Gestion de versions

avec `git`

Walter Rudametkin

Avec les contributions de
M.E. Kessaci, O. Caron, J. Dequidt, F. Boulier

Walter.Rudametkin@polytech-lille.fr

`https://rudametw.github.io/teaching/`

Bureau F011

Moi... *(et ma décharge de responsabilité)*

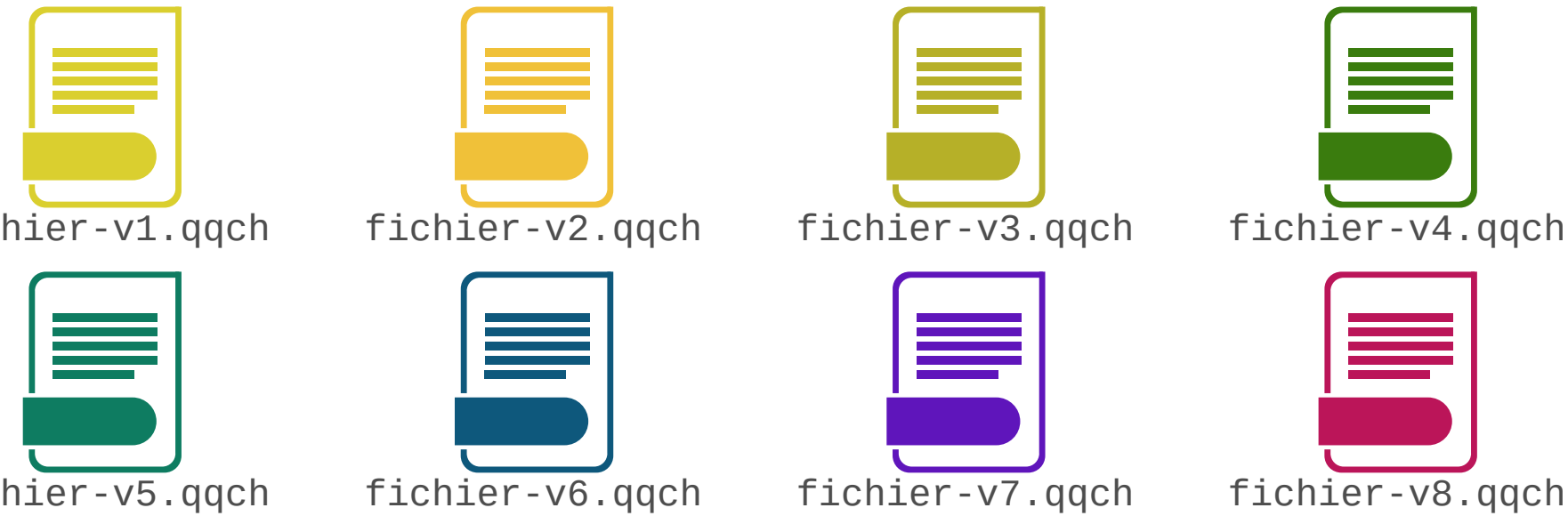
- Je suis étranger (hors UE)
- J'ai un accent
- Je me trompe beaucoup en français
  - et en info, et en math, et ...
  - n'hésitez pas à me corriger ou à me demander de répéter
- Je commence à enseigner
  - ce cours est tout nouveau
  - j'accepte des critiques (constructives mais pas que) et surtout des recommandations
  - n'hésitez pas à poser des questions
- Je ne suis pas un expert de Git

Comment gérez-vous vos fichiers ?

- Garder l'historique
- Partager

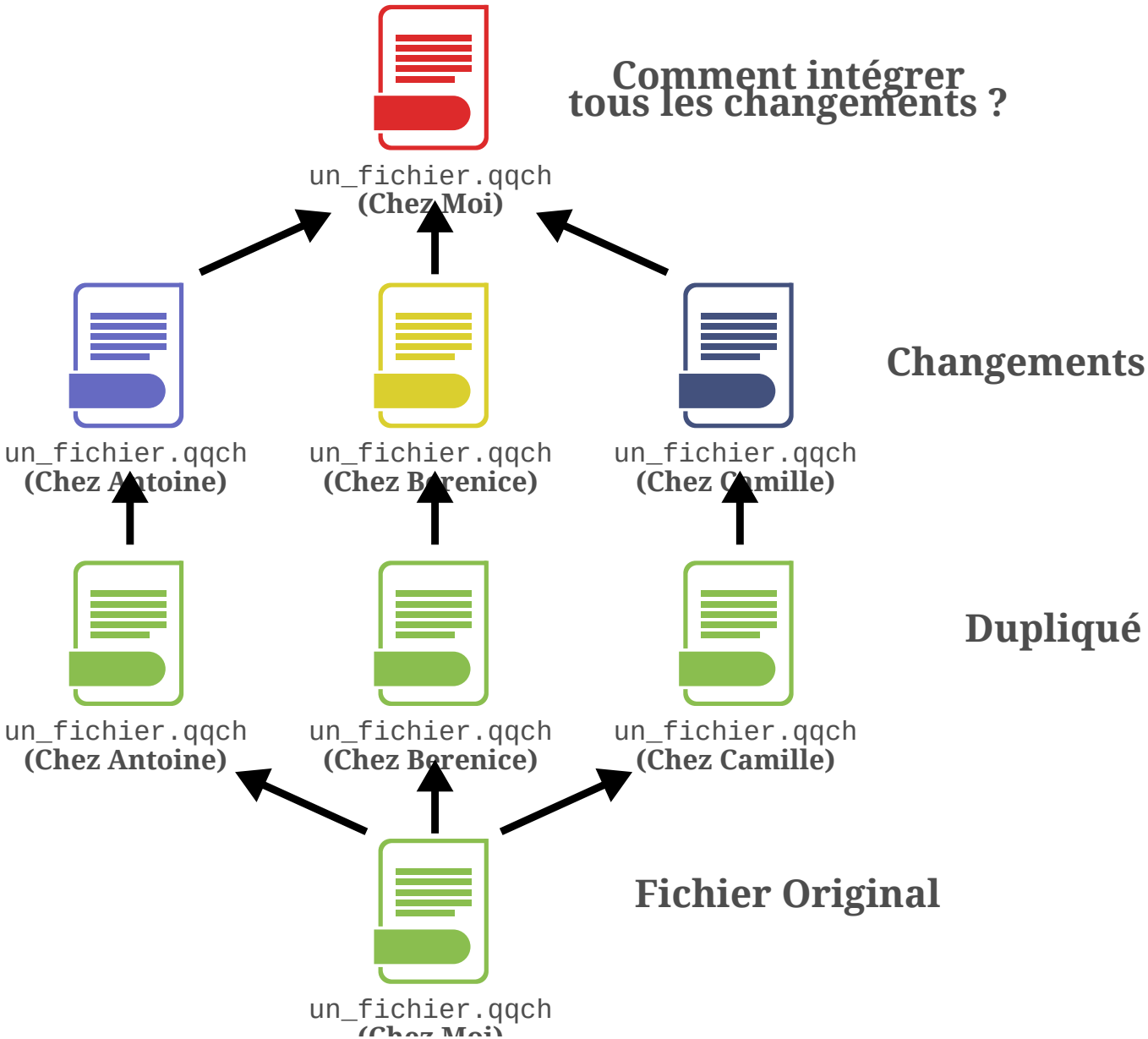
Comment gérez-vous vos fichiers ?

