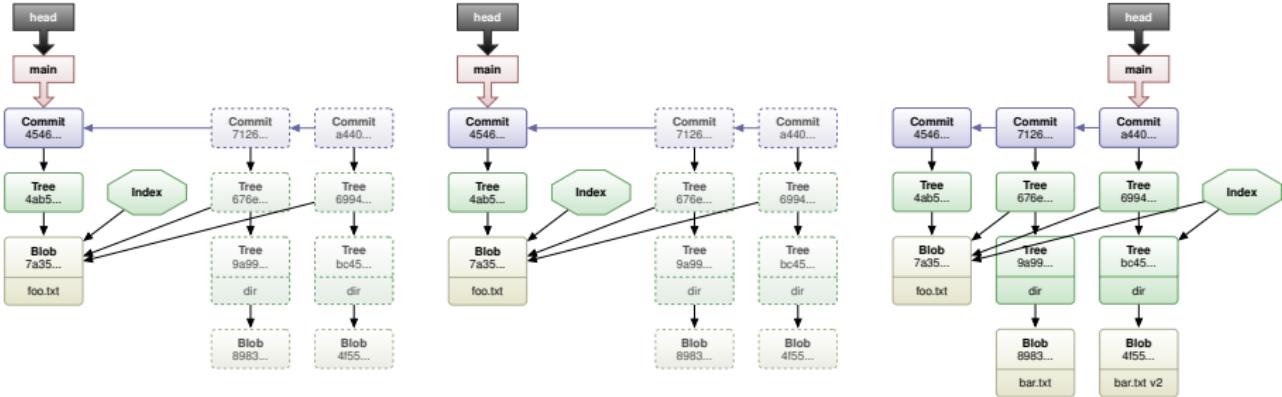
```
$ ls -R
.: foo.txt
dir: bar.txt

$ git reset 4546
```

```
$ ls -R
.: foo.txt
dir: bar.txt

$ ls -R
.: foo.txt
dir: bar.txt
```

Les différents types de reset.



```
$ ls -R
.: foo.txt
dir: bar.txt

$ git reset --hard
4546

$ ls -R
.: foo.txt
```

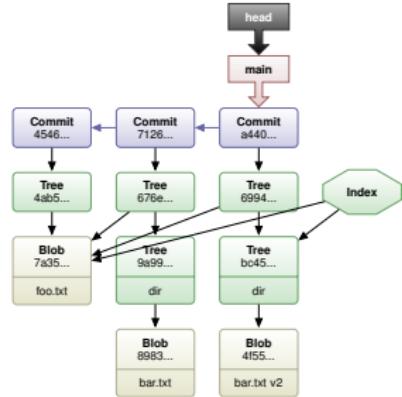
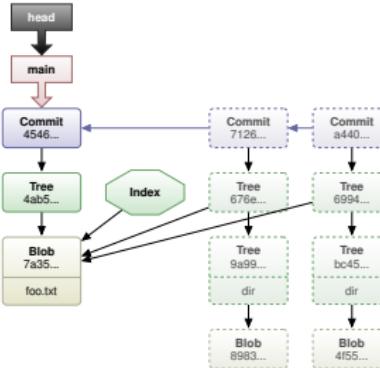
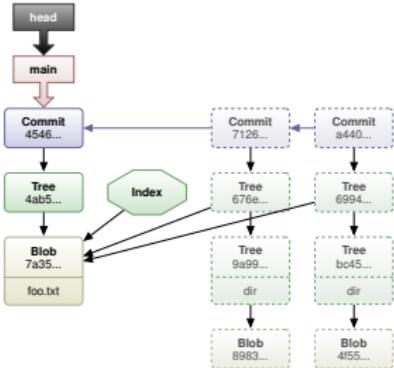
```
$ ls -R
.: foo.txt
dir: bar.txt

$ git reset 4546
```

```
$ ls -R
.: foo.txt
dir: bar.txt

$ ls -R
.: foo.txt
dir: bar.txt
```

Les différents types de reset.



```
$ ls -R  
.: foo.txt  
dir: bar.txt
```

```
$ git reset --hard  
4546
```

```
$ ls -R  
.: foo.txt
```

```
$ ls -R  
.: foo.txt  
dir: bar.txt
```

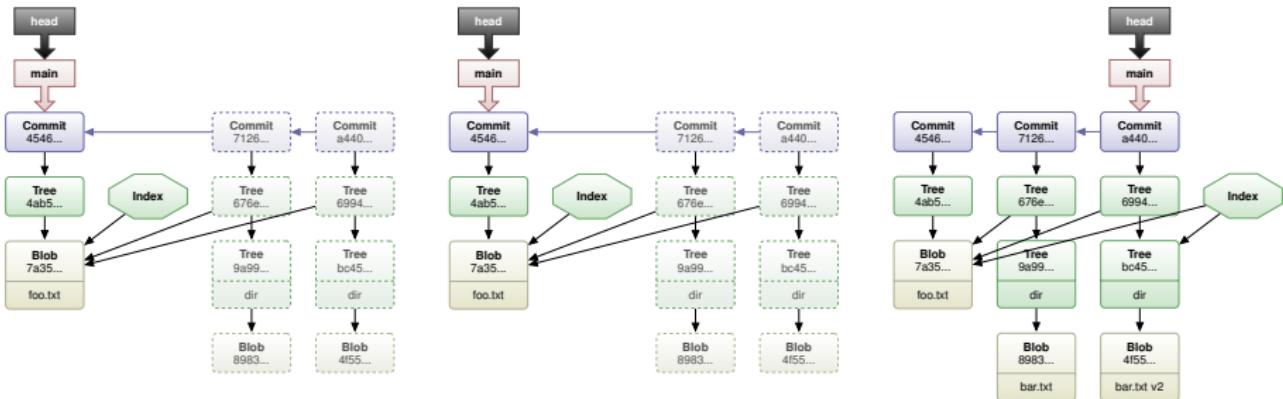
```
$ git reset 4546
```

```
$ ls -R  
.: foo.txt  
dir: bar.txt
```

```
$ ls -R  
.: foo.txt  
dir: bar.txt
```

```
$ ls -R  
.: foo.txt  
dir: bar.txt
```

Les différents types de reset.



```
$ ls -R
.: foo.txt
dir: bar.txt
```

```
$ git reset --hard
4546
```

```
$ ls -R
.: foo.txt
```

```
$ ls -R
.: foo.txt
dir: bar.txt
```

```
$ git reset 4546
```

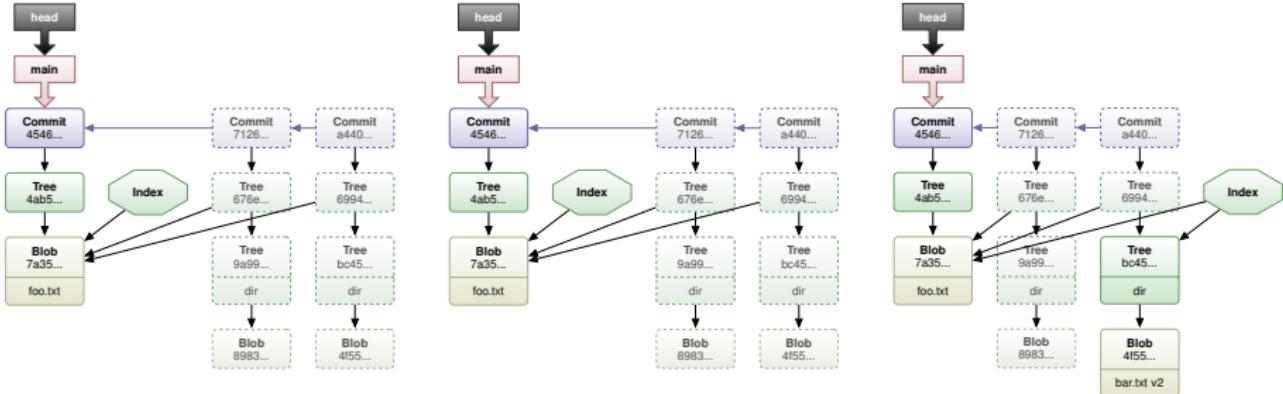
```
$ ls -R
.: foo.txt
dir: bar.txt
```

```
$ ls -R
.: foo.txt
dir: bar.txt
```

```
$ git reset --soft
4546
```

```
$ ls -R
.: foo.txt
dir: bar.txt
```

Les différents types de reset.



```
$ ls -R
.: foo.txt
dir: bar.txt
```

```
$ git reset --hard
4546
```

```
$ ls -R
.: foo.txt
```

```
$ ls -R
.: foo.txt
dir: bar.txt
```

```
$ git reset 4546
```

```
$ ls -R
.: foo.txt
dir: bar.txt
```

```
$ ls -R
.: foo.txt
dir: bar.txt
```

```
$ git reset --soft
4546
```

```
$ ls -R
.: foo.txt
dir: bar.txt
```

Outline

Git c'est quoi ?

Architecture interne de git

Gestion des versions dans le dépôt local.

Les commits

Les branches

Les merges

Les rebases

Les remords

Utilisation de l'historique

Synchronisation avec les dépôts distants.

Les outils graphiques

Conclusion

Comparaison : git diff

- ▶ Différences entre le répertoire de travail et l'index :

```
$ git diff
```

- ▶ Différences entre HEAD et l'index :

```
$ git diff --staged
```

- ▶ Différences entre répertoire de travail et HEAD :

```
$ git diff HEAD
```

- ▶ Différences entre répertoire de travail et un autre commit :

```
$ git diff <commit_1>
```

- ▶ Différences entre deux commit :

Information sur un commit : git show

```
$ git show  
commit 7a618a3f3c23262ad281c9afe69e145ef867d43b  
Author: Julien SOPENA <julien.sopena@lip6.fr>  
Date: Mon Oct 25 03:55:27 2010 +0200  
  
Ceci est un petit exemple de commit.  
  
diff -git a/test b/test  
index 808a2c4..99810fa 100644  
-- a/test  
+++ b/test  
@@ -1,3 +1,3 @@  
Ligne de texte non modifié par ce commit.  
-Ligne de texte supprimée.  
+Nouvelle ligne de texte  
Suite du texte non modifié.
```

Outline

Git c'est quoi ?

Architecture interne de git

Gestion des versions dans le dépôt local.

Synchronisation avec les dépôts distants.

Principe des DVCS : Gestionnaire de versions décentralisé.

Les dépôts distants.

Modèles de travail coopératif

Les outils graphiques

Conclusion

Outline

Git c'est quoi ?

Architecture interne de git

Gestion des versions dans le dépôt local.

Synchronisation avec les dépôts distants.

Principe des DVCS : Gestionnaire de versions décentralisé.

Les dépôts distants.

Modèles de travail coopératif

Les outils graphiques

Conclusion

Dépôts centralisés

CVS ou Subversion



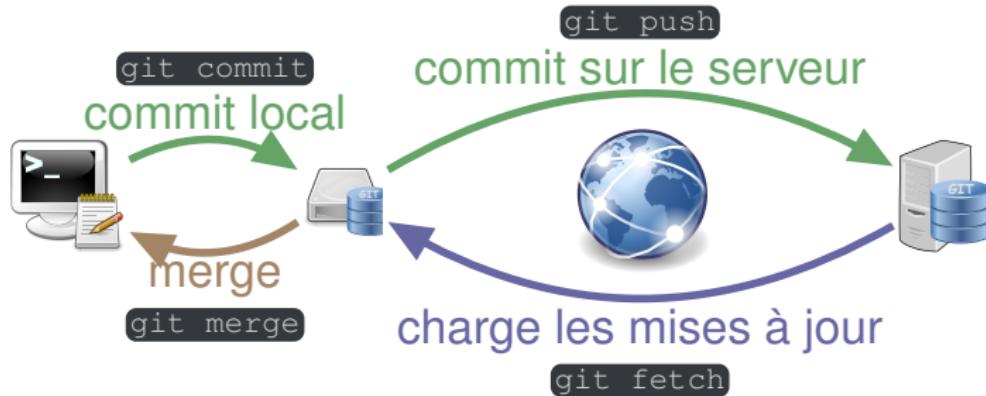
Dépôts centralisés

CVS ou Subversion

- ▶ Toutes les versions des fichiers sont entreposées dans un unique dépôt accessible aux clients autorisés
- ▶ Les clients ne travaillent que sur une partie des données, en général, une simple branche
- ▶ Chaque changement de branche nécessite un téléchargement de l'ensemble des données de la branche.
- ▶ Impossibilité de versionner *Off-line*
- ▶ Inclure un contributeur nécessite d'ouvrir le dépôt en écriture.

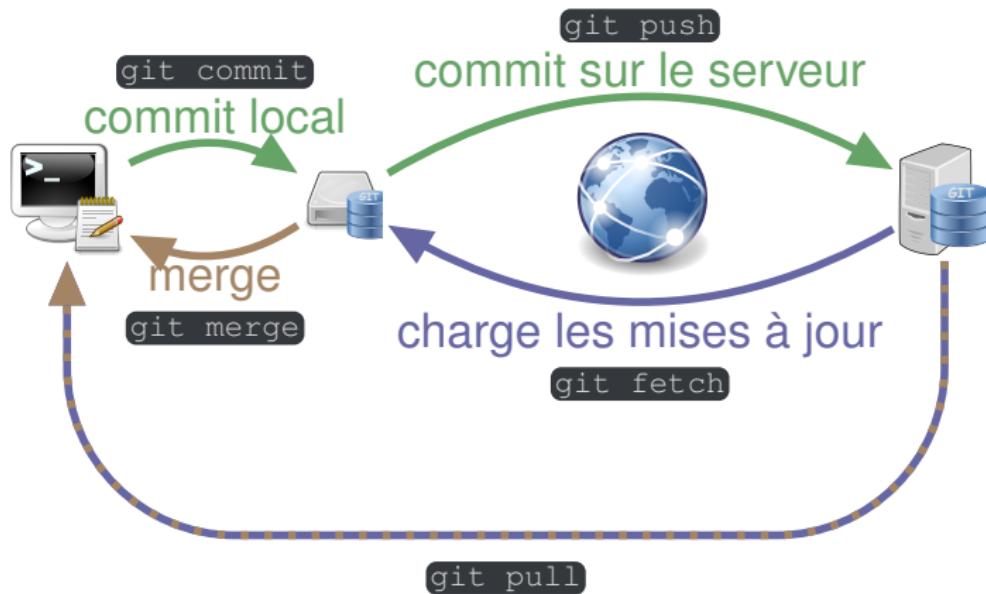
Dépôts décentralisés

Arch, Bazaar-Ng, Git, monotone etc.



Dépôts décentralisés

Arch, Bazar-Ng, Git, monotone etc.



Dépôts décentralisés

Arch, Bazar-Ng, Git, monotone etc.

- ▶ Chaque client a l'ensemble des fichiers dans son dépôt local :
 - ▶ On peut travailler *Off-Line* (à la plage, à la montagne,...)
 - ▶ Le changement de branche est rapide et est donc, une des méthodes de développement : **utiliser les branches**
- ▶ Les seules actions du client nécessitant un accès au dépôt distant sont :
 - ▶ la mise à jour du dépôt local depuis l'extérieur
 - ▶ l'envoi d'information.
- ▶ **Le client peut versionner en local !**

Conflits

Definition

On appelle **conflits**, toute modification d'un fichier dans un dépôt qui n'a pas été élaborée à partir de la version actuelle du fichier sur ce dépôt, *i.e.*, lorsqu'il y a une modification de ce fichier sur le dépôt pendant son édition.

Apparition sur CVS/Subversion :

- ▶ au téléchargement des modifications : cvs **update** ;
- ▶ au commit avec un refus d'enregistrer sur le dépôt.

Apparition sur Git et les autres :

- ▶ Uniquement lors des **fusions** de branches :
locale/locale, locale/distante ou distante/locale.

Outline

Git c'est quoi ?

Architecture interne de git

Gestion des versions dans le dépôt local.

Synchronisation avec les dépôts distants.

Principe des DVCS : Gestionnaire de versions décentralisé.

Les dépôts distants.

Modèles de travail coopératif

Les outils graphiques

Conclusion

Branches distantes et branches locales

Un projet décentralisé possède deux types de branches :

Definition

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.

Definition

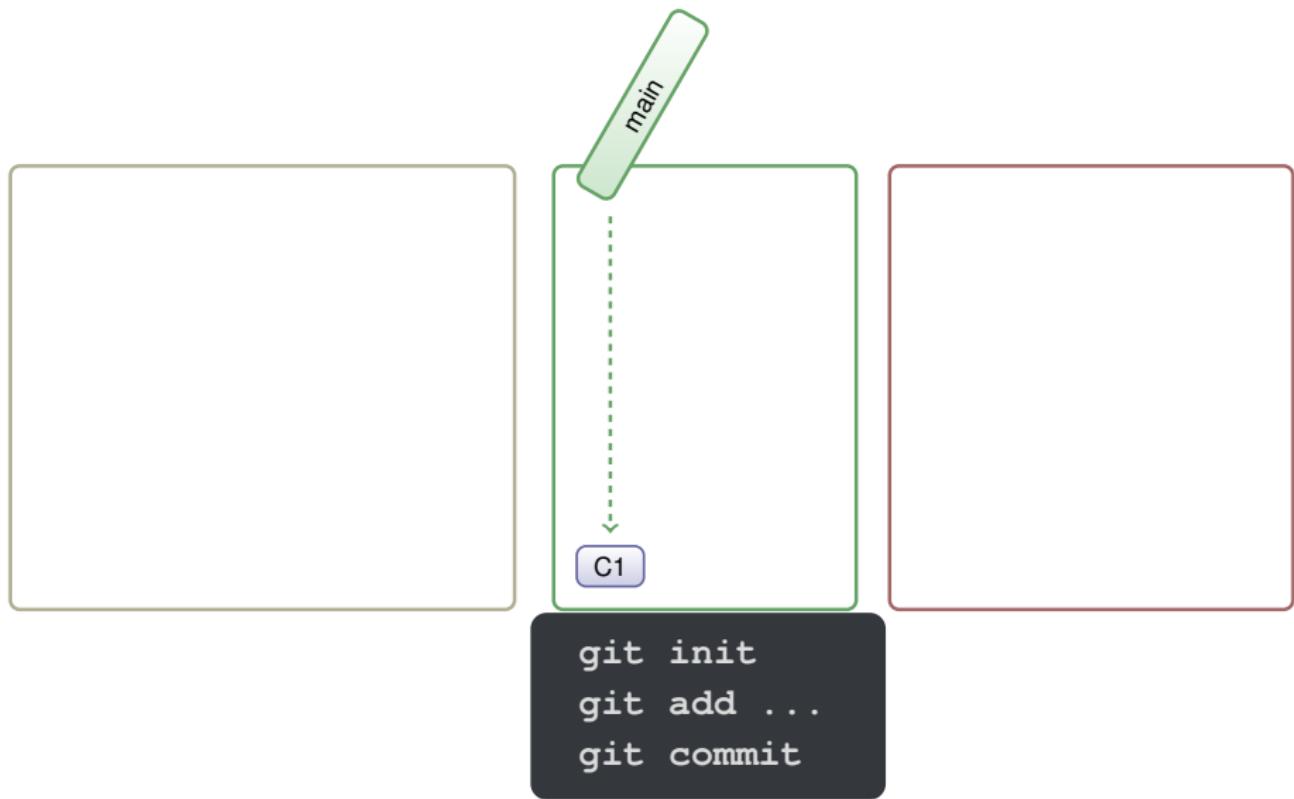
On appelle **branche locale**, une branche propre au dépôt local. Pour être envoyées, les données d'une telle branche doivent être fusionnées avec une branche distante.

Dépôts distants et gestion de la concurrence.

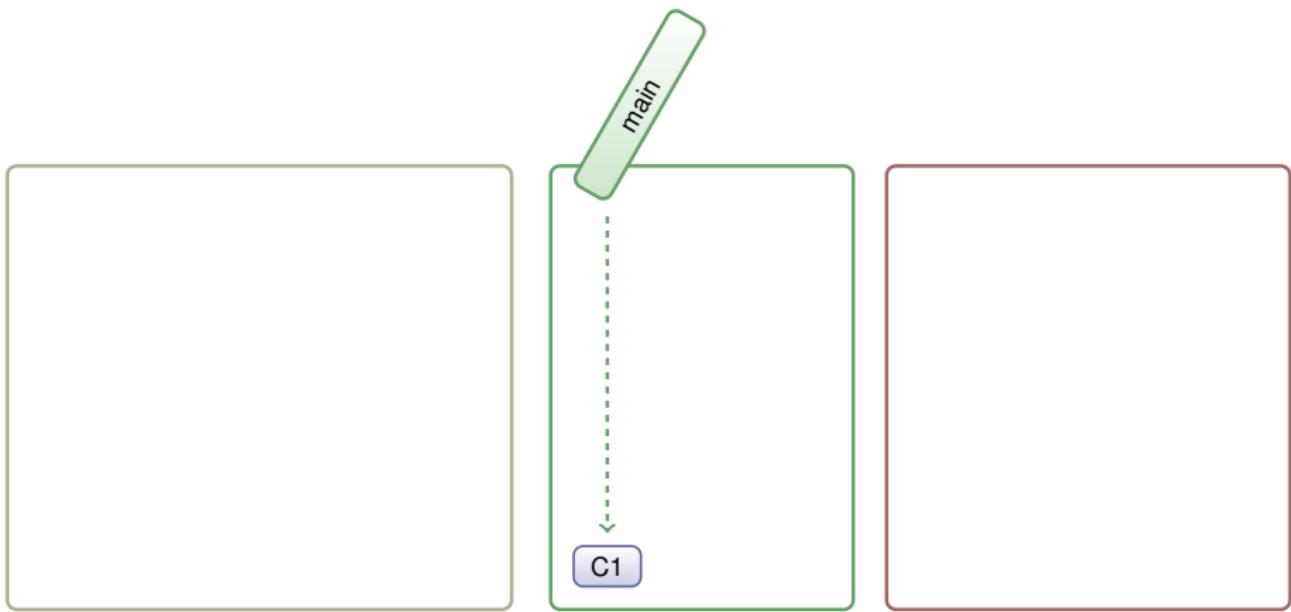


```
git init  
git add ...  
git commit
```

Dépôts distants et gestion de la concurrence.