- Garder l'historique
- Partager

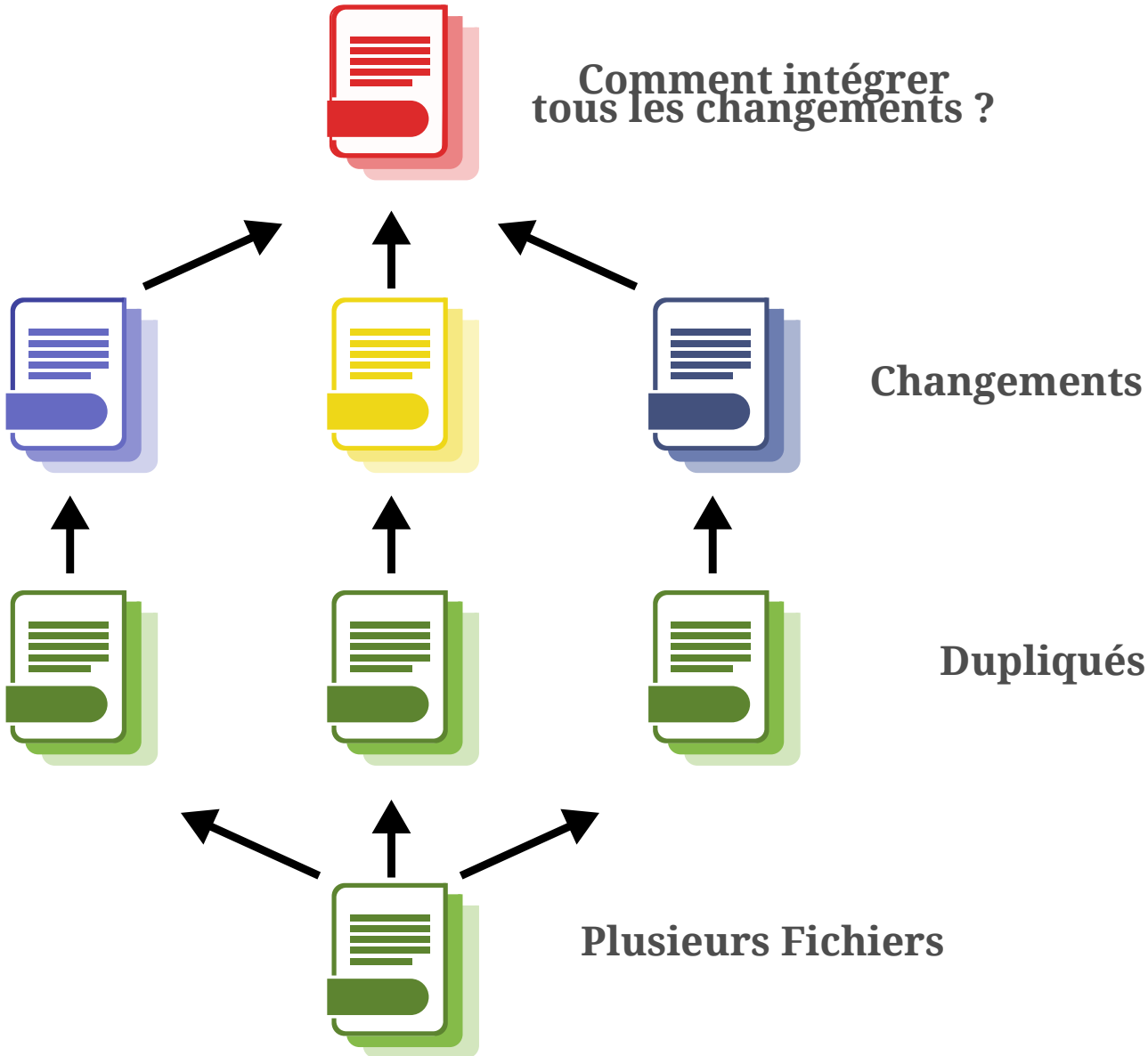


Versionnement manuel de fichiers

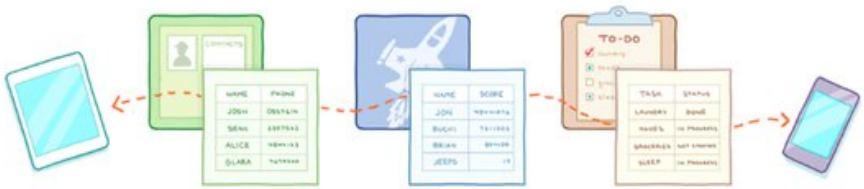
Comment collaborer sur un fichier ?



Comment collaborer sur plusieurs fichiers ?



D’autres solutions ?



Dropbox

Google docs

Google Drive

OneDrive

DROPBOX

Problématique : développement logiciel

- Un projet de développement logiciel est une activité longue et complexe.
- Concerne plusieurs fichiers (milliers !)
- De multiples itérations sont nécessaires.
- A certains moments, on peut identifier des versions et/ou variantes du logiciel.
- Les erreurs sont possibles, revenir en arrière est parfois nécessaire.
- Un projet peut se faire a plusieurs, les développeurs peuvent travailler sur les memes fichiers (conflits)

Définitions

Simple

- Un **gestionnaire de versions** est un logiciel qui **enregistre les évolutions dun ensemble de fichiers** au cours du temps de manière a ce qu'on puisse rappeler une version antérieure à tout moment.

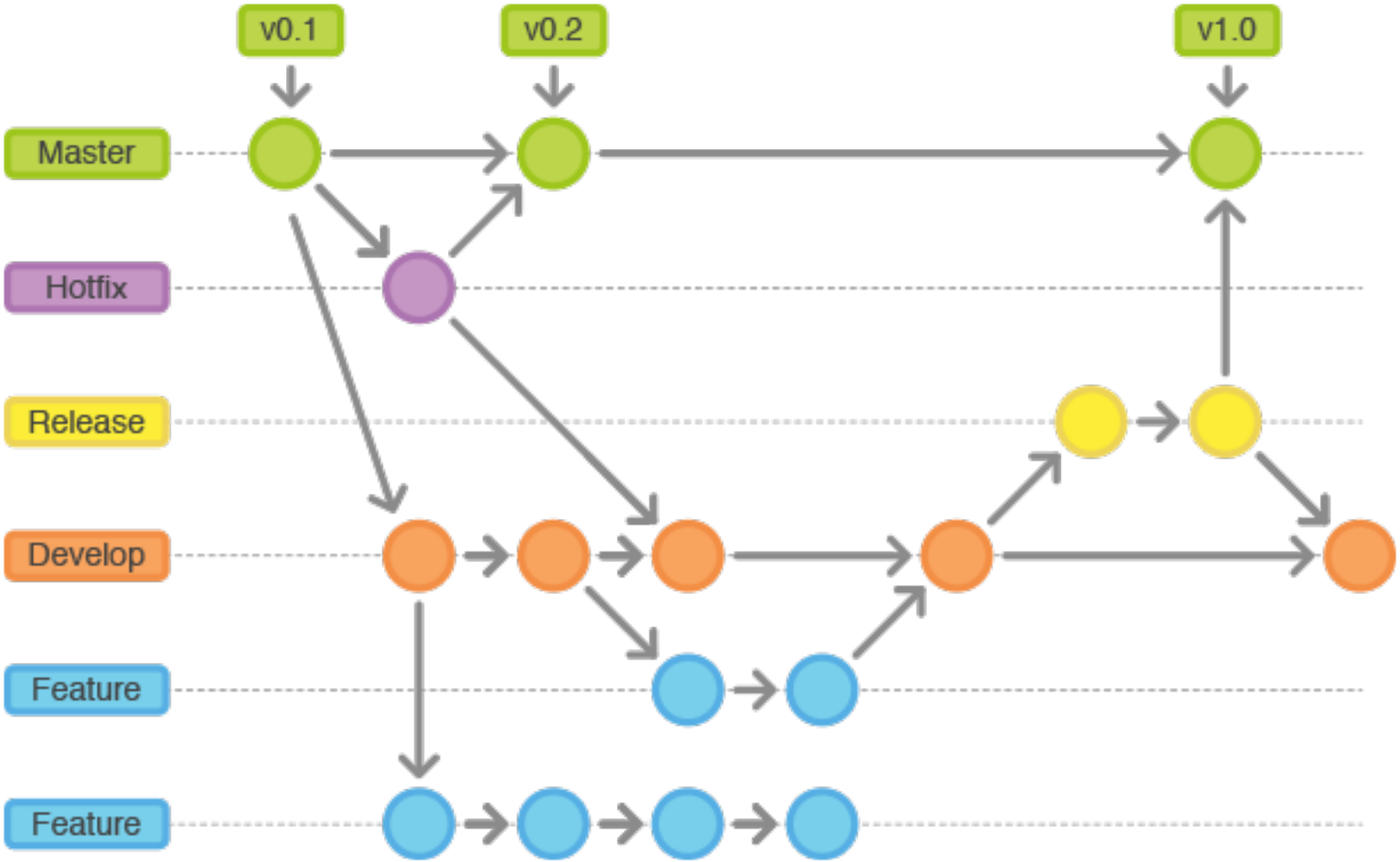
Définition Wikipedia[1]

- La **gestion de versions** (en anglais *version control* ou *revision control*) consiste à maintenir **l'ensemble des versions d'un ou plusieurs fichiers** (généralement en texte). Essentiellement utilisée dans le domaine de la création de logiciels, elle concerne surtout **la gestion des codes source**.