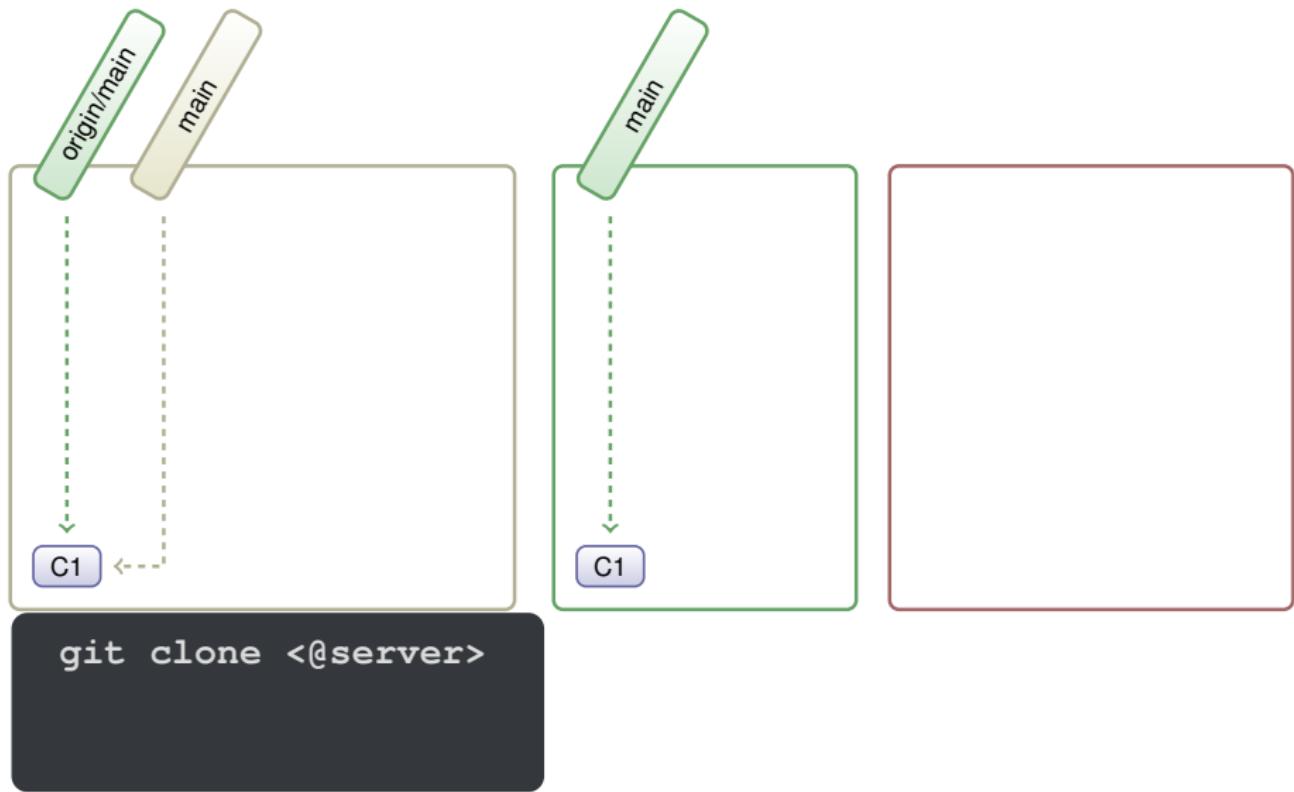


Dépôts distants et gestion de la concurrence.

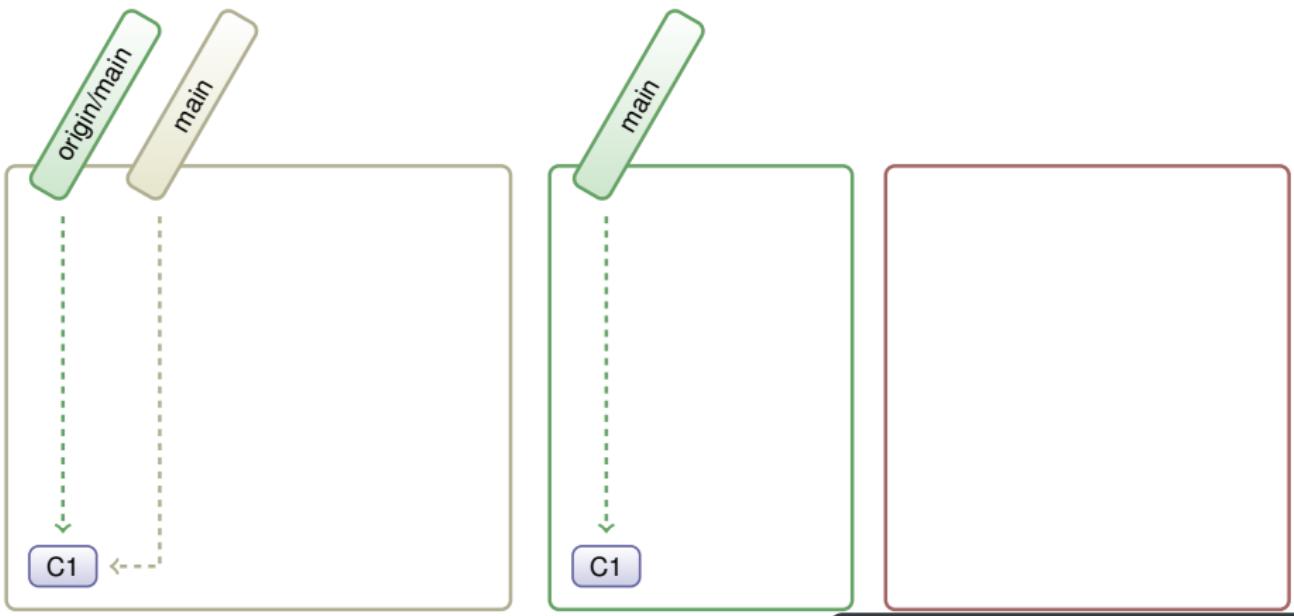


```
git clone <@server>
```

Dépôts distants et gestion de la concurrence.

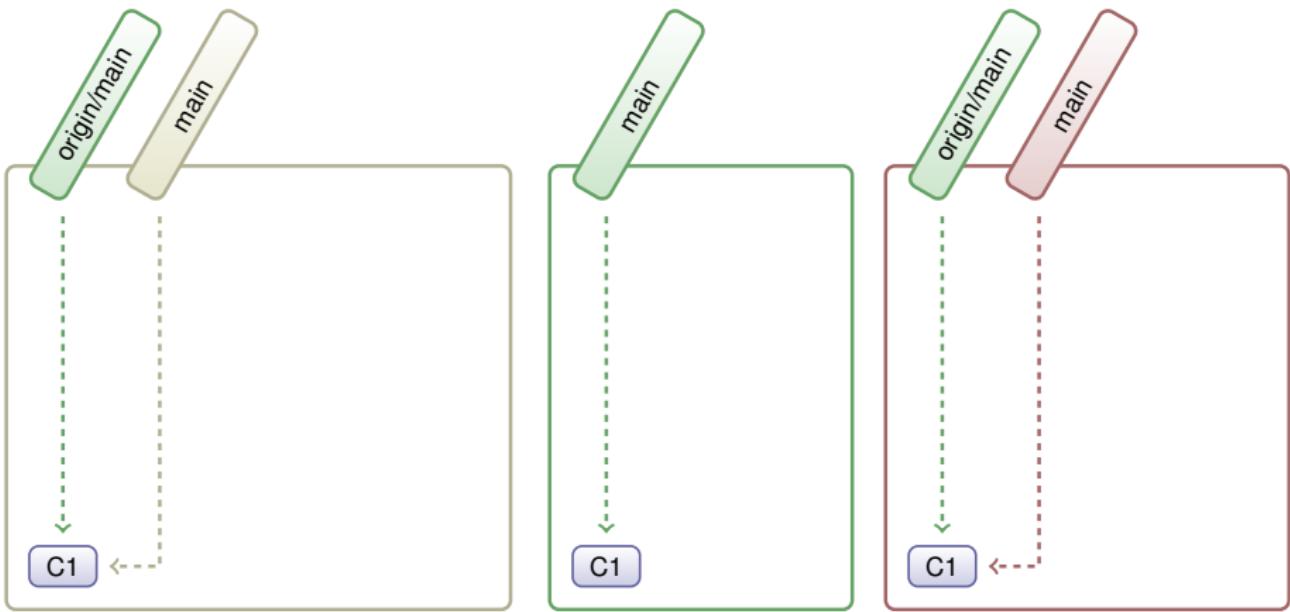


Dépôts distants et gestion de la concurrence.



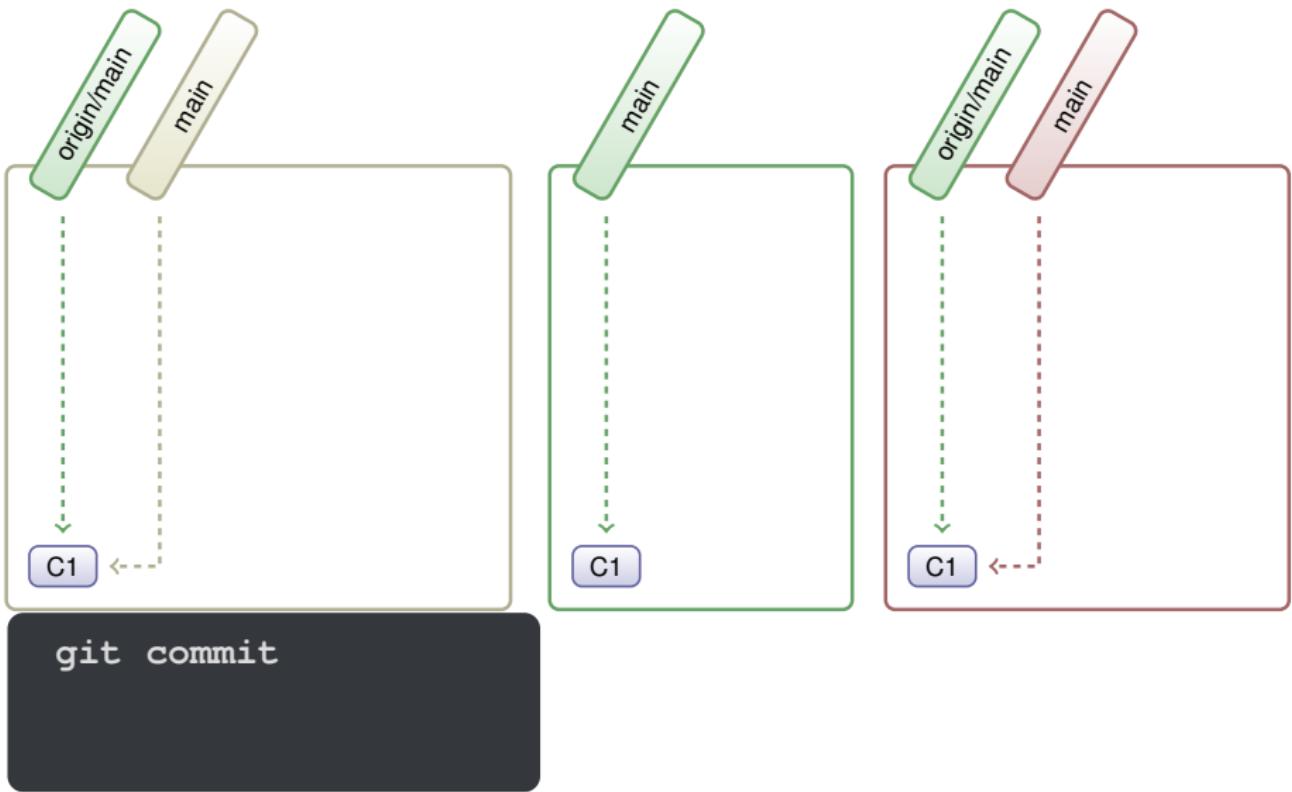
```
git clone <@server>
```

Dépôts distants et gestion de la concurrence.

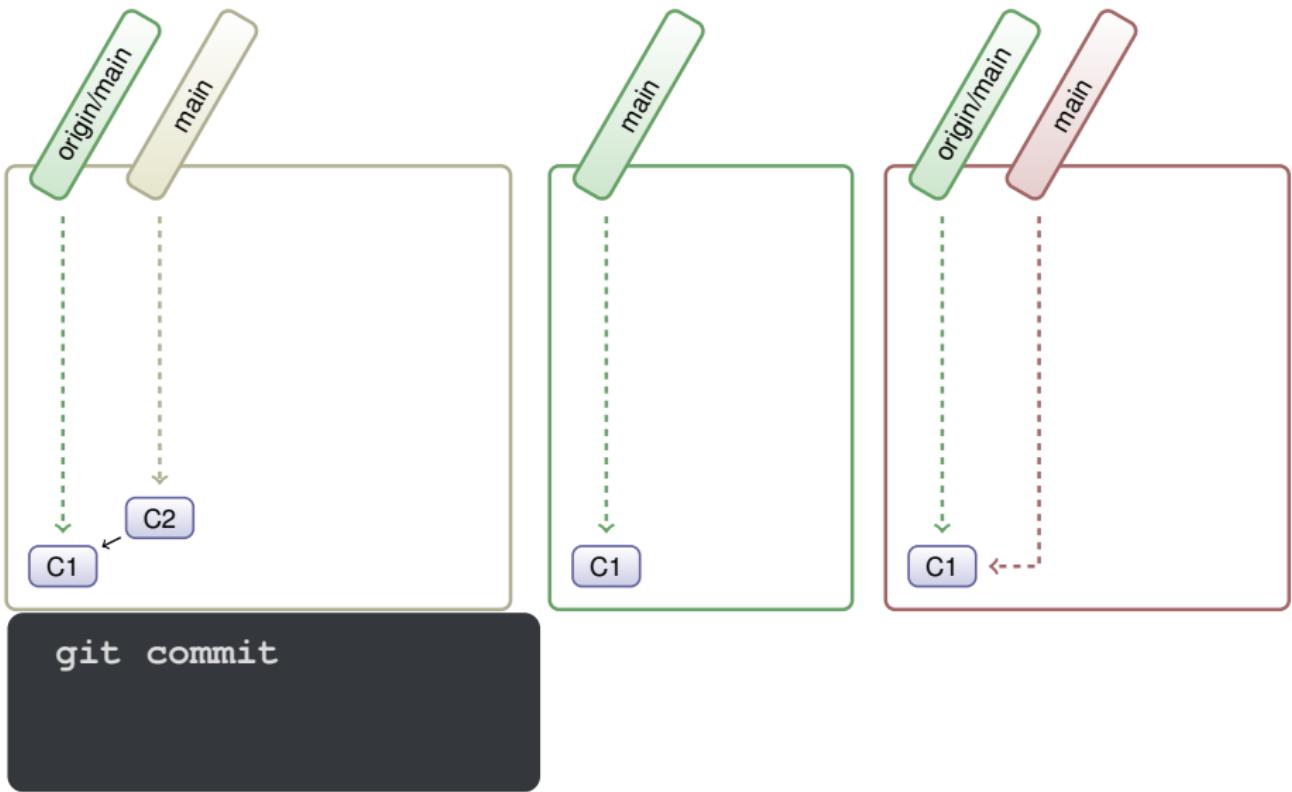


```
git clone <@server>
```

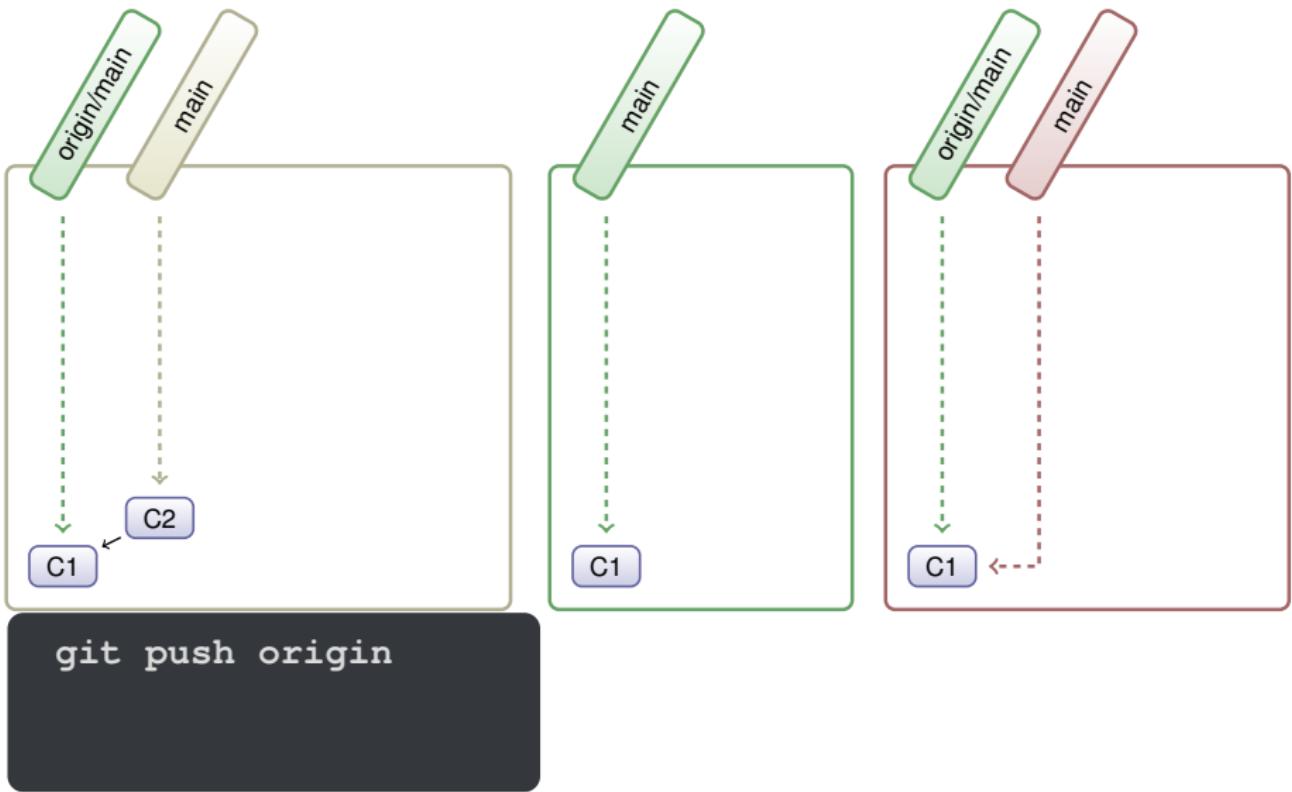
Dépôts distants et gestion de la concurrence.



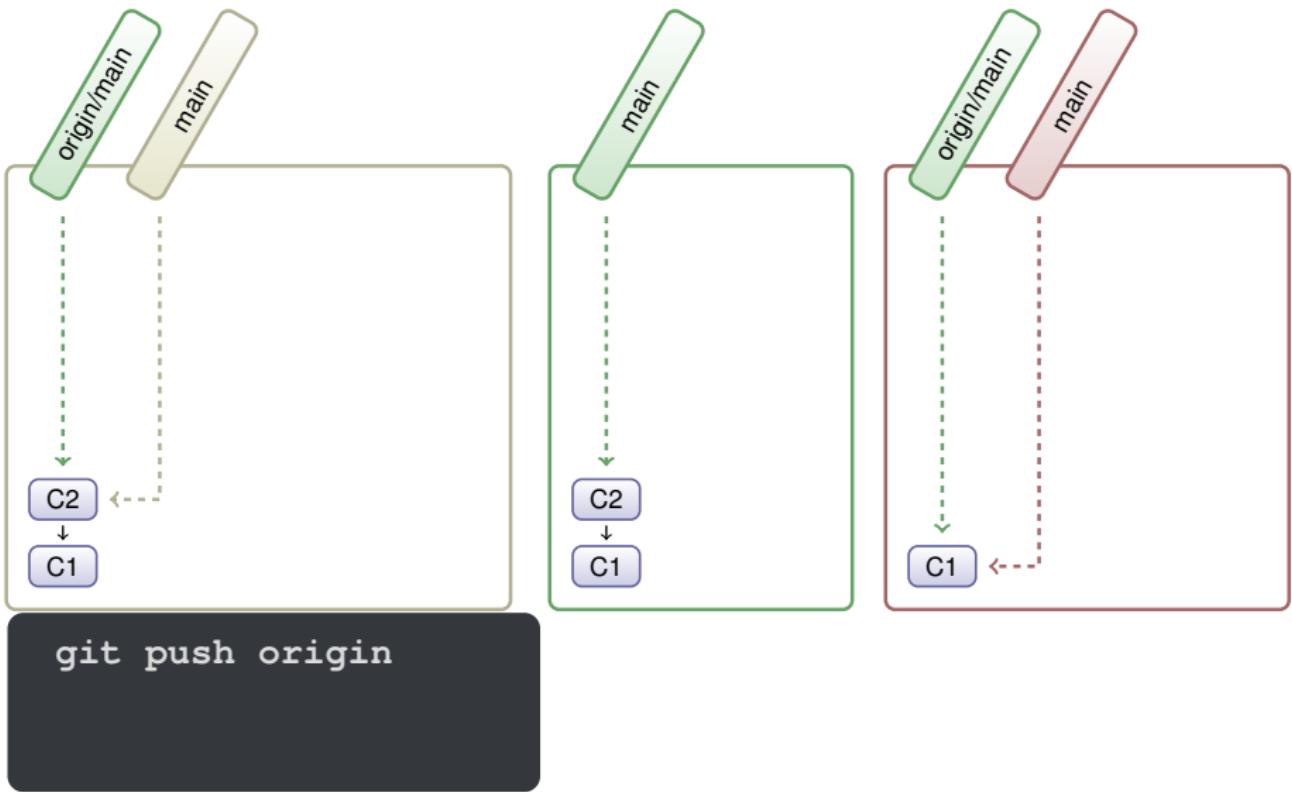
Dépôts distants et gestion de la concurrence.



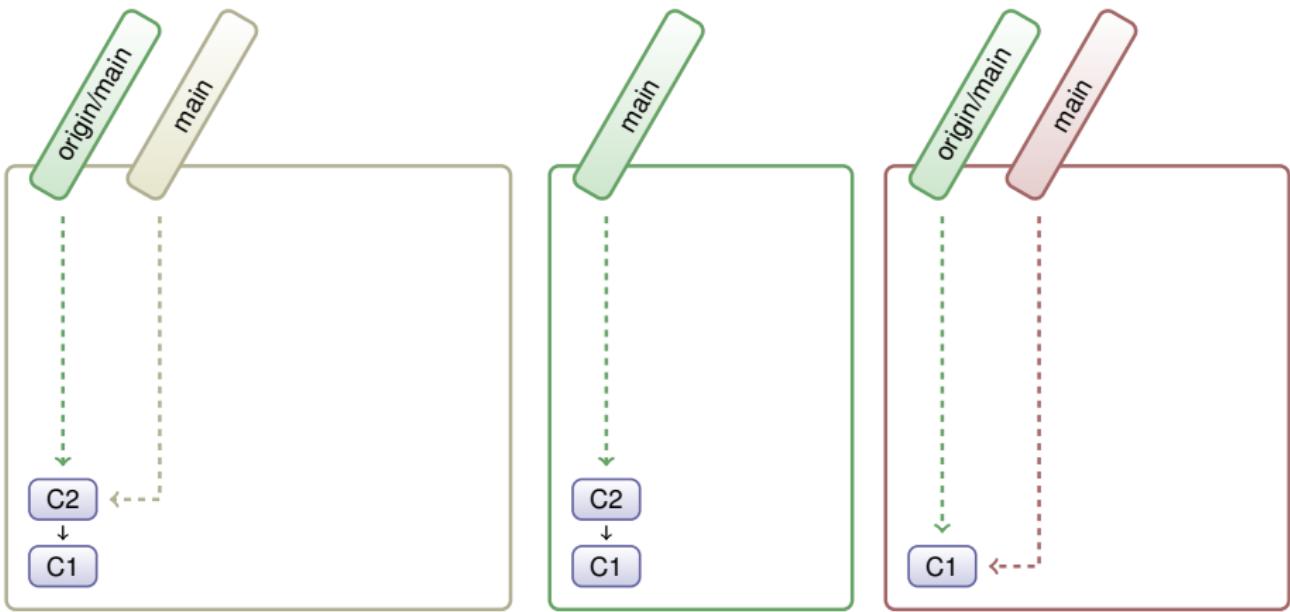
Dépôts distants et gestion de la concurrence.



Dépôts distants et gestion de la concurrence.