[1] `https://fr.wikipedia.org/wiki/Gestion_de_versions`

Gestion de versions

Le développement logiciel est un processus sinueux à notion de branche (chaque noeud représente un ensemble de fichiers à un temps t) :



Avantages de la gestion de versions

- Sauvegarde / Restauration
- Synchronisation du travail (partage, collaboration)
- Suivi de changements (très détaillé)
- Suivi de responsabilités / propriétaires / coupables
- *Sandboxing* (espace confiné, environnement de test, isolatio
- *Branching and merging*
- Passage à l'échelle (10, 100, 1.000, 10.000 développeurs)

Que mettre dans un Logiciel de Gestion de Versions ?

- Tous les sources du projet
  - code source (`.c .cpp .java .py ...`)
  - scripts de build (`Makefile pom.xml ...`)
  - Documentation (`.txt .tex Readme ...`)
  - Ressources (images ...)
  - Scripts divers (déploiement, `.sql, .sh ...`)

Que mettre dans un Logiciel de Gestion de Versions ?

- Tous les sources du projet
  - code source (`.c .cpp .java .py ...`)
  - scripts de build (`Makefile pom.xml ...`)
  - Documentation (`.txt .tex Readme ...`)
  - Ressources (images ...)
  - Scripts divers (déploiement, `.sql, .sh ...`)

À NE PAS METTRE

- Les fichiers générés
  - Résultat de compilation (`.class .o .exe .jar ...`)
  - Autres fichiers générés (`.ps .dvi .pdf javadoc ...`)

Why *the* `git` ?

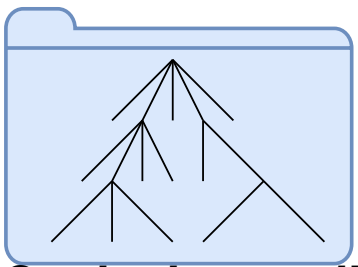
C'est *Ze Standard*

- *git - the stupid content tracker*
- Linus Torvalds (2005)
- Outil professionnel, rapide, multi-plateforme, flexible, puissant, complètement distribué

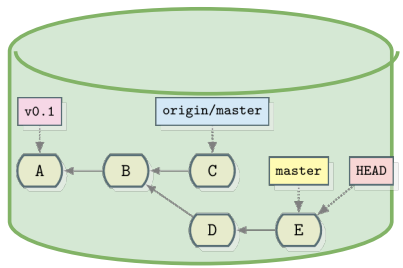
To Share or Not to Share ?

- Enrichissez vos CV
  - Faites un compte sur `https://github.com/`
- Choisir sa licence
  - Code — GPL, Apache, BSD, MIT, Propriétaire `https://choosealicense.com/`
  - Documents/Rapports — Creative commons `https://creativecommons.org/`

Concepts et commandes `git`

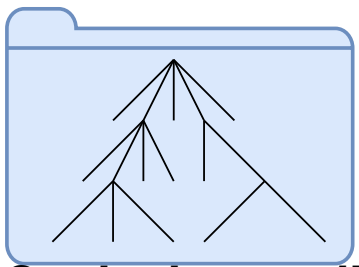


Copie de travail
(Working Directory)

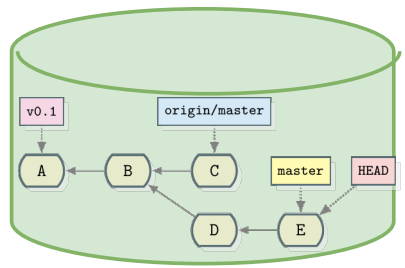


Dépôt

Concepts et commandes `git`

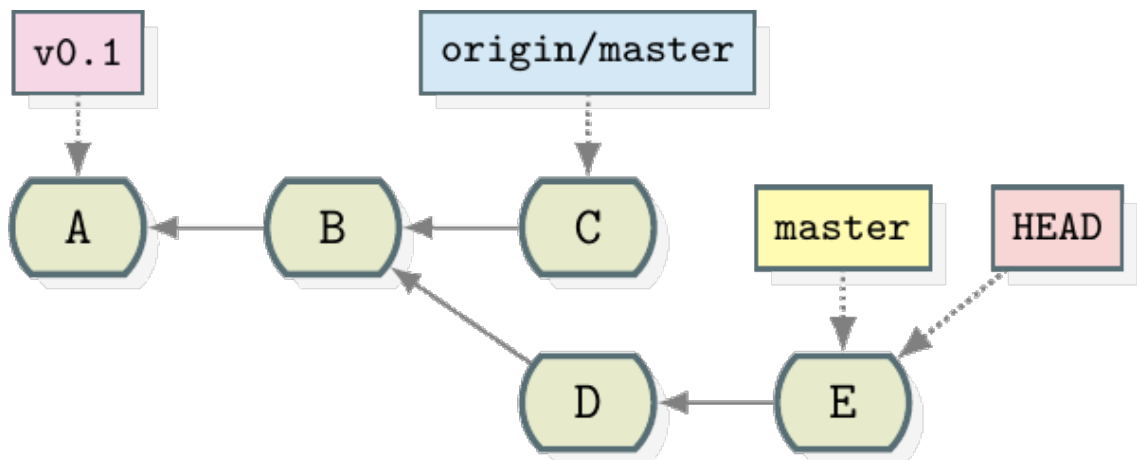


Copie de travail
(Working Directory)