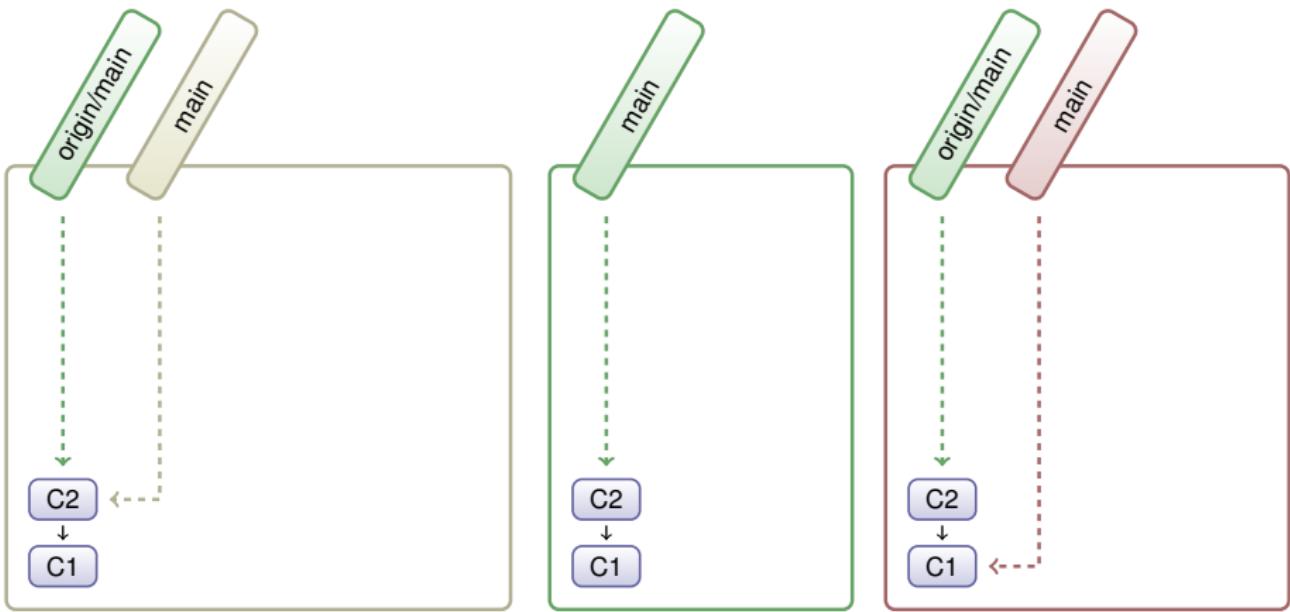


Dépôts distants et gestion de la concurrence.



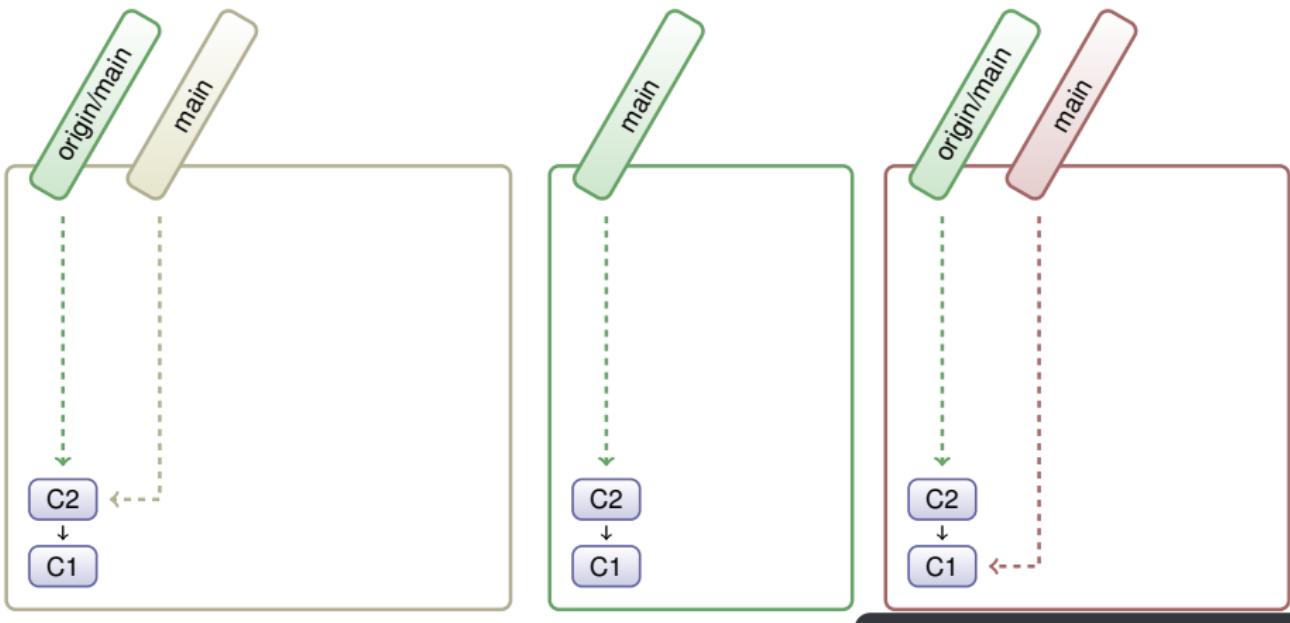
```
git fetch origin
```

Dépôts distants et gestion de la concurrence.



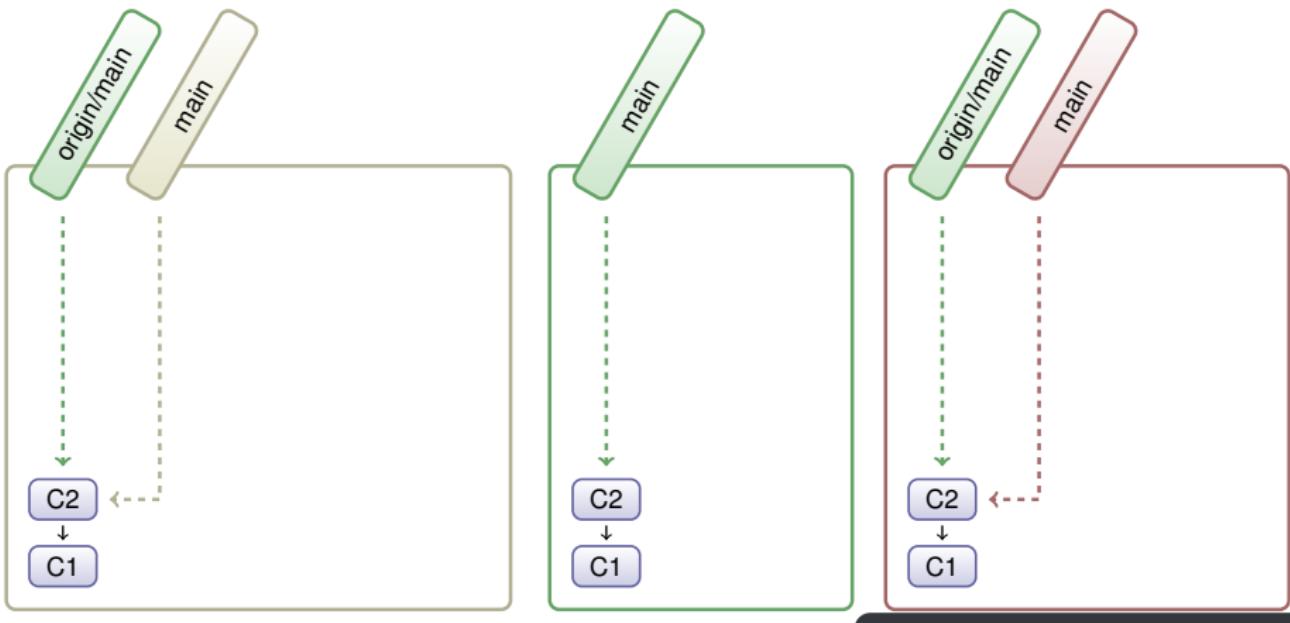
```
git fetch origin
```

Dépôts distants et gestion de la concurrence.

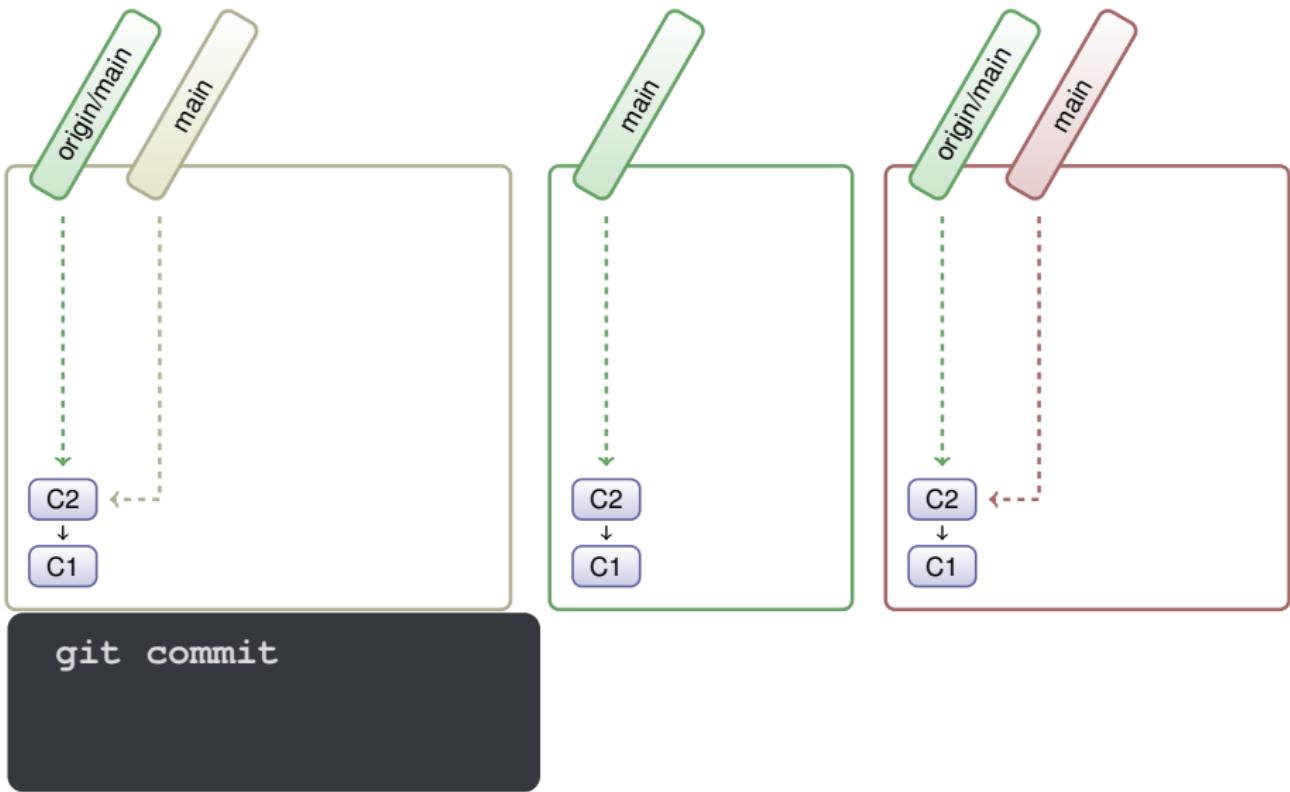


`git merge`

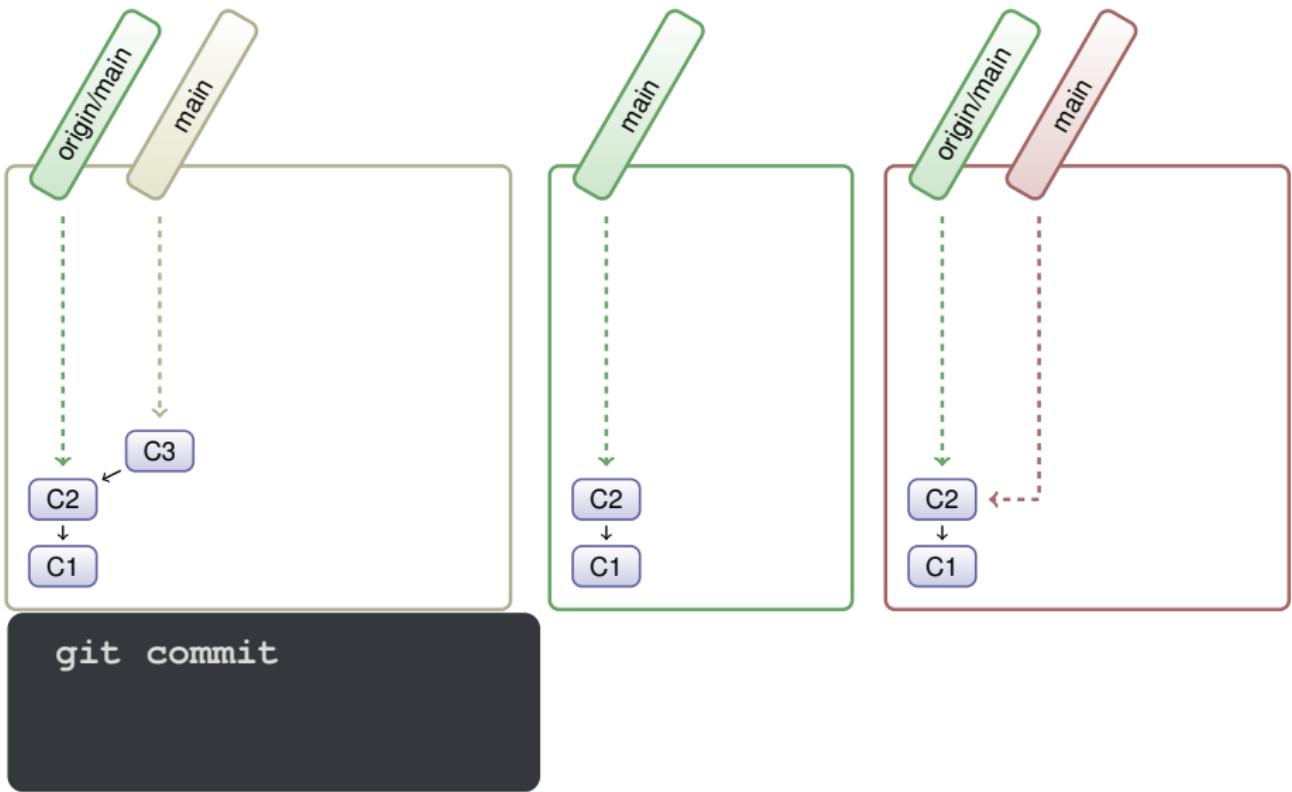
Dépôts distants et gestion de la concurrence.



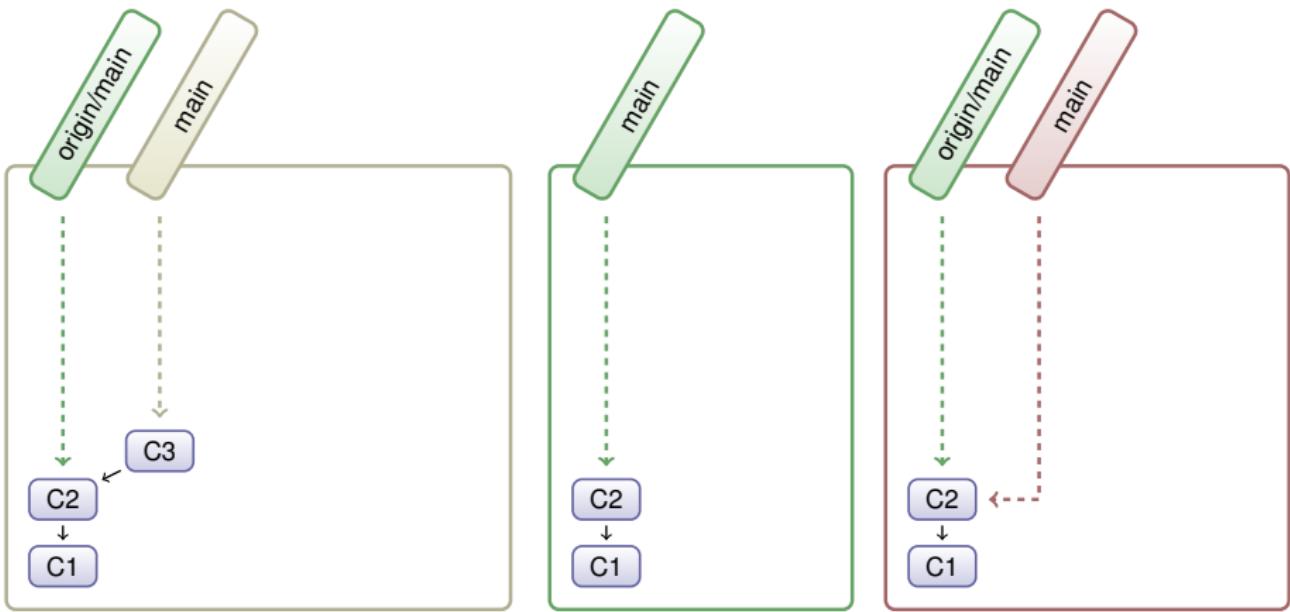
Dépôts distants et gestion de la concurrence.



Dépôts distants et gestion de la concurrence.

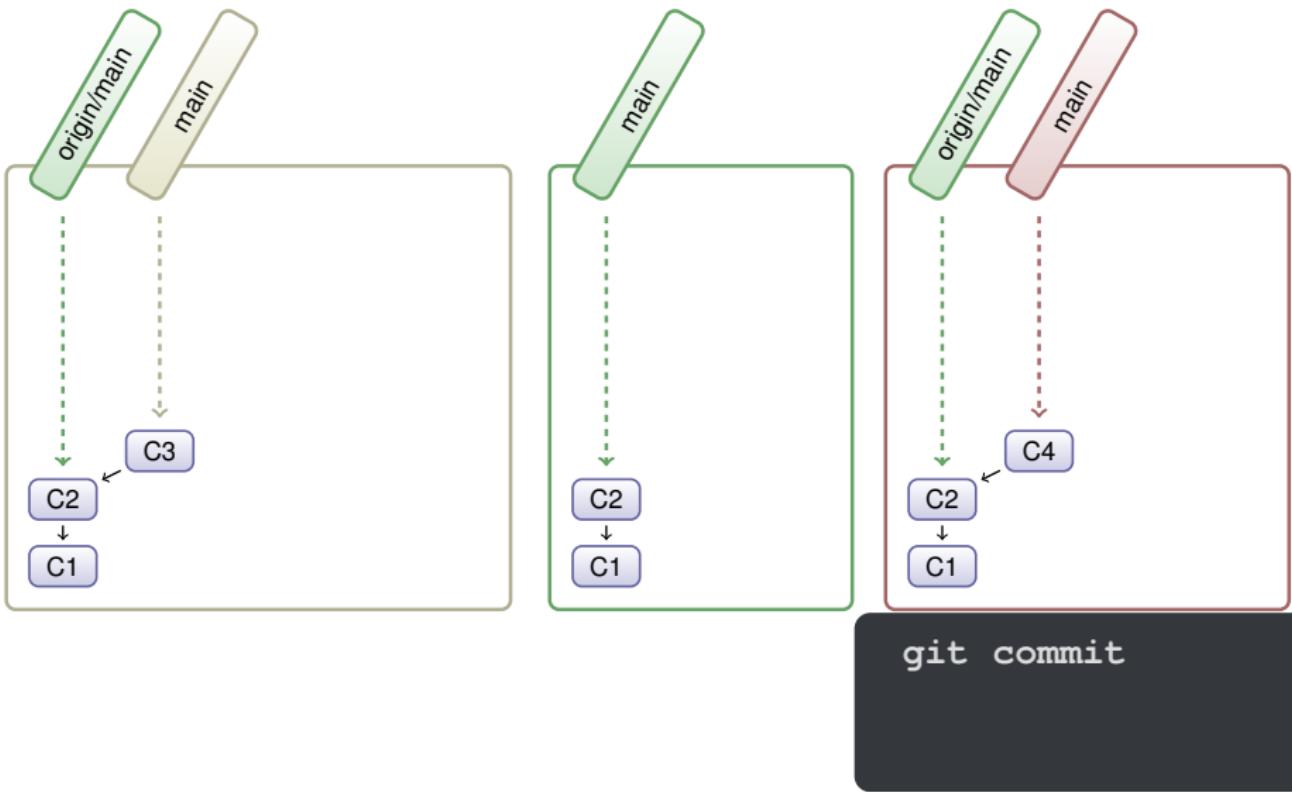


Dépôts distants et gestion de la concurrence.

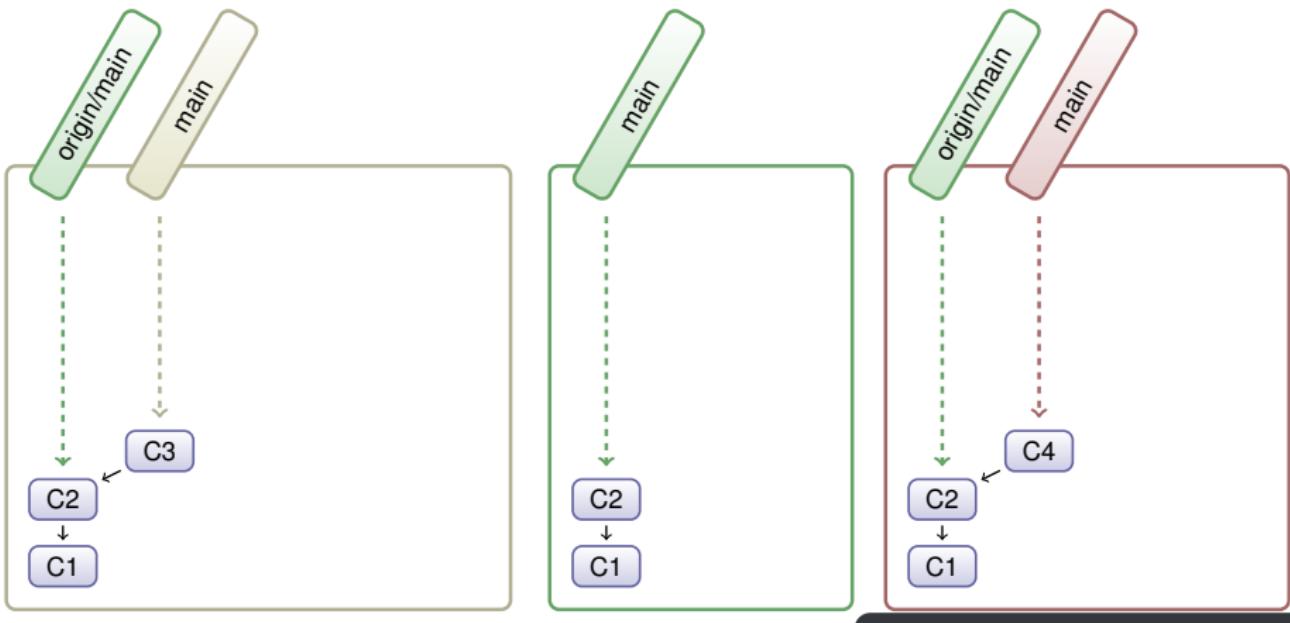


`git commit`

Dépôts distants et gestion de la concurrence.

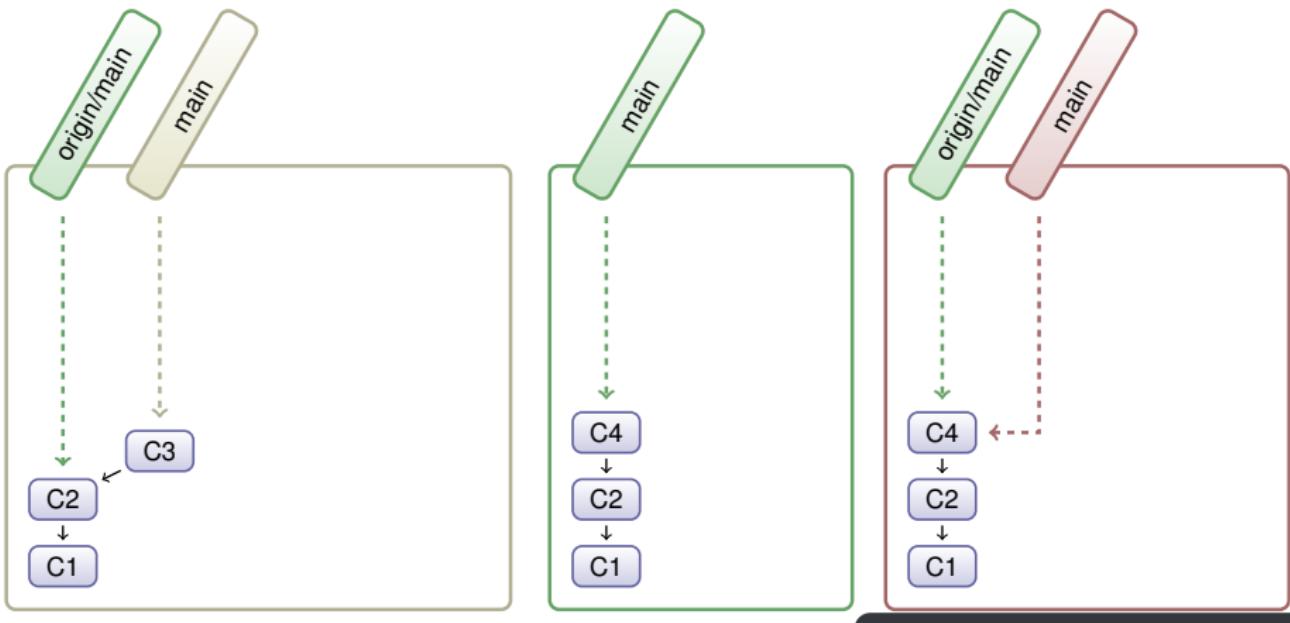


Dépôts distants et gestion de la concurrence.



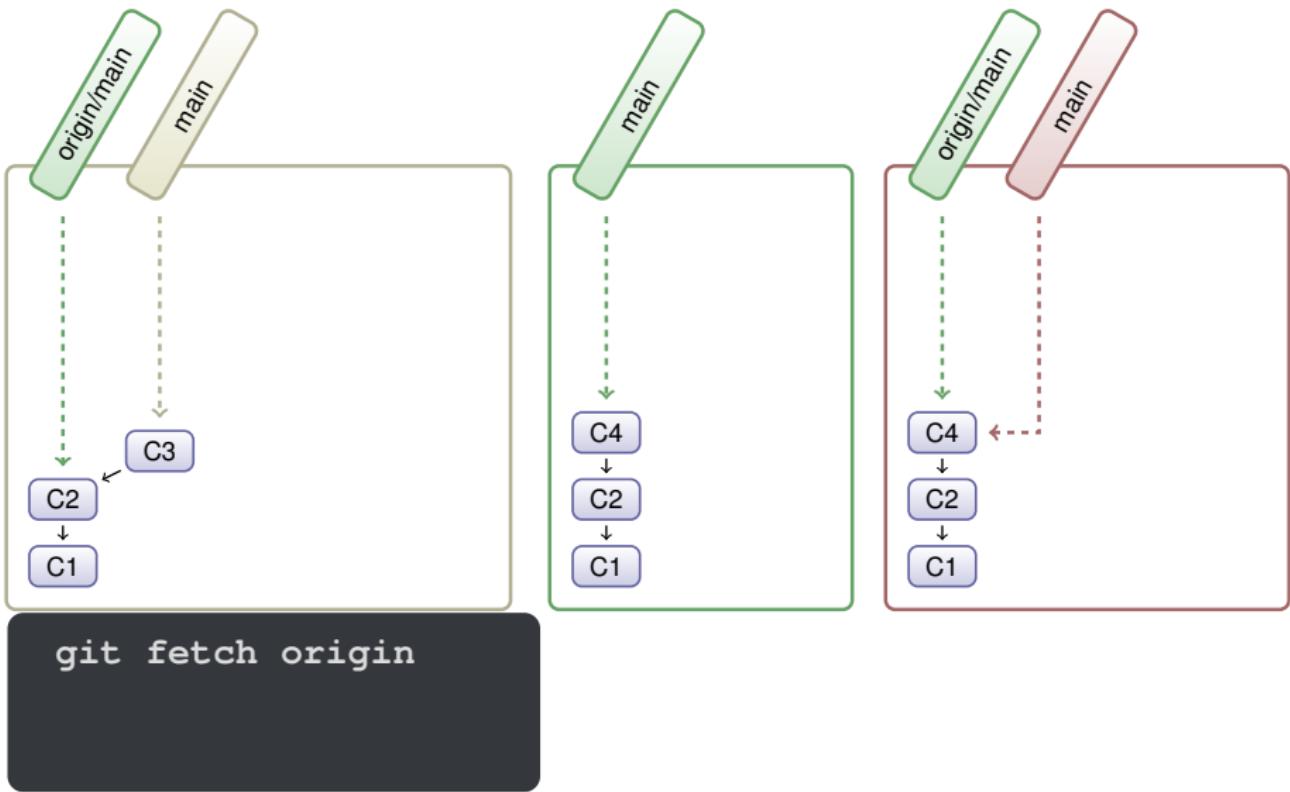
```
git push origin
```

Dépôts distants et gestion de la concurrence.

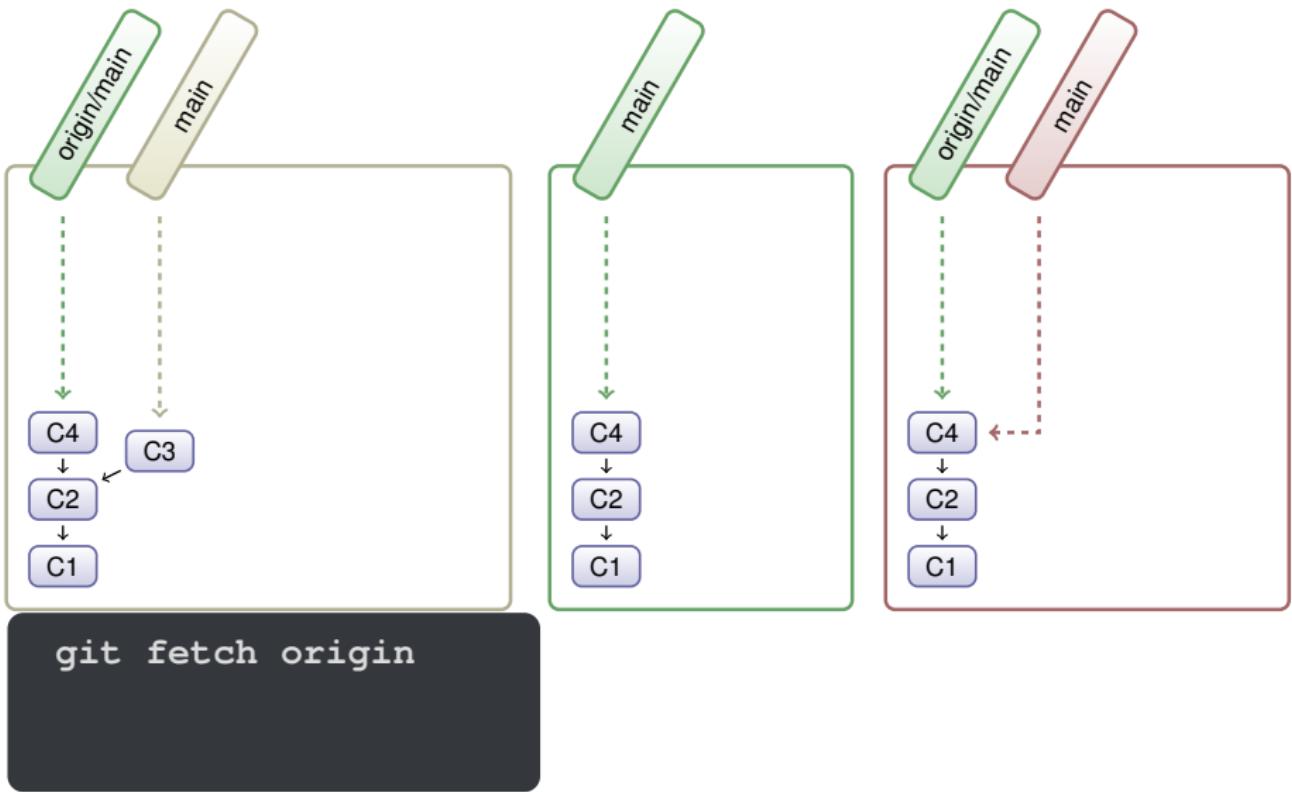


`git push origin`

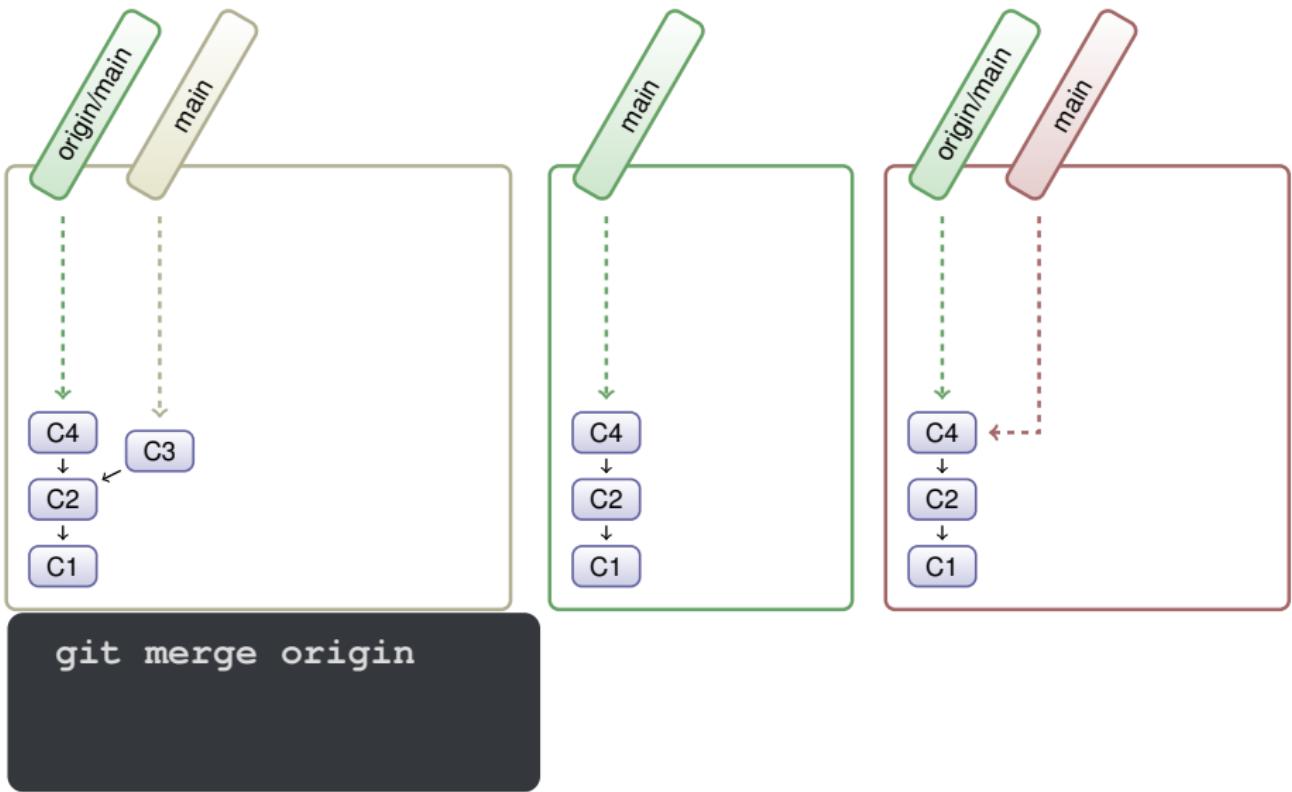
Dépôts distants et gestion de la concurrence.



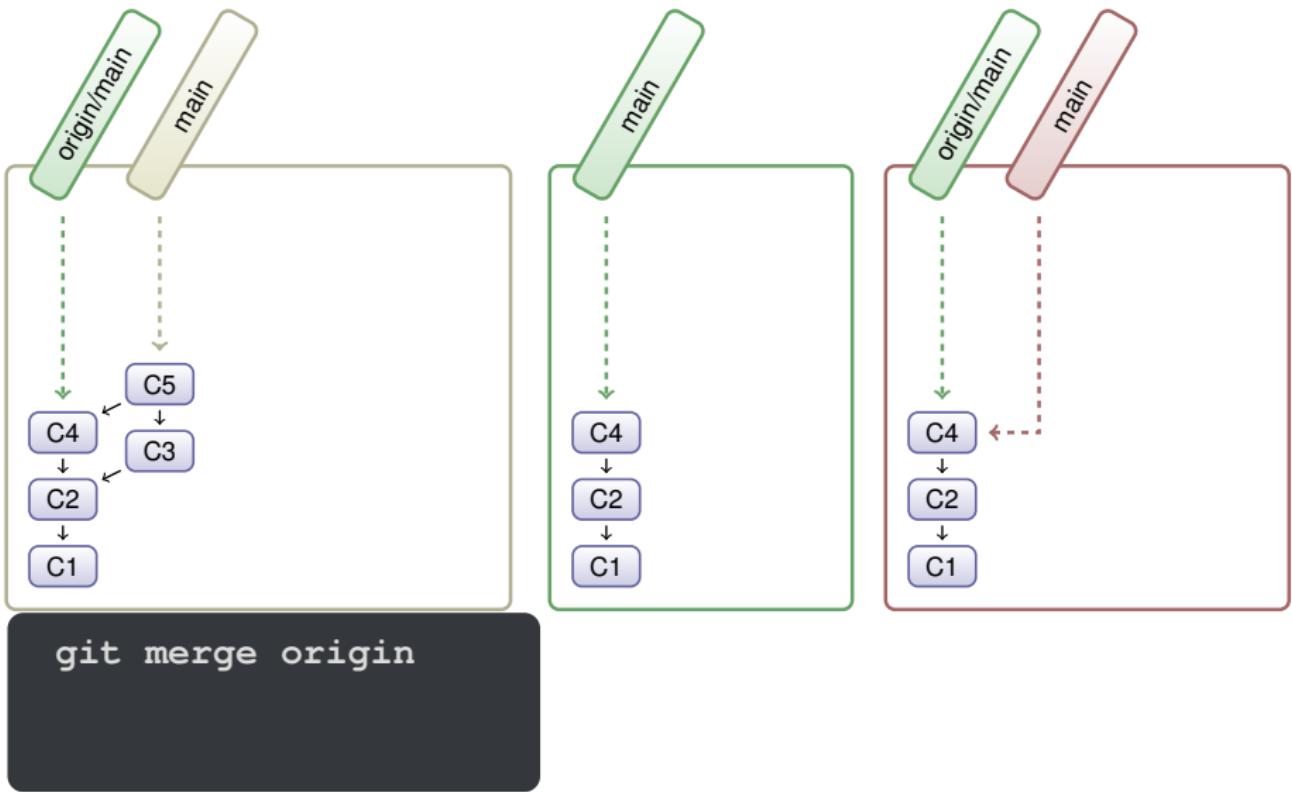
Dépôts distants et gestion de la concurrence.



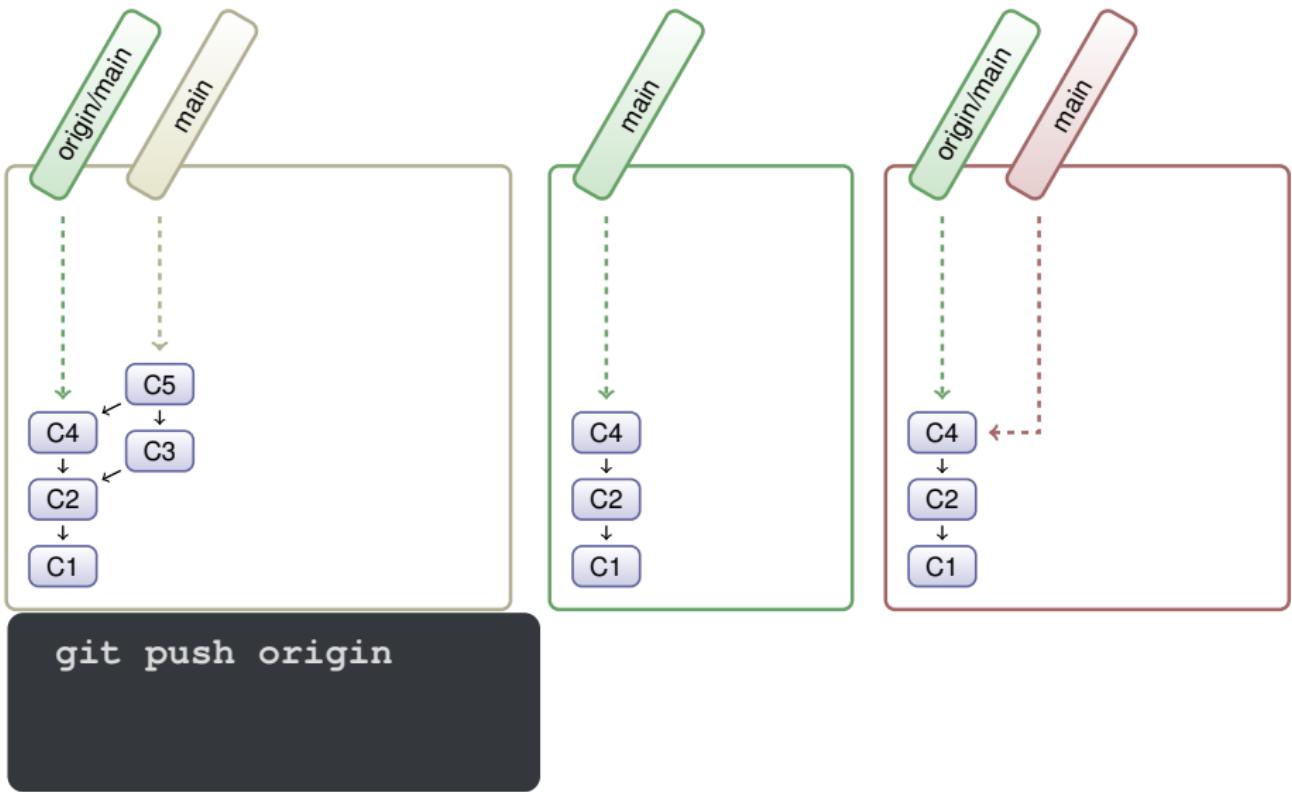
Dépôts distants et gestion de la concurrence.



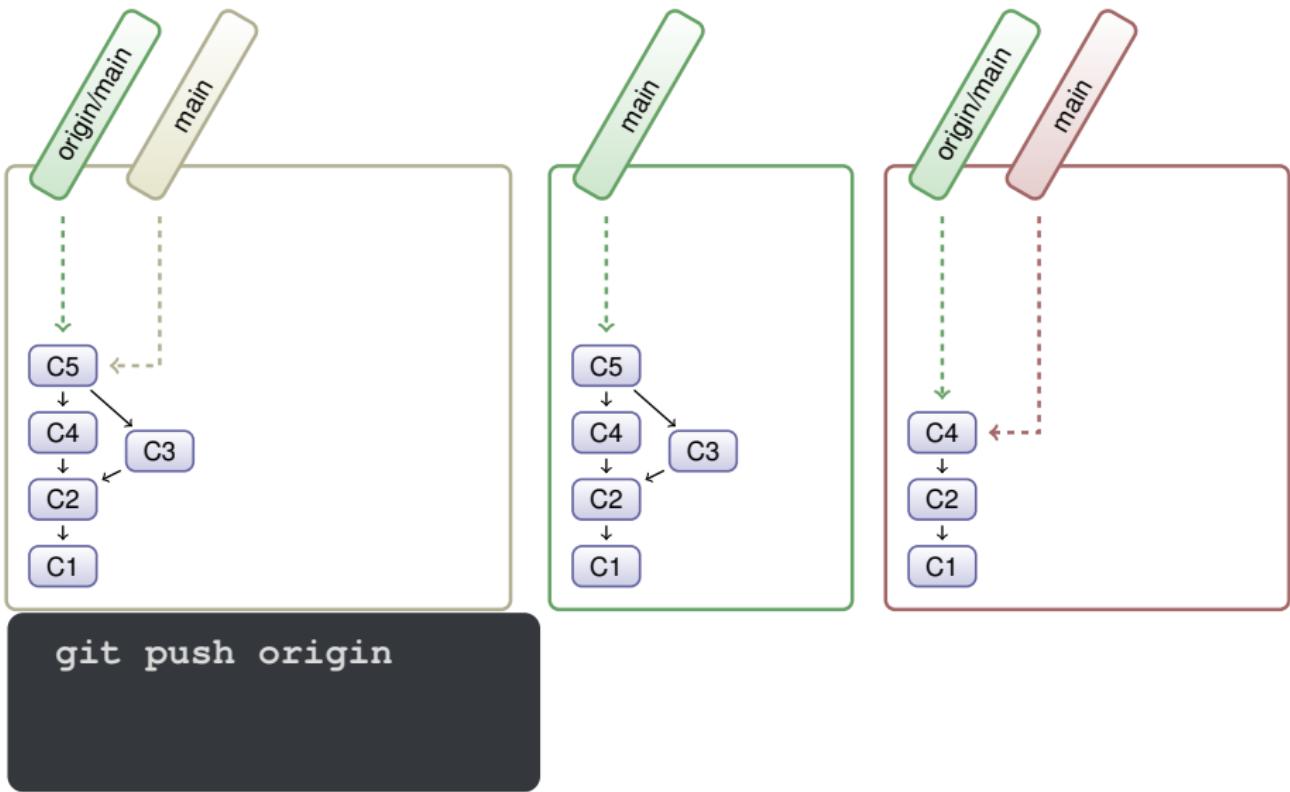
Dépôts distants et gestion de la concurrence.