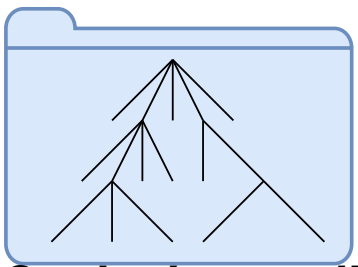
Dépôt

```
.
├── bin
│   └── program.exe
├── build
├── doc
├── INSTALL
├── lib
│   ├── smixer-hda.so
│   └── libgui.a
├── LICENSE
├── Makefile
├── README
└── src
    ├── program.c
    ├── utils.c
    └── utils.h
```

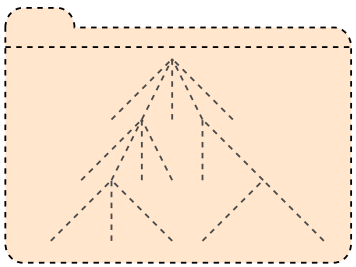


Concepts et commandes `git`

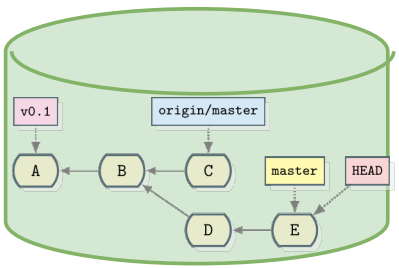
Réseau



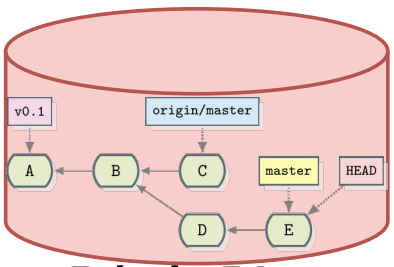
Copie de travail
(Working Directory)



Index (Stage)