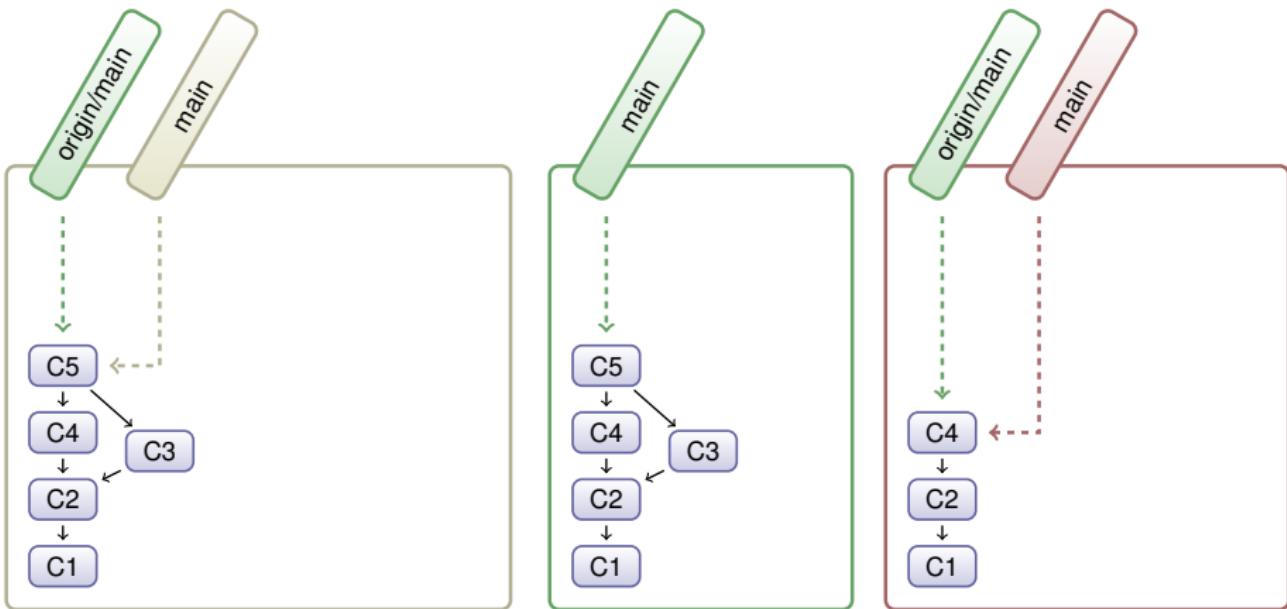
Dépôts distants et gestion de la concurrence.



Dépôts distants et gestion de la concurrence.

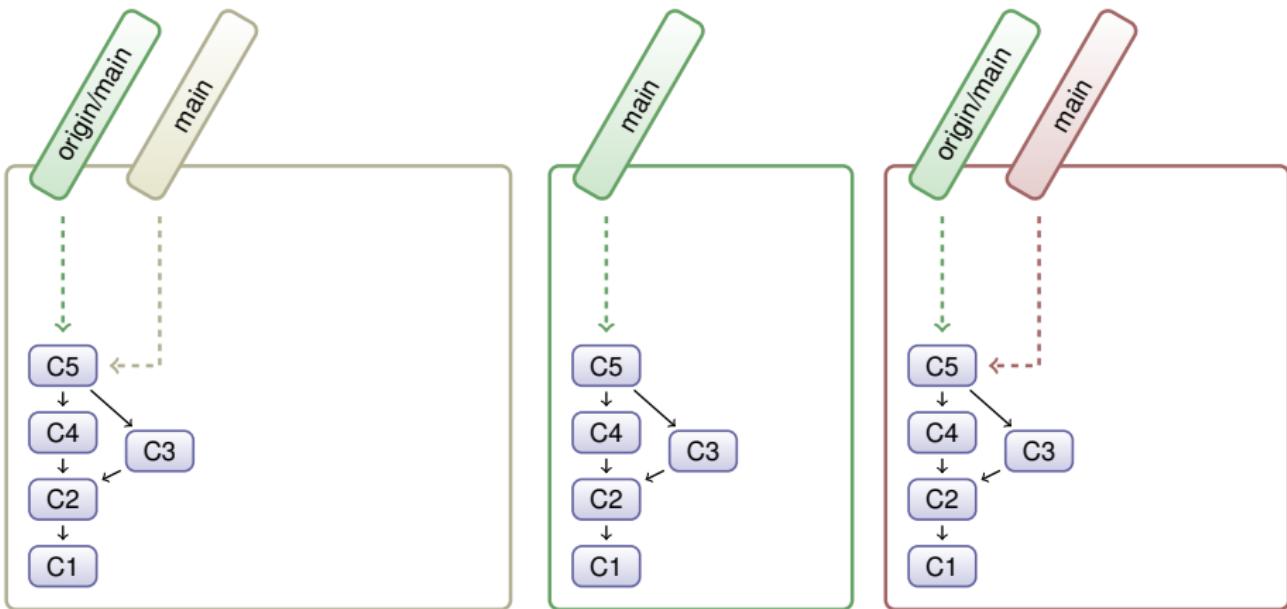


Dépôts distants et gestion de la concurrence.



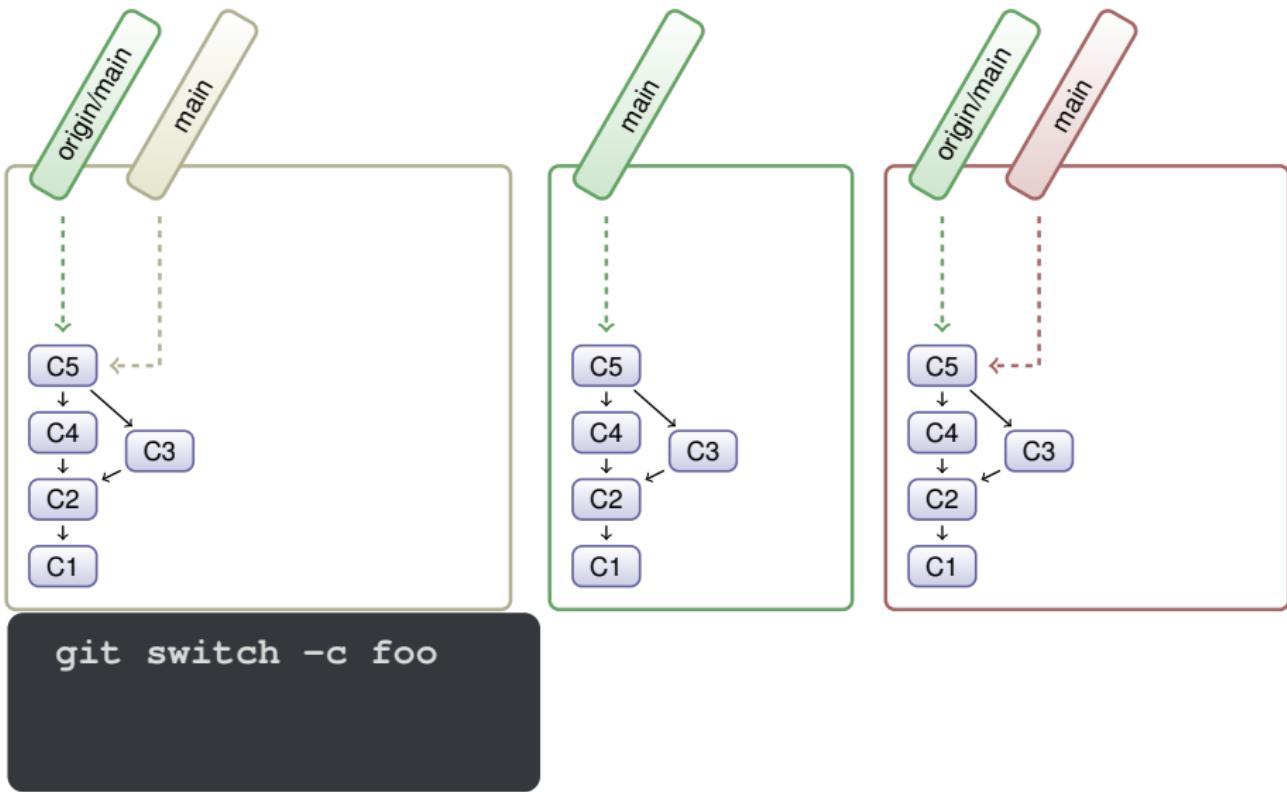
`git pull origin`

Dépôts distants et gestion de la concurrence.

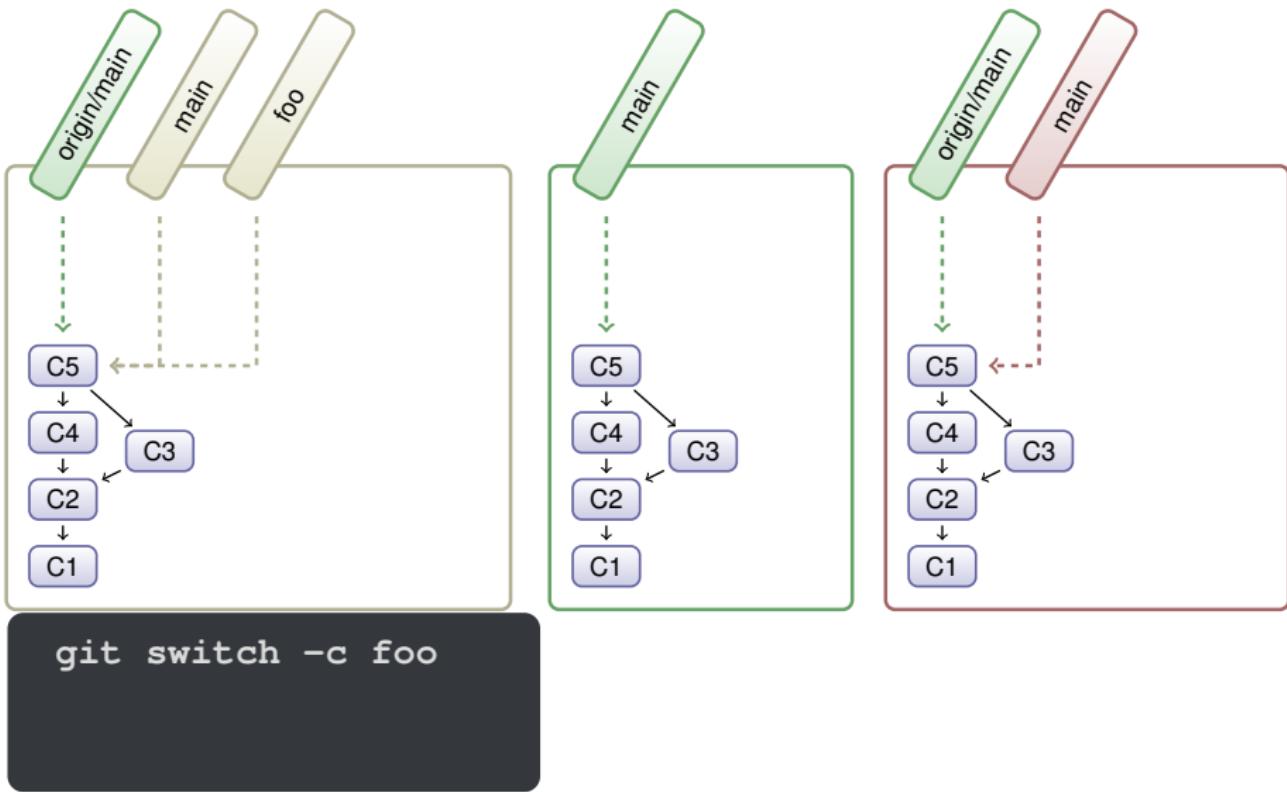


`git pull origin`

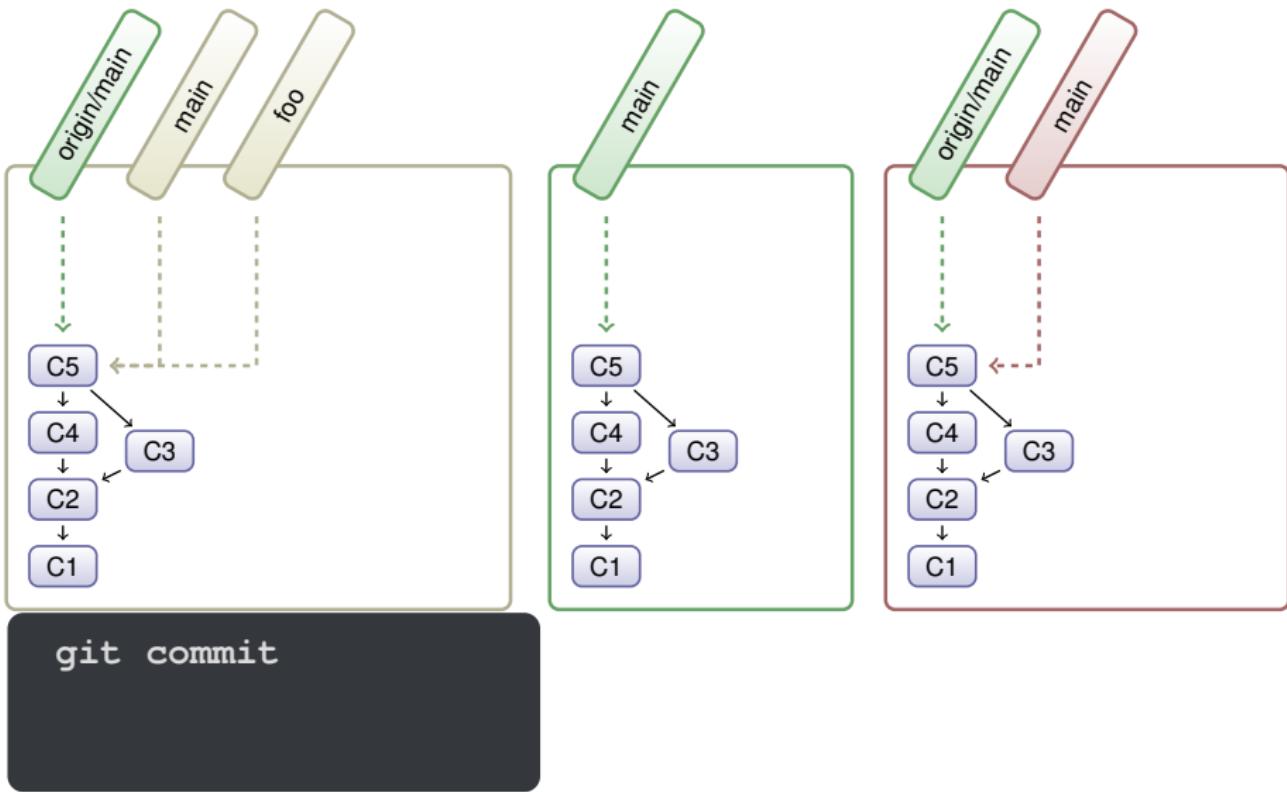
Dépôts distants et gestion de la concurrence.



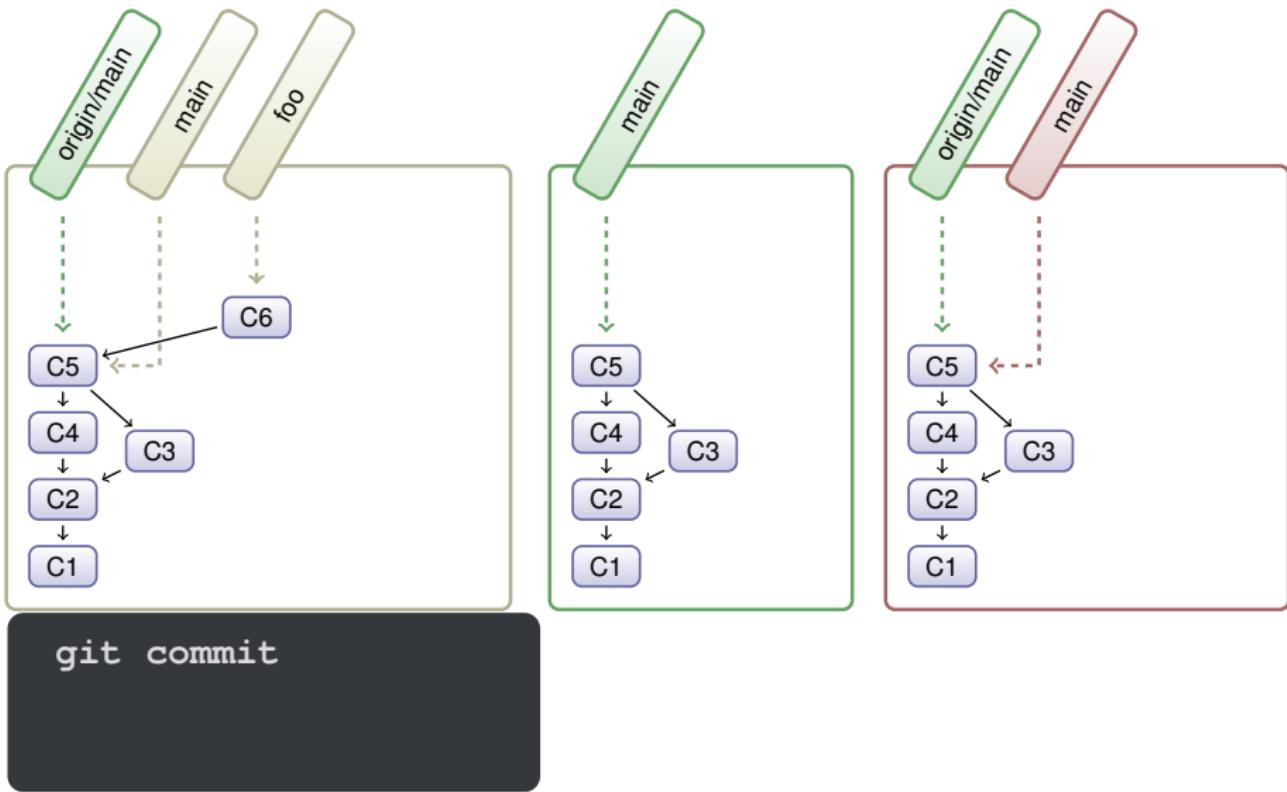
Dépôts distants et gestion de la concurrence.



Dépôts distants et gestion de la concurrence.

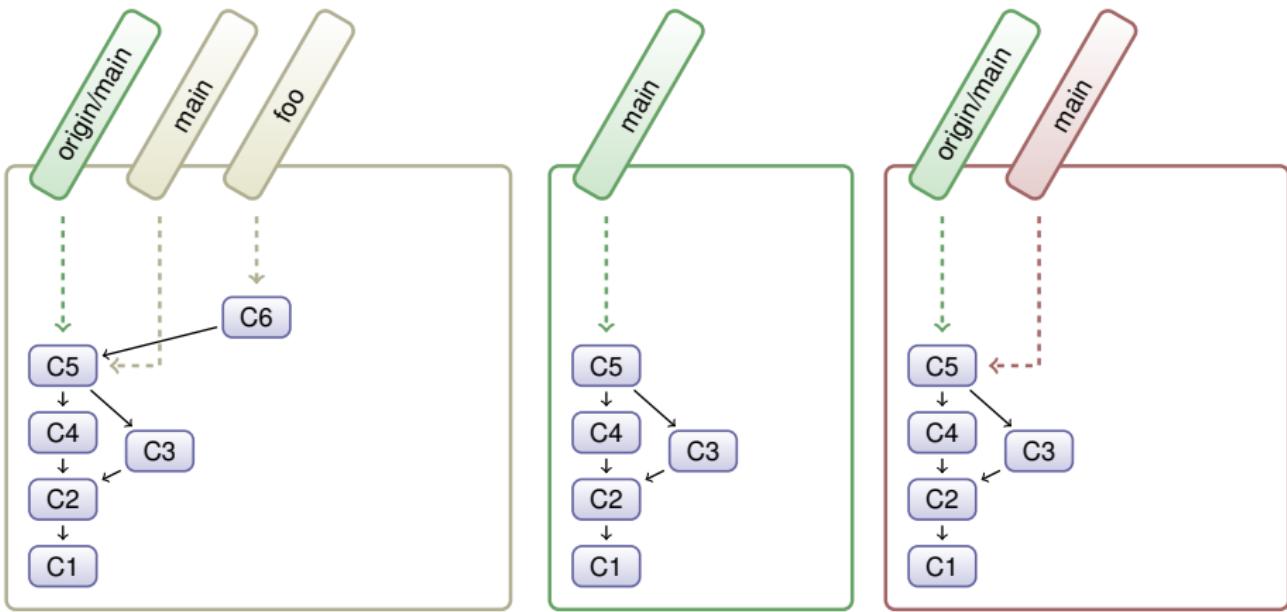


Dépôts distants et gestion de la concurrence.



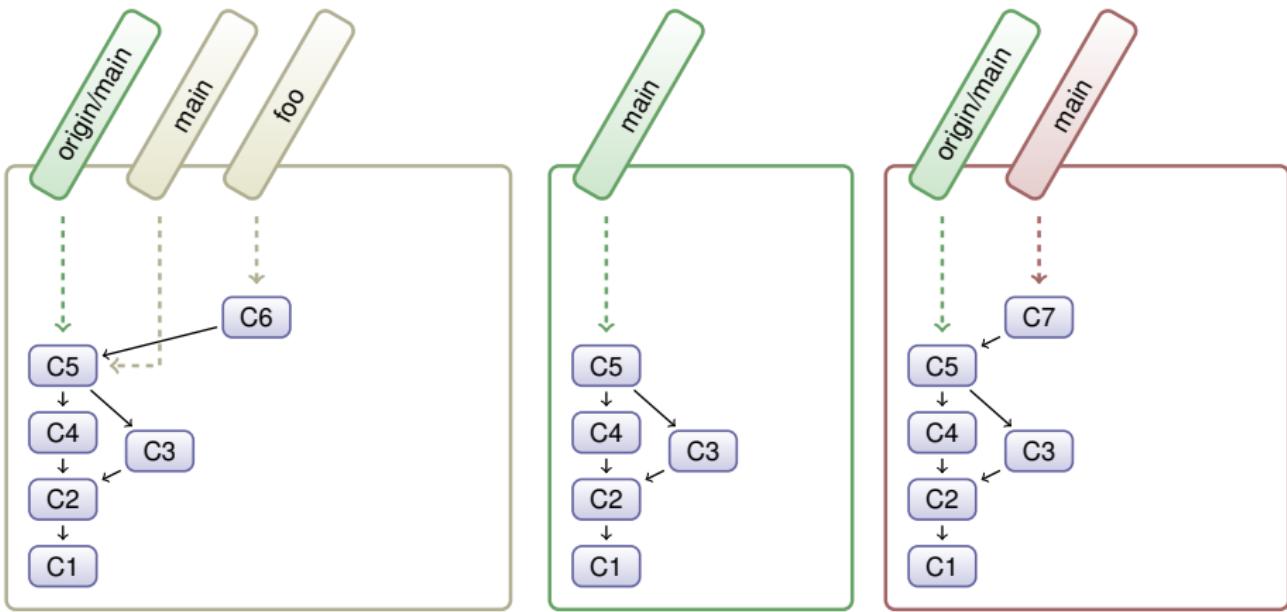
git commit

Dépôts distants et gestion de la concurrence.