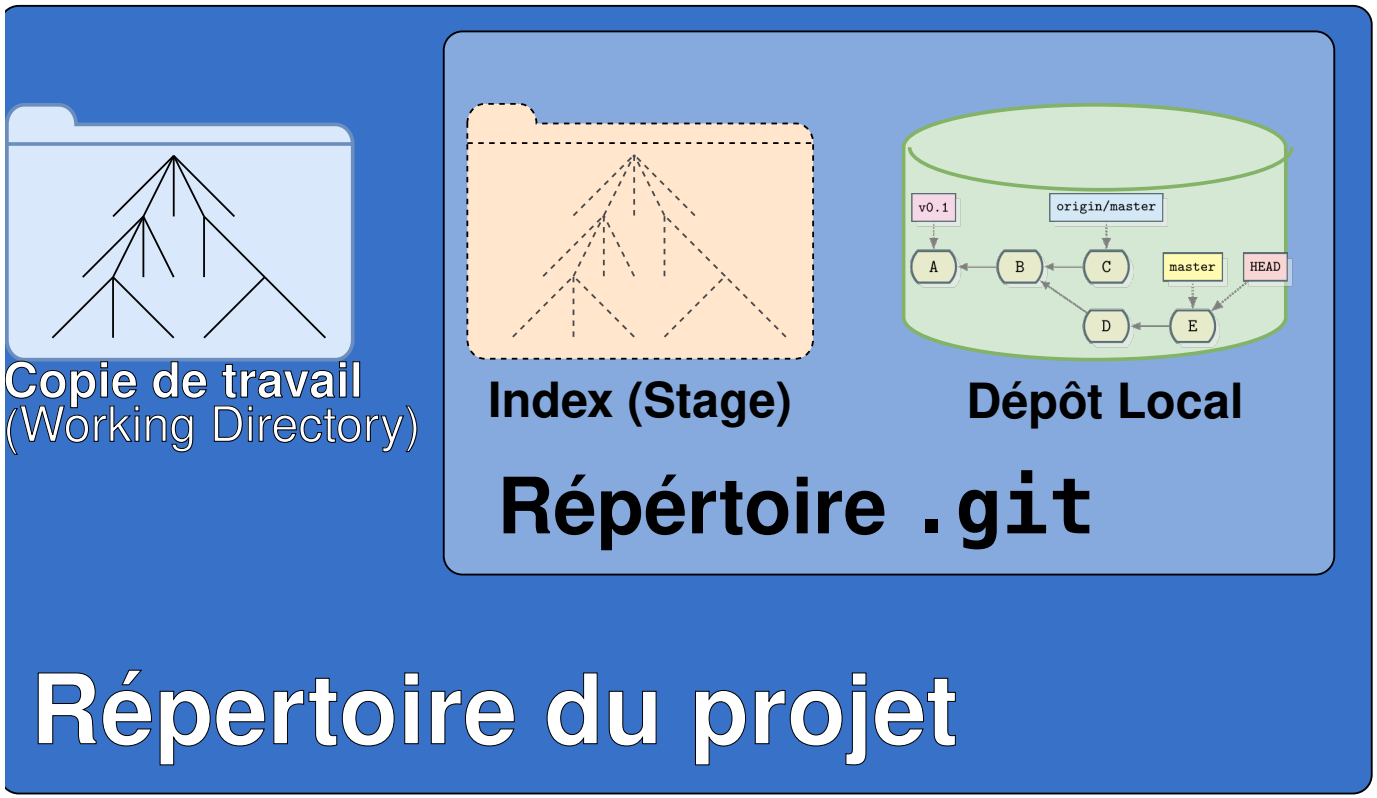
Dépôt Local

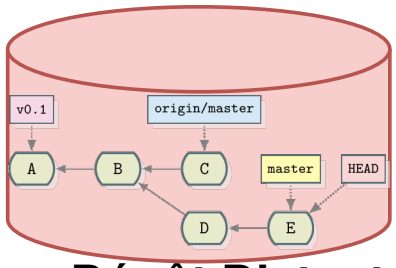


Dépôt Distant
(Remote)