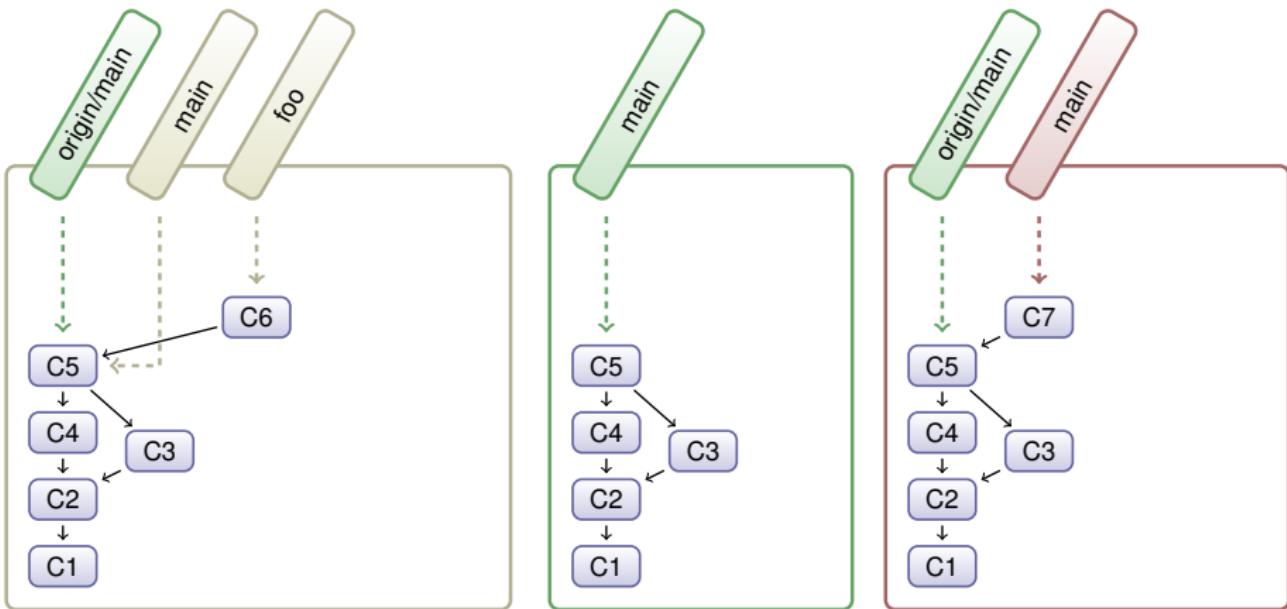
git commit

Dépôts distants et gestion de la concurrence.



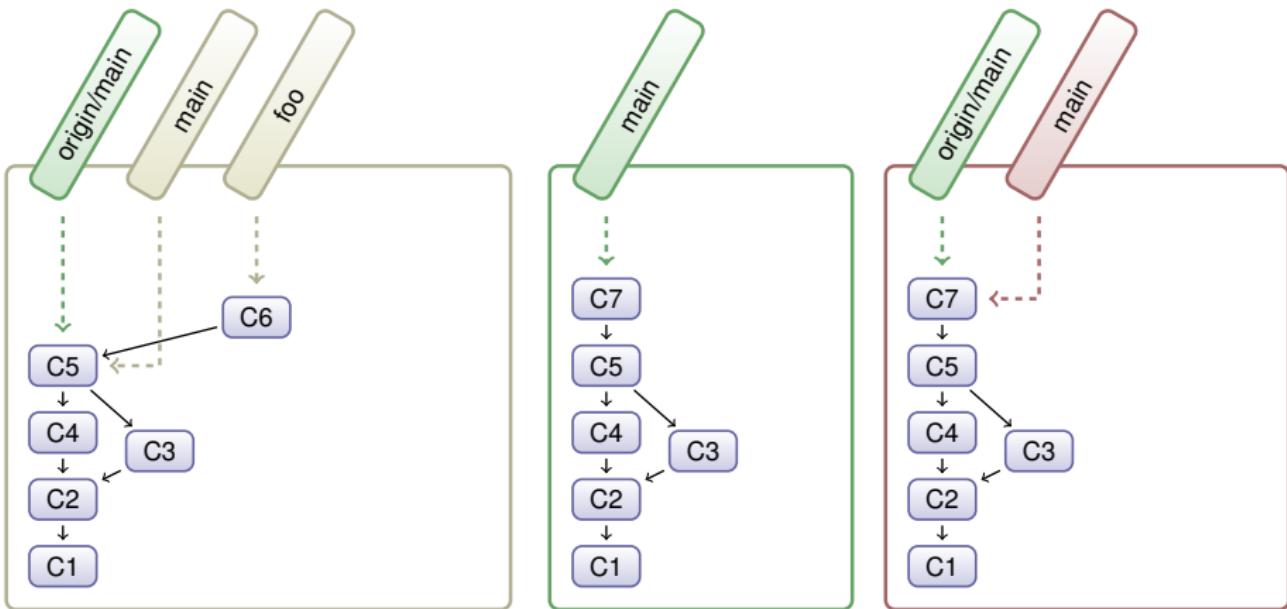
git commit

Dépôts distants et gestion de la concurrence.



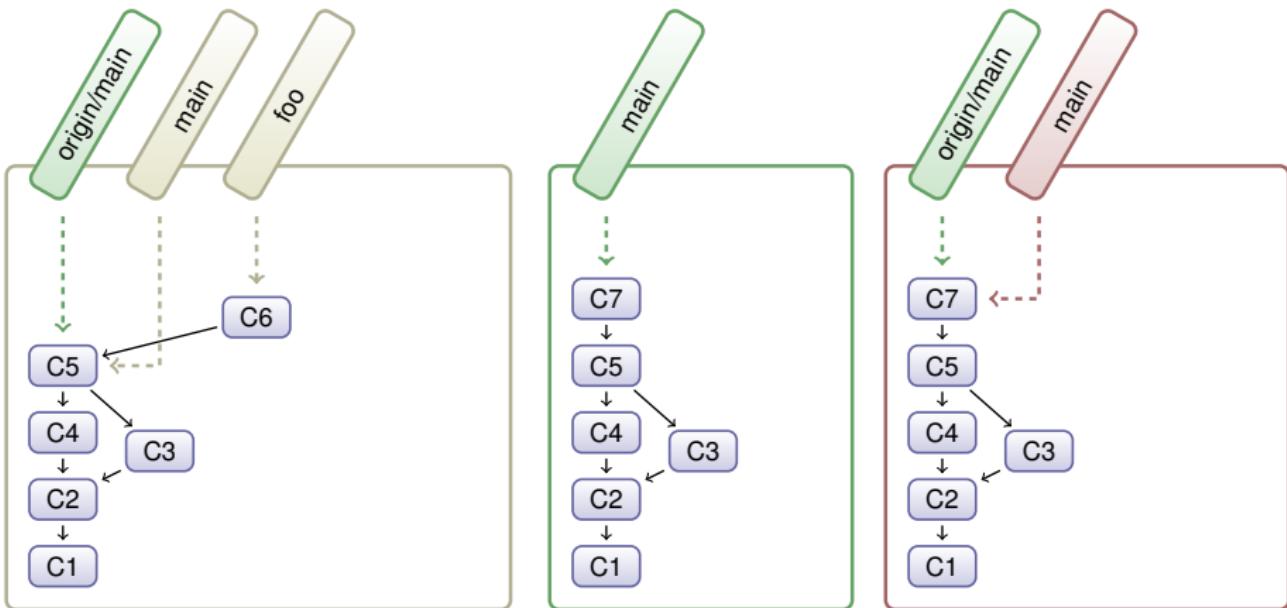
`git push origin`

Dépôts distants et gestion de la concurrence.



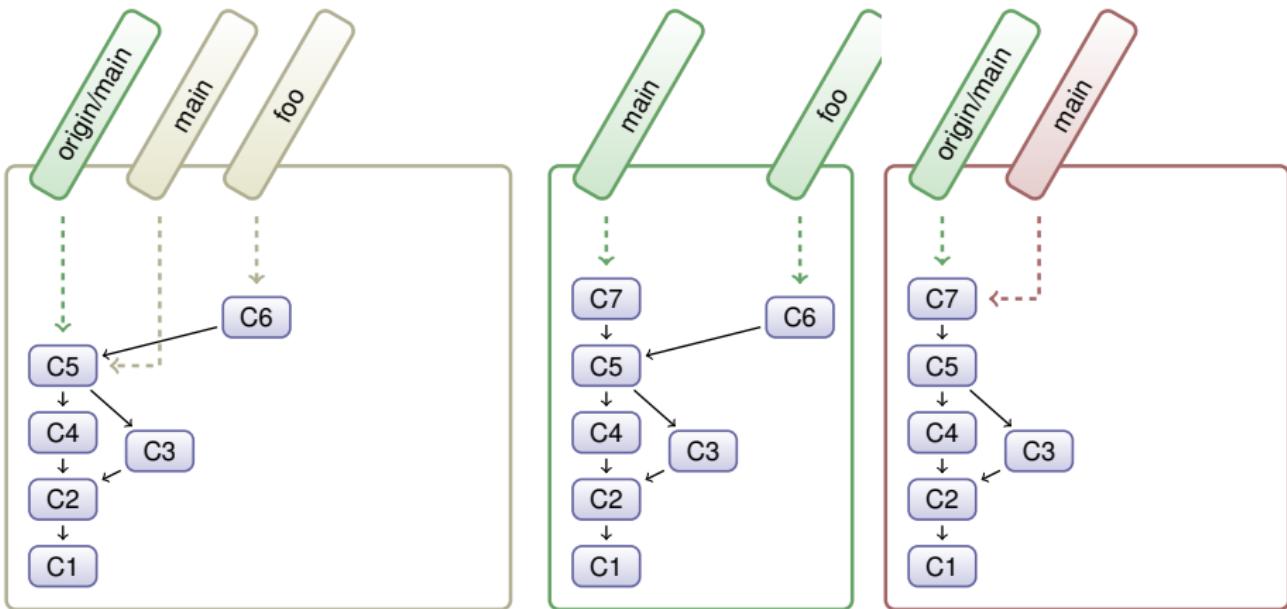
```
git push origin
```

Dépôts distants et gestion de la concurrence.



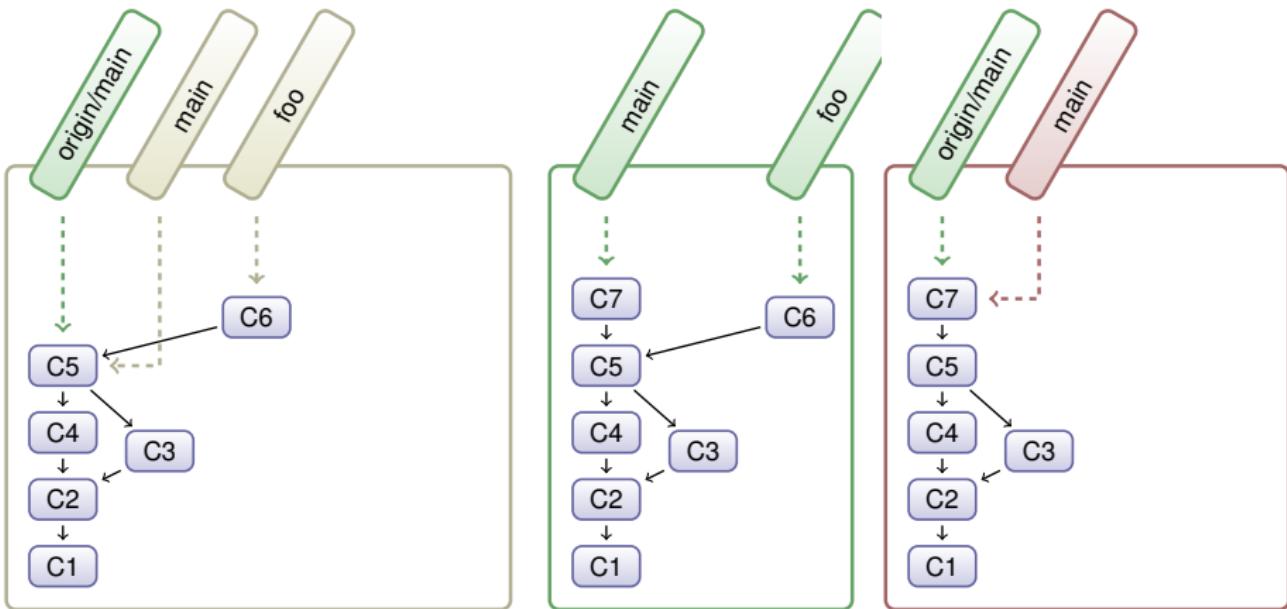
```
git push origin foo
```

Dépôts distants et gestion de la concurrence.



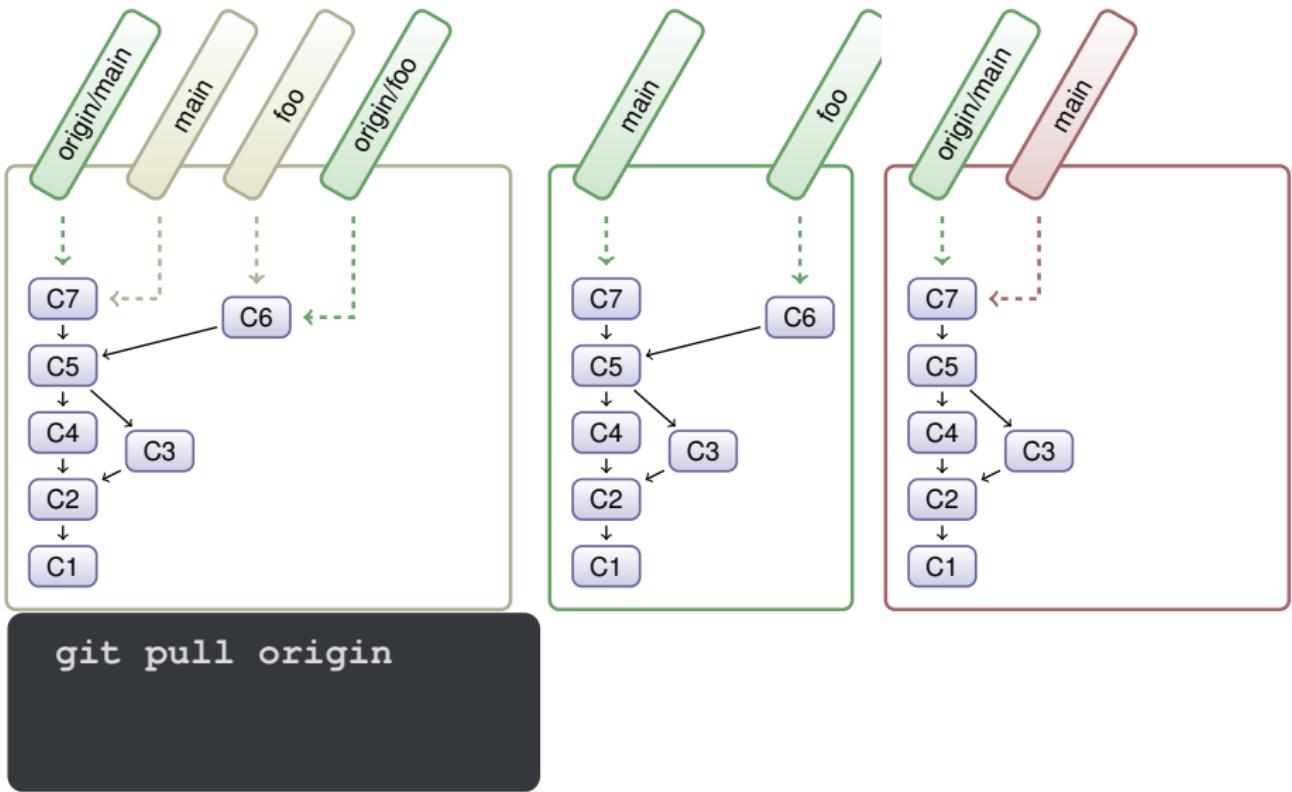
```
git push origin foo
```

Dépôts distants et gestion de la concurrence.

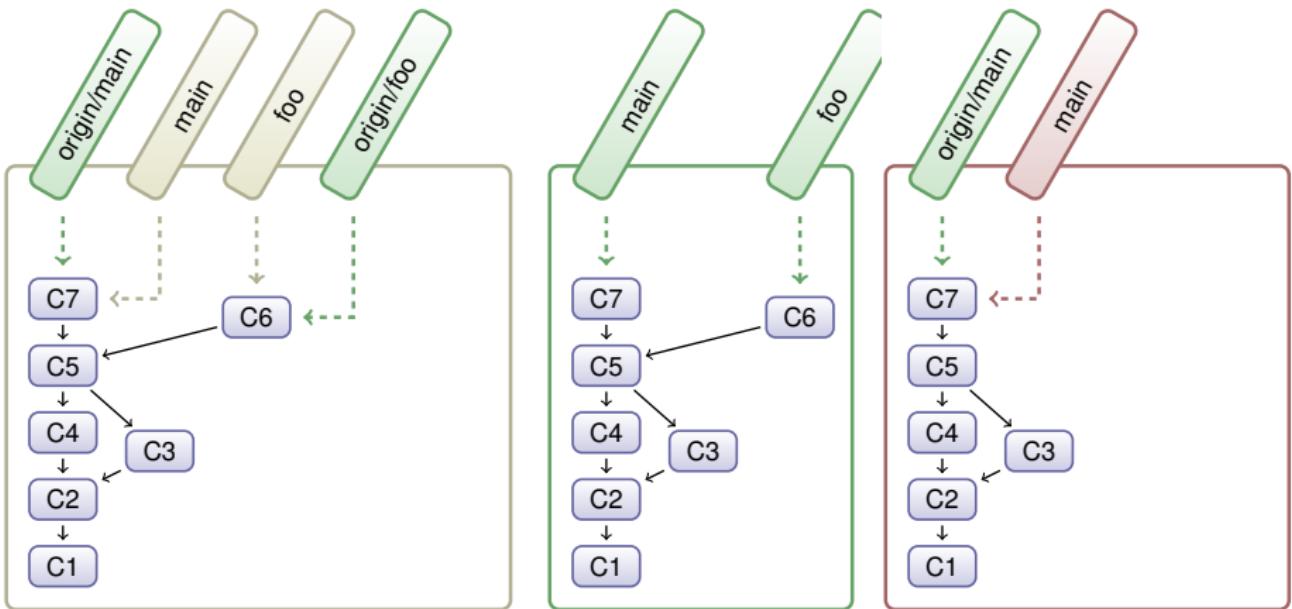


```
git pull origin
```

Dépôts distants et gestion de la concurrence.

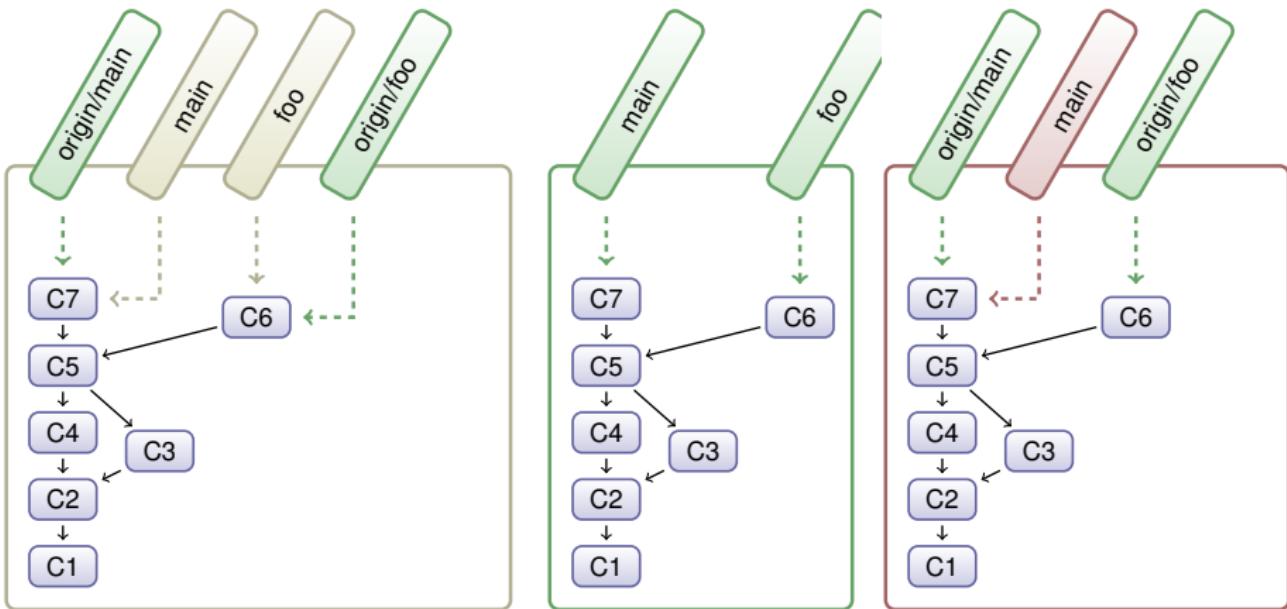


Dépôts distants et gestion de la concurrence.



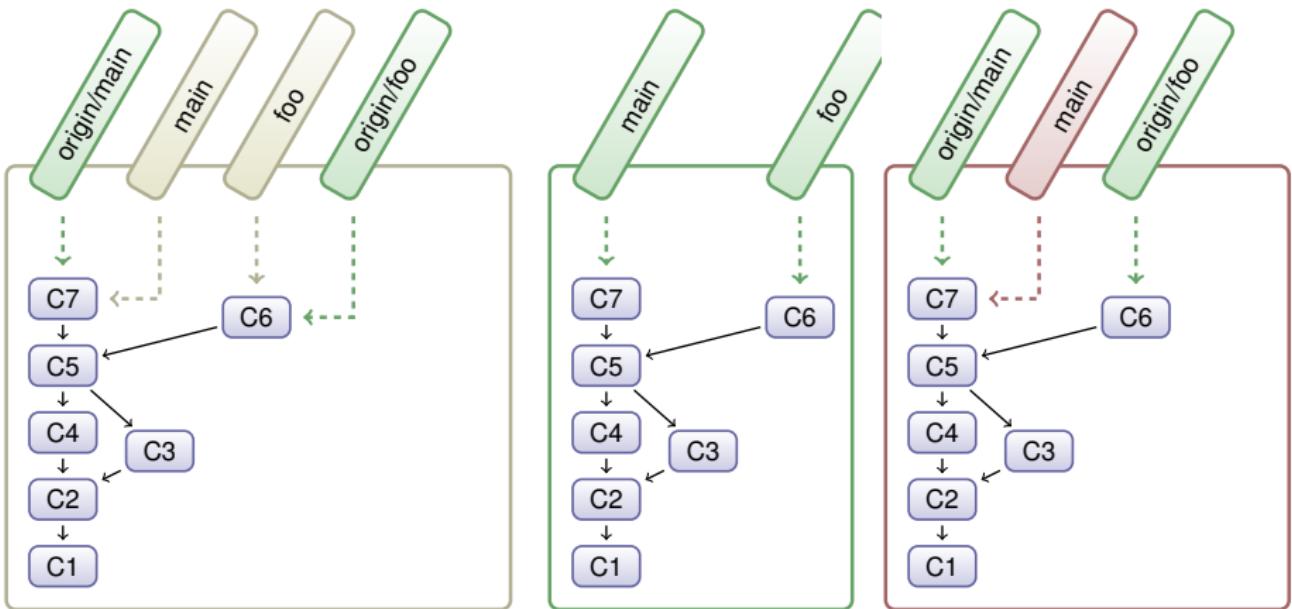
git pull origin

Dépôts distants et gestion de la concurrence.



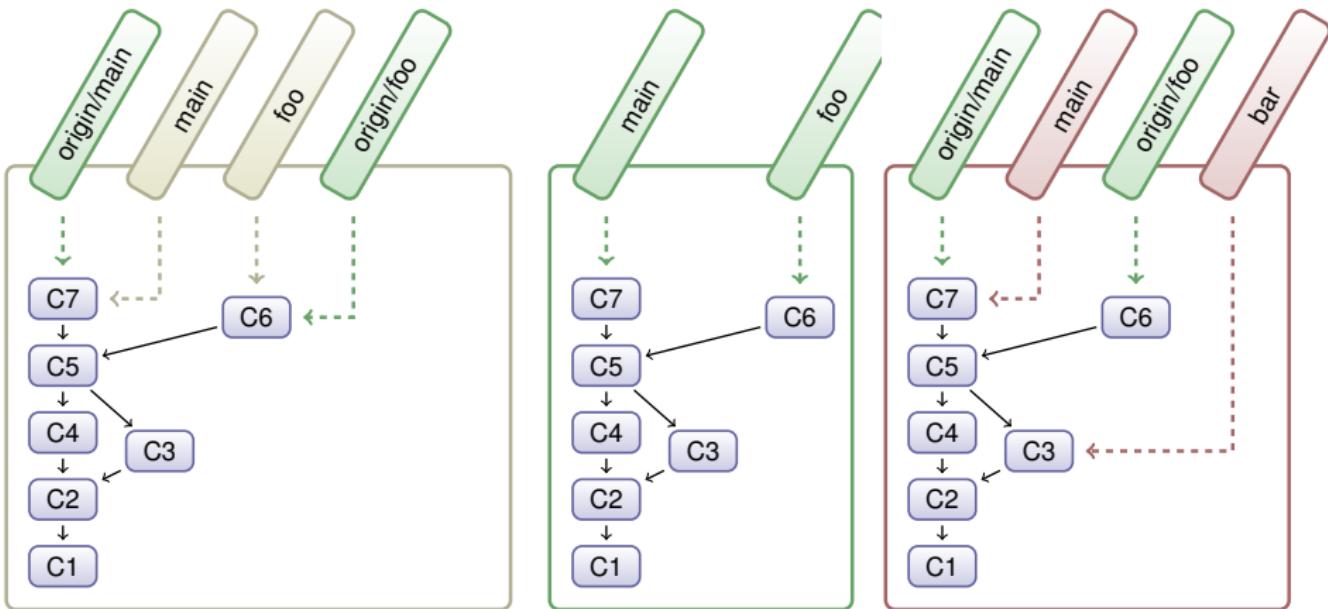
`git pull origin`

Dépôts distants et gestion de la concurrence.



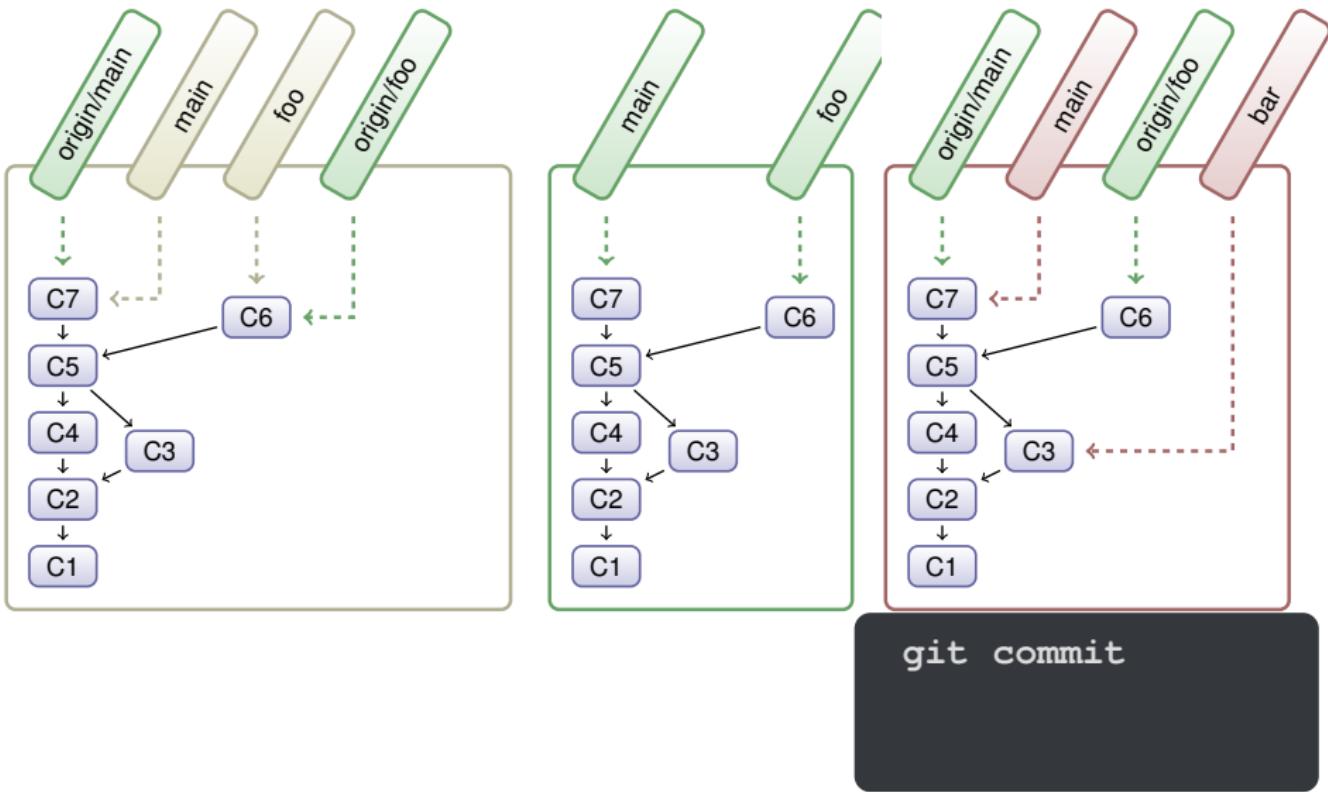
```
git branch bar  
C3  
git switch bar
```

Dépôts distants et gestion de la concurrence.

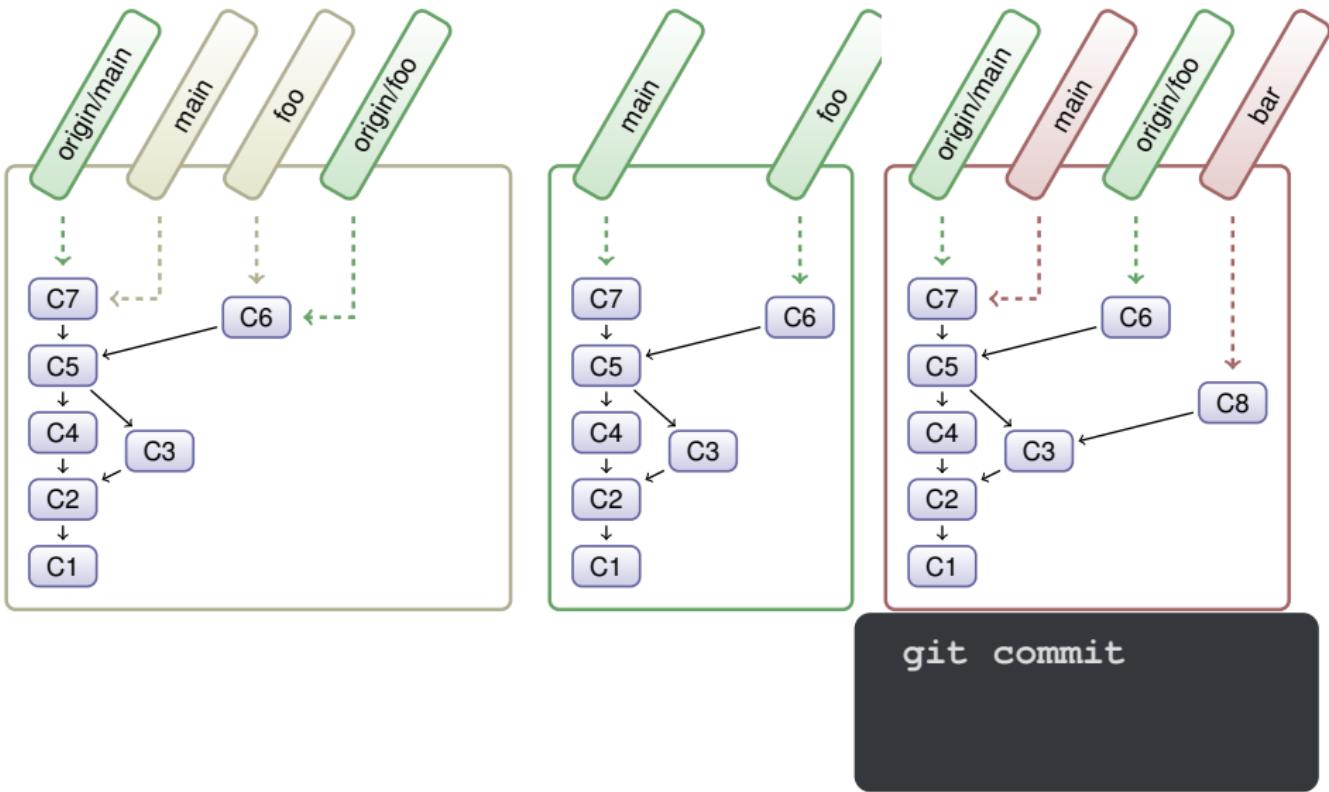


```
git branch bar  
C3  
git switch bar
```

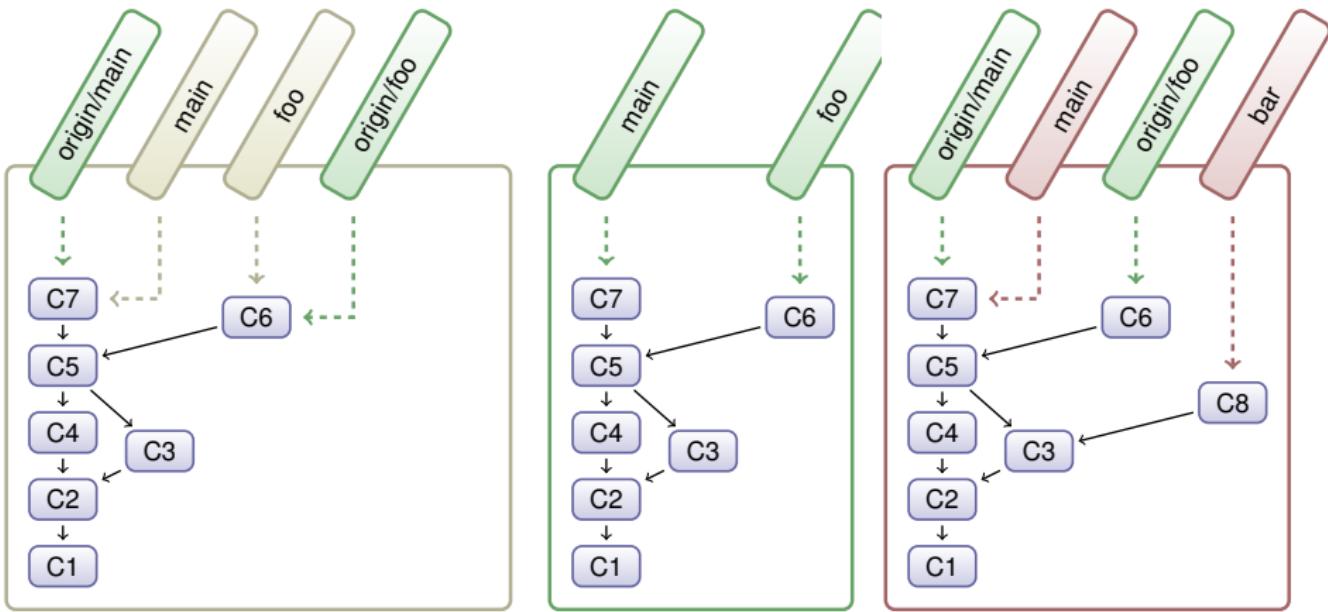
Dépôts distants et gestion de la concurrence.



Dépôts distants et gestion de la concurrence.

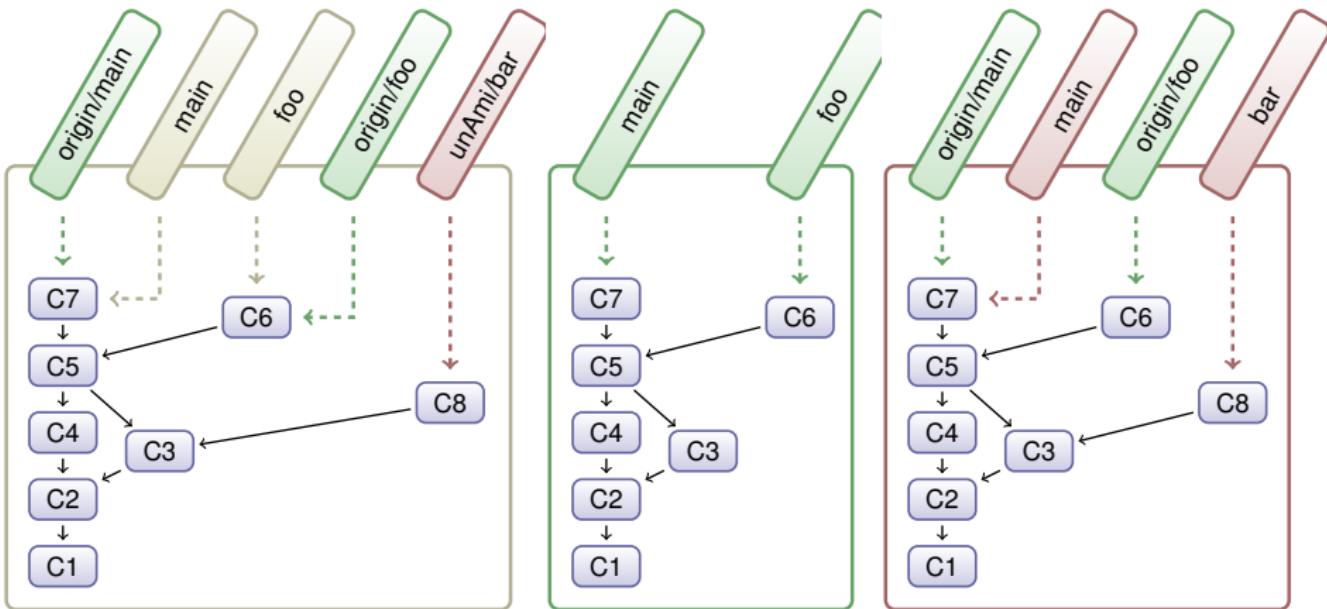


Dépôts distants et gestion de la concurrence.



```
git remote add unAmi  
git://...  
git fetch unAmi/bar
```

Dépôts distants et gestion de la concurrence.



```
git remote add unAmi  
git://...  
git fetch unAmi/bar
```

Outline

Git c'est quoi ?

Architecture interne de git

Gestion des versions dans le dépôt local.

Synchronisation avec les dépôts distants.

Principe des DVCS : Gestionnaire de versions décentralisé.

Les dépôts distants.

Modèles de travail coopératif

Les outils graphiques

Conclusion

Modes de développement

Git permet aux développeurs de gérer leurs sources de 4 manières :

- ▶ un dépôt centralisé à la CVS, SVN tout en conservant les avantages de la conservation du dépôt local ;
- ▶ un dépôt pour chaque développeur, chacun se synchronise chez les autres, méthode traditionnelle de Arch ;
- ▶ un dépôt pour chaque développeur et manager qui se synchronise, fait les merges nécessaires et envoie le tout sur un dépôt public.
- ▶ une gestion par mails de dépôts, méthode de développement du noyau Linux et de son équipe de maintenance.

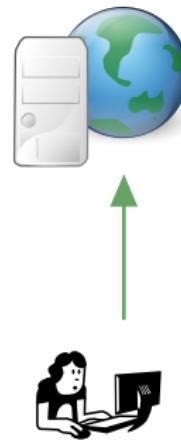
Modèle avec dépôt centralisé

"Centralized Workflow."



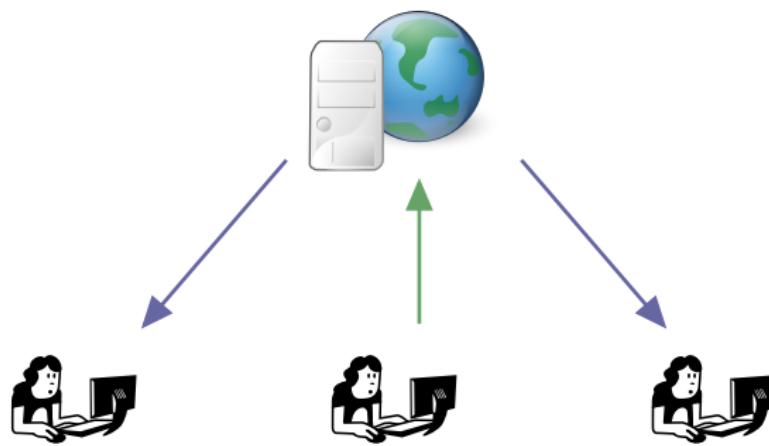
Modèle avec dépôt centralisé

"Centralized Workflow."



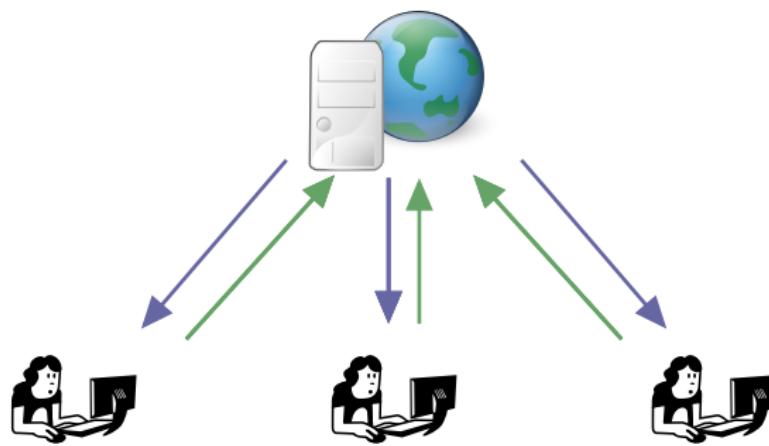
Modèle avec dépôt centralisé

"Centralized Workflow."



Modèle avec dépôt centralisé

"Centralized Workflow."



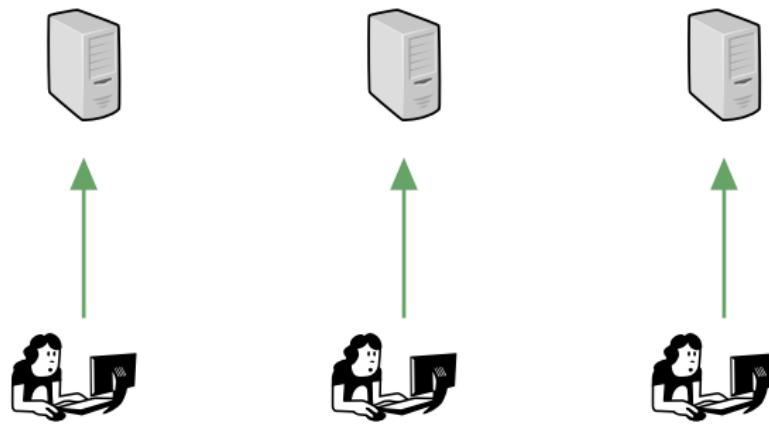
Modèle décentralisé avec dépôts publics.

"Cooperative and Decentralized Workflow."



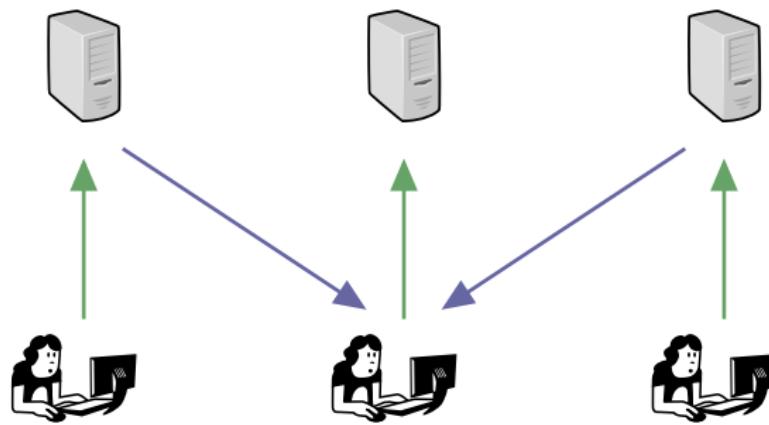
Modèle décentralisé avec dépôts publics.

"Cooperative and Decentralized Workflow."



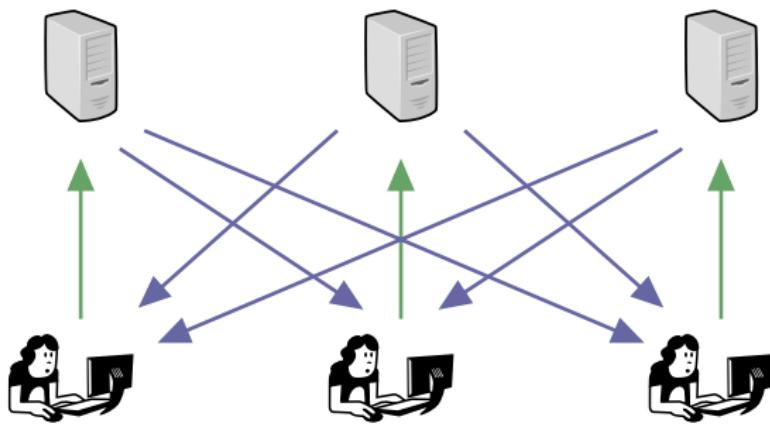
Modèle décentralisé avec dépôts publics.

"Cooperative and Decentralized Workflow."



Modèle décentralisé avec dépôts publics.

"Cooperative and Decentralized Workflow."



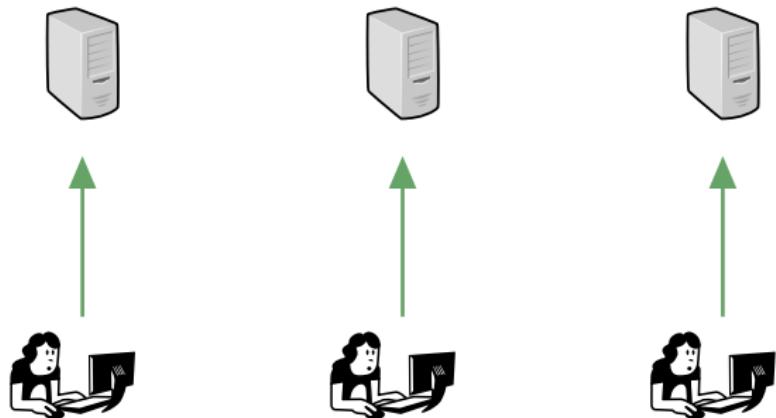
Modèle avec manager de dépôt

"Integration-Manager Workflow."



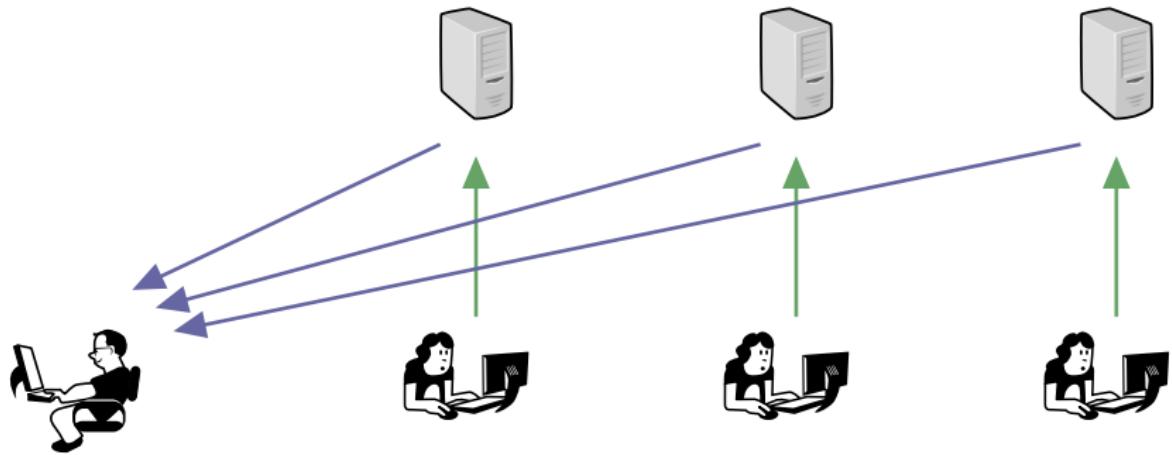
Modèle avec manager de dépôt

"Integration-Manager Workflow."