Concepts et commandes `git`

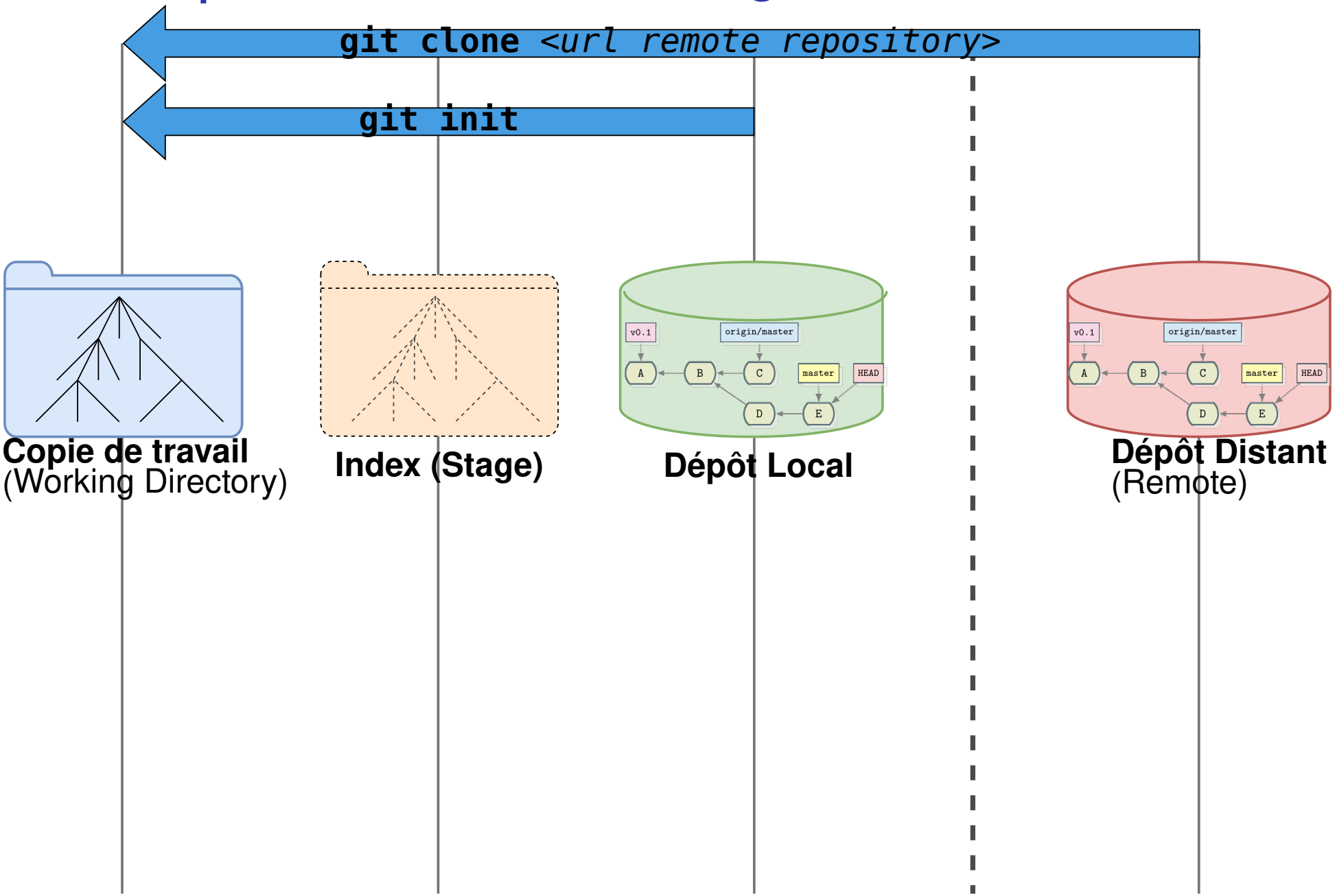
Réseau



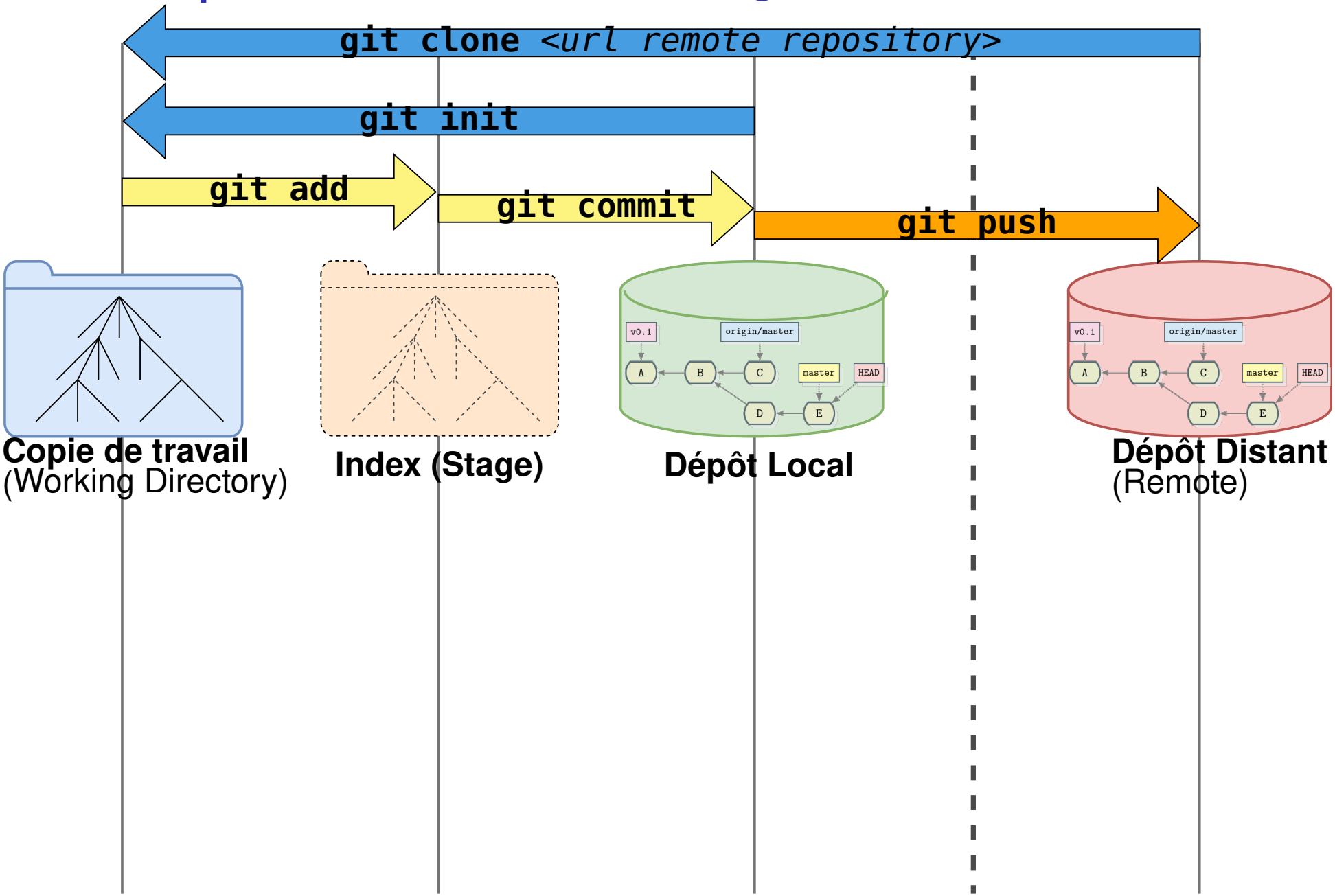


Dépôt Distant
(Remote)