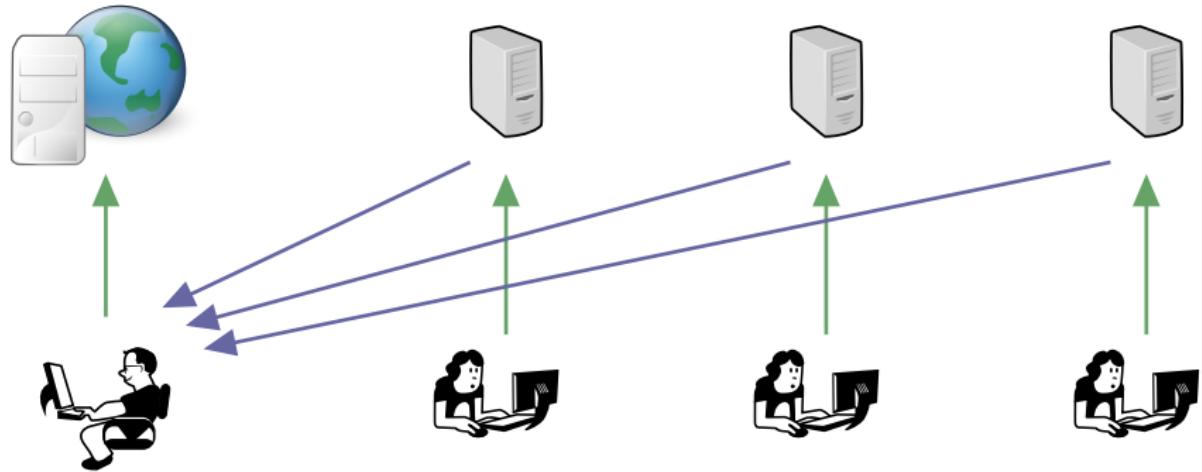
Modèle avec manager de dépôt

"Integration-Manager Workflow."



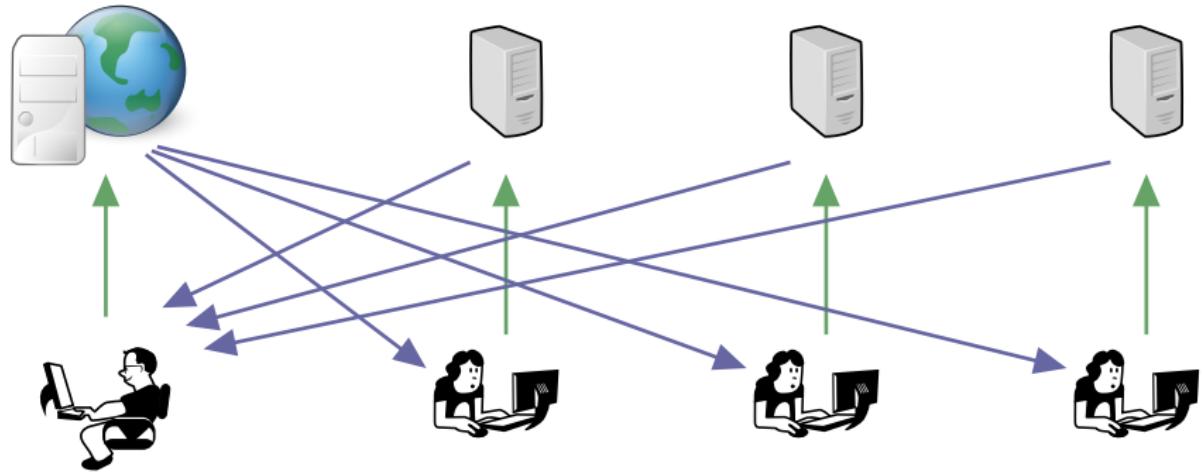
Modèle avec manager de dépôt

"Integration-Manager Workflow."



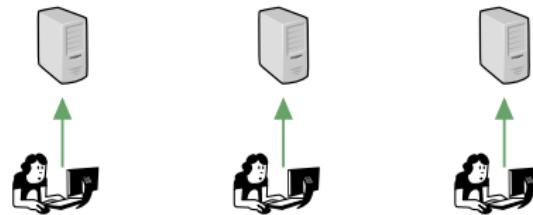
Modèle avec manager de dépôt

"Integration-Manager Workflow."



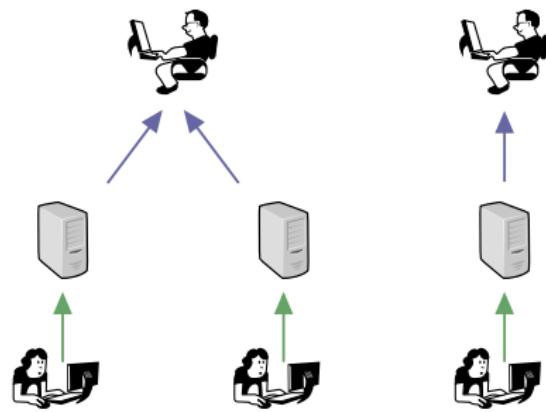
Modèle avec dictateur et lieutenants.

"Dictator and Lieutenants Workflow."



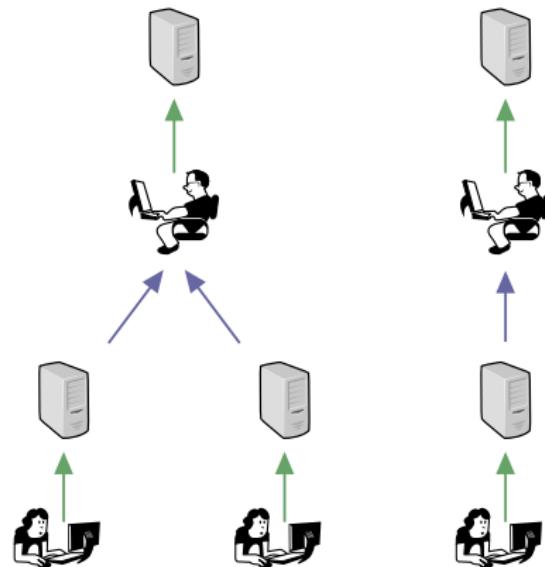
Modèle avec dictateur et lieutenants.

"Dictator and Lieutenants Workflow."



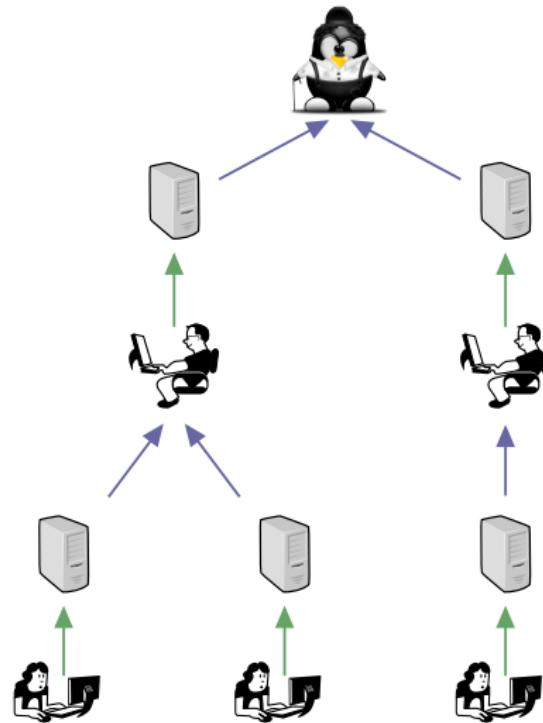
Modèle avec dictateur et lieutenants.

"Dictator and Lieutenants Workflow."



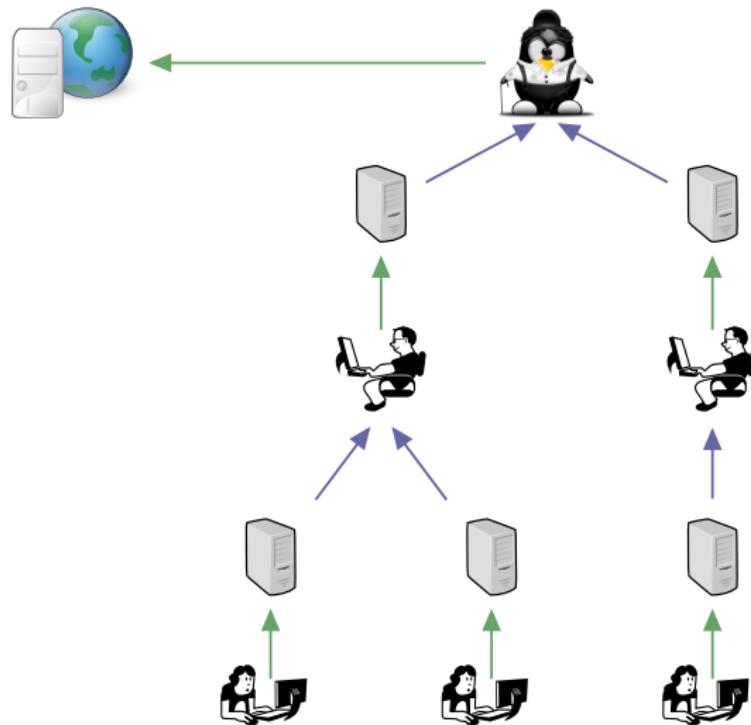
Modèle avec dictateur et lieutenants.

"Dictator and Lieutenants Workflow."



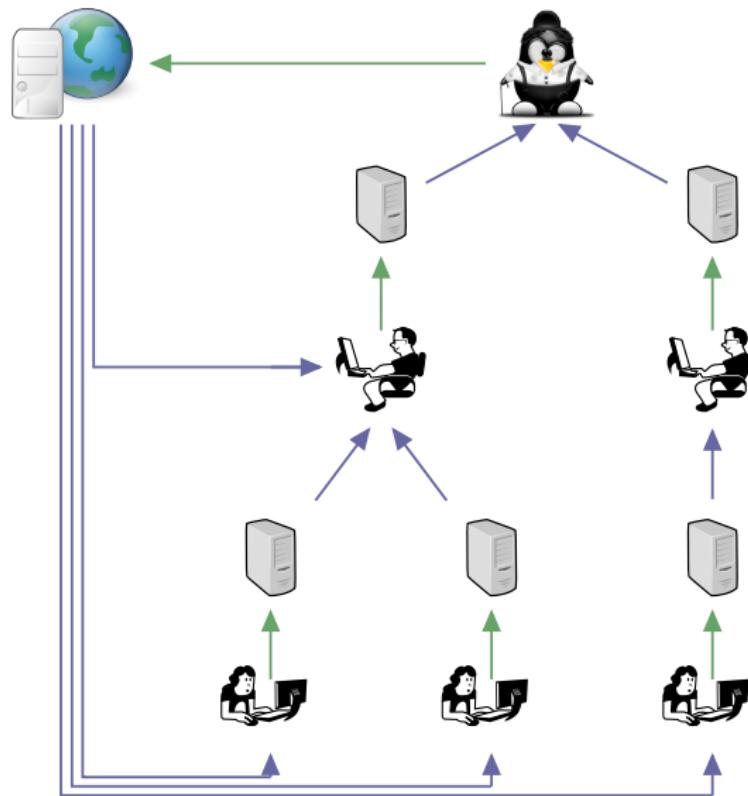
Modèle avec dictateur et lieutenants.

"Dictator and Lieutenants Workflow."



Modèle avec dictateur et lieutenants.

"Dictator and Lieutenants Workflow."



Outline

Git c'est quoi ?

Architecture interne de git

Gestion des versions dans le dépôt local.

Synchronisation avec les dépôts distants.

Les outils graphiques

git-gui et gitk

Exemple de gitg

Conclusion

Outline

Git c'est quoi ?

Architecture interne de git

Gestion des versions dans le dépôt local.

Synchronisation avec les dépôts distants.

Les outils graphiques

git-gui et gitk

Exemple de gitg

Conclusion

Les interfaces graphiques

Il existe de nombreuses interfaces graphiques permettant de gérer vos projets quelle que soit votre plate-forme de développement :

- ▶ **Avec git :**
 - ▶ **gitk** : l'interface de visualisation détaillée et graphique d'un historique git
 - ▶ **git-gui** : outil permettant de construire les commits
- ▶ **Linux** : gitg, Giggle, ...
- ▶ **Windows** : TortoiseGit, GitExtensions, ...
- ▶ **Apple** : GitX, Gitti, ...
- ▶ **Eclipse** : EGit, ...

Outline

Git c'est quoi ?

Architecture interne de git

Gestion des versions dans le dépôt local.

Synchronisation avec les dépôts distants.

Les outils graphiques

git-gui et gitk

Exemple de gitg

Conclusion

gitg : visualisation de l'historique

The screenshot shows the gitg application interface, which is a graphical front-end for Git. It displays a timeline of commits on the left and a detailed view of a specific commit on the right.

Commit Timeline (Left):

- Commits are color-coded by author.
- Branches are shown as colored lines connecting commits.
- Some commits have yellow circles with numbers (e.g., 13404 & t7509) indicating they are part of a merge.
- A large blue circle highlights a commit from Junio C Hamano.

Detailed Commit View (Right):

Author	Date
Junio C Hamano	mer. 08 sept. 2010 18:17:01 CEST
Junio C Hamano	mer. 08 sept. 2010 18:17:00 CEST
Junio C Hamano	mer. 08 sept. 2010 18:17:00 CEST
Junio C Hamano	mer. 08 sept. 2010 17:54:01 CEST
Johannes Stöckl	mar. 07 sept. 2010 21:39:02 CEST
Nicolas Pitre	mar. 07 sept. 2010 02:29:57 CEST
Junio C Hamano	mar. 07 sept. 2010 02:40:18 CEST
Elijah Newren	lun. 06 sept. 2010 23:53:24 CEST
Elijah Newren	lun. 06 sept. 2010 23:40:16 CEST
Junio C Hamano	mar. 07 sept. 2010 01:57:05 CEST
Thiago Farinha	mar. 07 sept. 2010 01:52:55 CEST
Jens Lehmann	lun. 06 sept. 2010 20:41:06 CEST
Junio C Hamano	lun. 06 sept. 2010 09:12:04 CEST
Junio C Hamano	lun. 06 sept. 2010 09:11:59 CEST
Junio C Hamano	lun. 06 sept. 2010 07:32:05 CEST
Jared Hance	dim. 05 sept. 2010 21:36:33 CEST
Jens Lehmann	dim. 05 sept. 2010 14:56:11 CEST
Junio C Hamano	sam. 04 sept. 2010 17:17:09 CEST
Junio C Hamano	sam. 04 sept. 2010 17:15:36 CEST
Junio C Hamano	sam. 04 sept. 2010 07:45:58 CEST

Details:

- SHA : c2e0940b44dedb02fa02be95c35b231fe633c1
- Auteur : Jens Lehmann
- Date : lun. 06 sept. 2010 20:41:06 CEST
- Sujet : t3404 & t7509 cd inside subshell instead of around
- Parent : 4682693e9ccc04252d0fadef5e27fc05abdcba (Merge branch 'maint')

Code Editor (Bottom):

```
t3404-rebase-interactive.t
t7509-status.sh

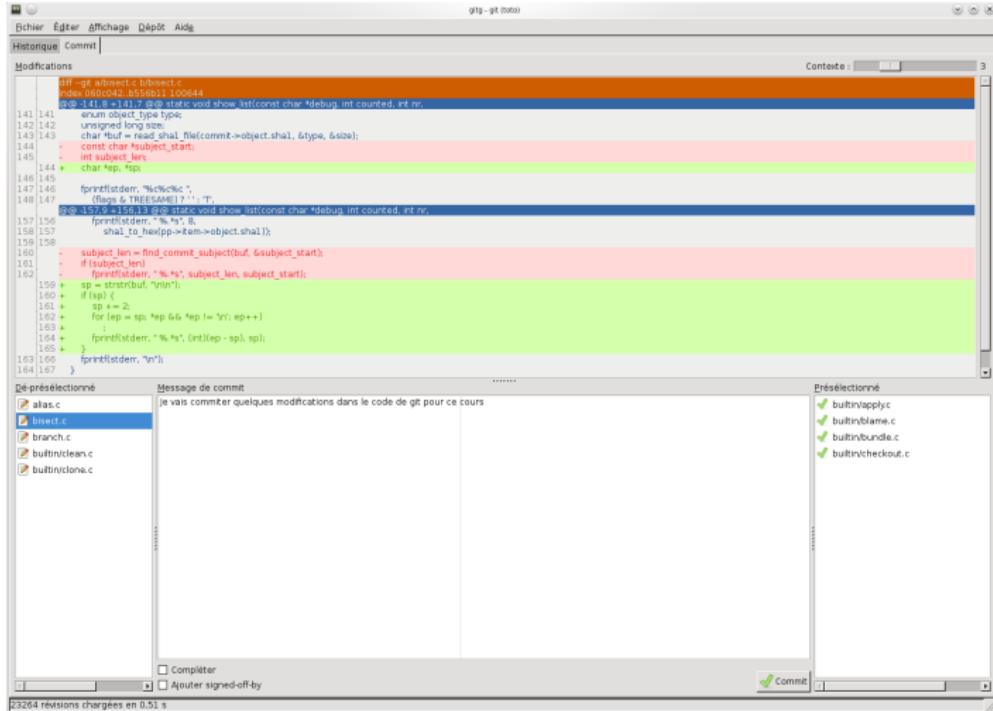
Found these places with "git grep -w \"cd \\".

Signed-off-by: Jens Lehmann <jens.lehmann@web.de>
Signed-off-by: Junio C Hamano <gitster@pobox.com>

JSH -git a/t3404-rebase-interactive.sh b/t3404-rebase-interactive.sh
index af3b663..7d20a74 100755
@@ -101,10 +101,10 @@ test_expect_success 'rebase -i with the exec command runs from root' '
101 102     git checkout master &&
103 103     git dir subdir && cd subdir &&
104 104     + mkdir subdir && cd subdir &&
105 105     FAKE_LINES="1 exec >touch-subdir"
106 106     - git rebase -i HEAD^ &&
107 107         + touch subfile &&
108 108         - rm subfile &&
109 109         + git add subfile &&
110 110         - git commit -m "add subfile" &&
111 111         + git rebase -i HEAD^ &&
112 112             - git checkout master &&
113 113             + rm subfile &&
114 114             - git add subfile &&
115 115             + git commit -m "rm subfile" &&
116 116             - git rebase -i HEAD^ &&
117 117                 + git checkout master &&
118 118                 - rm subfile &&
119 119                 + git add subfile &&
120 120                 - git commit -m "rm subfile" &&
121 121                 + git rebase -i HEAD^ &&
122 122                     - git checkout master &&
123 123                     + rm subfile &&
124 124                     - git add subfile &&
125 125                     + git commit -m "rm subfile" &&
126 126                     - git rebase -i HEAD^ &&
127 127                         + git checkout master &&
128 128                         - rm subfile &&
129 129                         + git add subfile &&
130 130                         - git commit -m "rm subfile" &&
131 131                         + git rebase -i HEAD^ &&
132 132                             - git checkout master &&
133 133                             + rm subfile &&
134 134                             - git add subfile &&
135 135                             + git commit -m "rm subfile" &&
136 136                             - git rebase -i HEAD^ &&
137 137                                 + git checkout master &&
138 138                                 - rm subfile &&
139 139                                 + git add subfile &&
140 140                                 - git commit -m "rm subfile" &&
141 141                                 + git rebase -i HEAD^ &&
142 142                                     - git checkout master &&
143 143                                     + rm subfile &&
144 144                                     - git add subfile &&
145 145                                     + git commit -m "rm subfile" &&
146 146                                     - git rebase -i HEAD^ &&
147 147                                         + git checkout master &&
148 148                                         - rm subfile &&
149 149                                         + git add subfile &&
150 150                                         - git commit -m "rm subfile" &&
151 151                                         + git rebase -i HEAD^ &&
152 152                                             - git checkout master &&
153 153                                             + rm subfile &&
154 154                                             - git add subfile &&
155 155                                             + git commit -m "rm subfile" &&
156 156                                             - git rebase -i HEAD^ &&
157 157                                                 + git checkout master &&
158 158                                                 - rm subfile &&
159 159                                                 + git add subfile &&
160 160                                                 - git commit -m "rm subfile" &&
161 161                                                 + git rebase -i HEAD^ &&
162 162                                                     - git checkout master &&
163 163                                                     + rm subfile &&
164 164                                                     - git add subfile &&
165 165                                                     + git commit -m "rm subfile" &&
166 166                                                     - git rebase -i HEAD^ &&
167 167                                                         + git checkout master &&
168 168                                                         - rm subfile &&
169 169                                                         + git add subfile &&
170 170                                                         - git commit -m "rm subfile" &&
171 171                                                         + git rebase -i HEAD^ &&
172 172                                                             - git checkout master &&
173 173                                                             + rm subfile &&
174 174                                                             - git add subfile &&
175 175                                                             + git commit -m "rm subfile" &&
176 176                                                             - git rebase -i HEAD^ &&
177 177                                                                 + git checkout master &&
```

23262 révisions chargées en 1.17 s

gitg : commit interactif



Outline

Git c'est quoi ?

Architecture interne de git

Gestion des versions dans le dépôt local.

Synchronisation avec les dépôts distants.

Les outils graphiques

Conclusion

Git vs Rhinocéros

Mercurial, Darcs, Bazaar, Arch, etc

Les points forts :

- + Certainement le plus rapide à appliquer des patchs
- + Simple à mettre en place (mode sans serveur)
- + Petits outils puissants (esprit Unix)
- + Git est distribué sous licence GNU GPL 2
- + Développement actif
- + Linus

Les points faibles :

- Courbe d'apprentissage
- Linus

Bibliographie sur le web

- ▶ Documentation officielle et usuelle : <http://www.kernel.org/pub/software/scm/git/docs/everyday.html>
- ▶ Documentation officielle complète : <http://www.kernel.org/pub/software/scm/git/docs/>
- ▶ Manuel de référence communautaire et en français :
<http://alexgirard.com/git-book/index.html>
- ▶ Manuel de référence communautaire (ultra complet) et en anglais :
<http://progit.org/book/>
- ▶ Tutoriel en vidéo et en anglais : <http://www.gitcasts.com/>
- ▶ Linus Torvalds parle de git chez Google (1 heure environ) :
<http://www.youtube.com/watch?v=4XpnKHJAok8>
- ▶ N'oubliez pas les *manpages*...

Références

- ▶ <http://download.ikaaro.org/doc/git/200607-ols.pdf>
- ▶ <http://kernel.org/pub/software/scm/cogito/>
- ▶ <http://git.or.cz/>
- ▶ <http://git.or.cz/gitwiki/>
- ▶ <http://www.kernel.org/pub/software/scm/git/docs/everyday.html>
- ▶ <http://www.kernel.org/pub/software/scm/git/docs/>