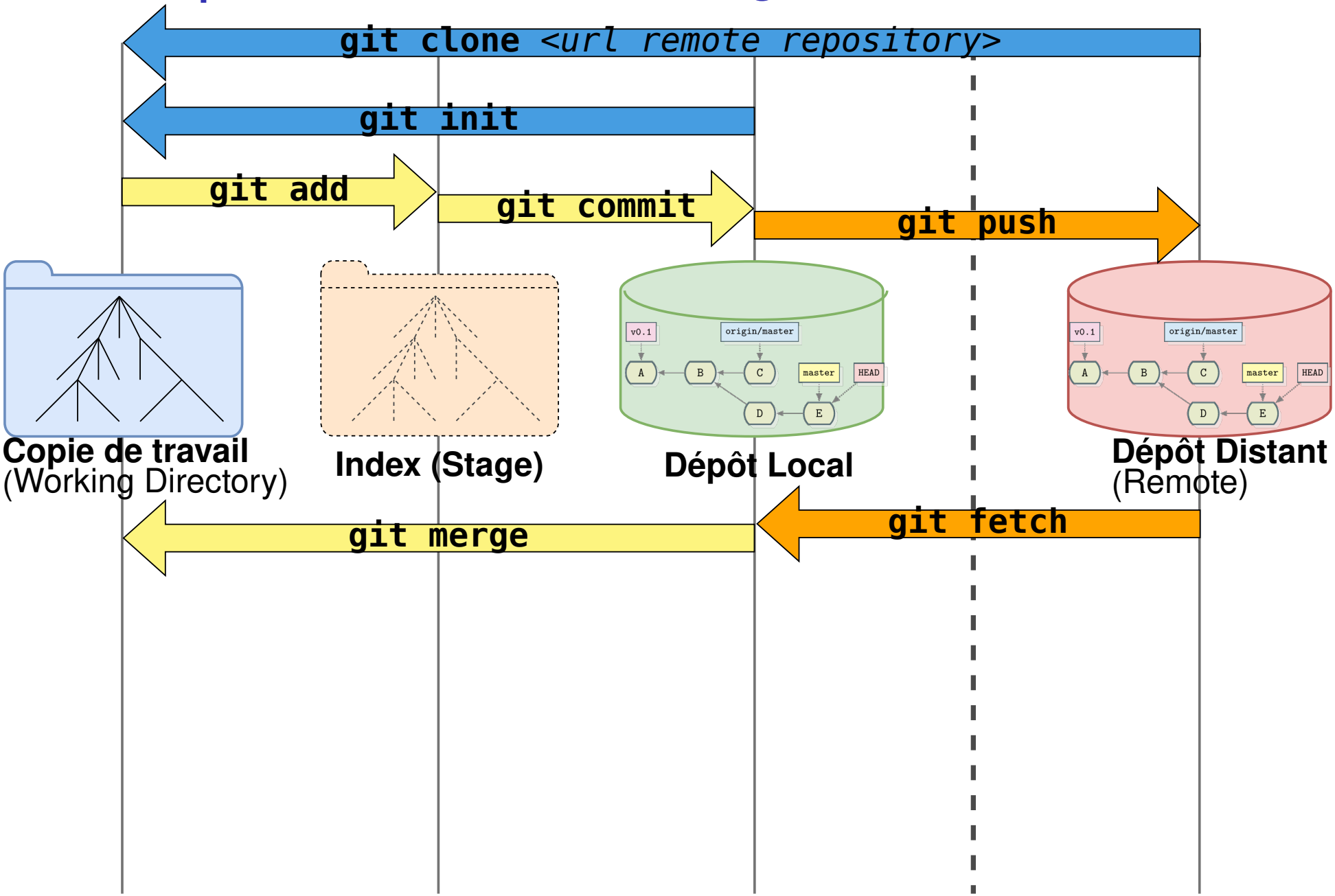
Concepts et commandes `git`



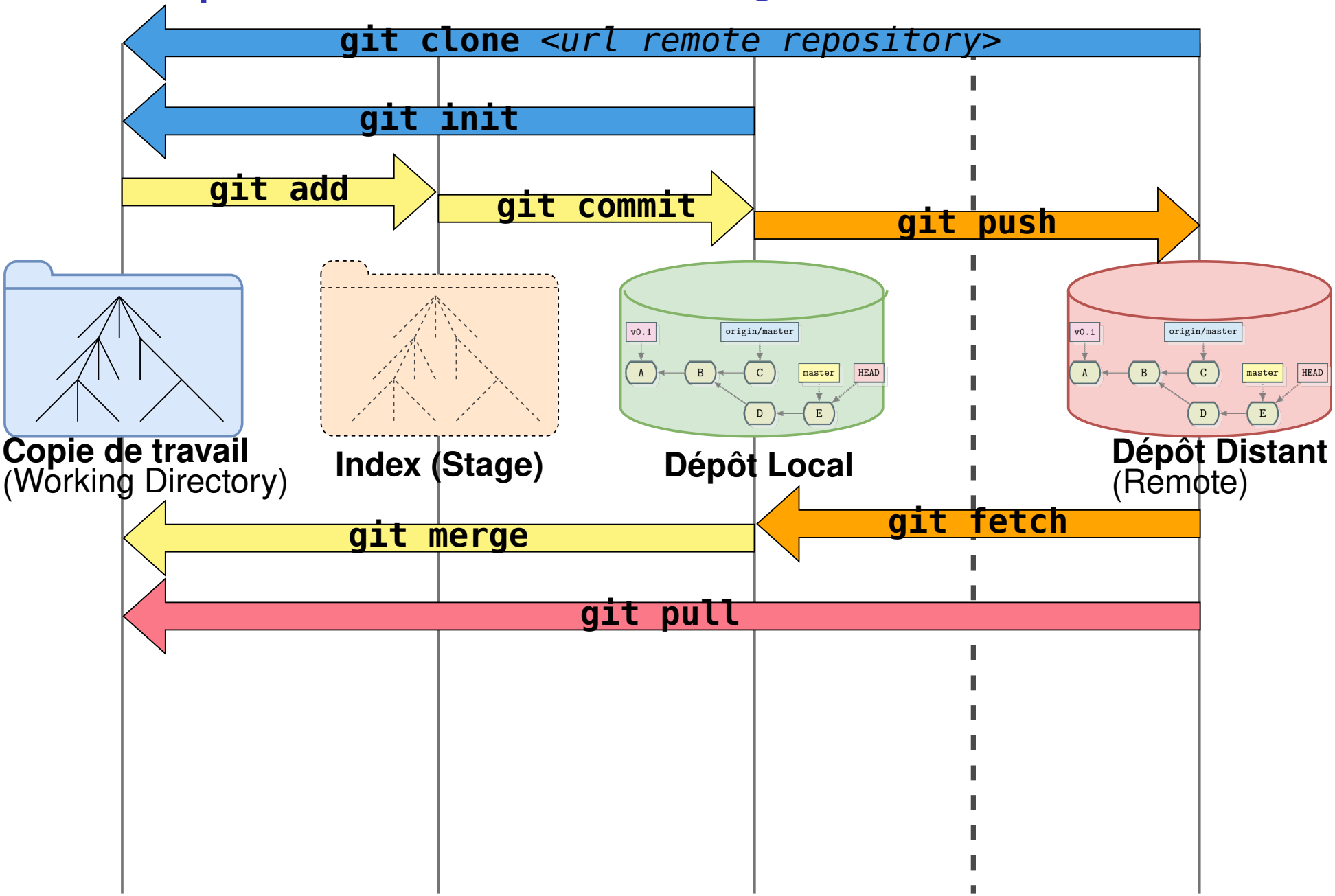
Concepts et commandes `git`



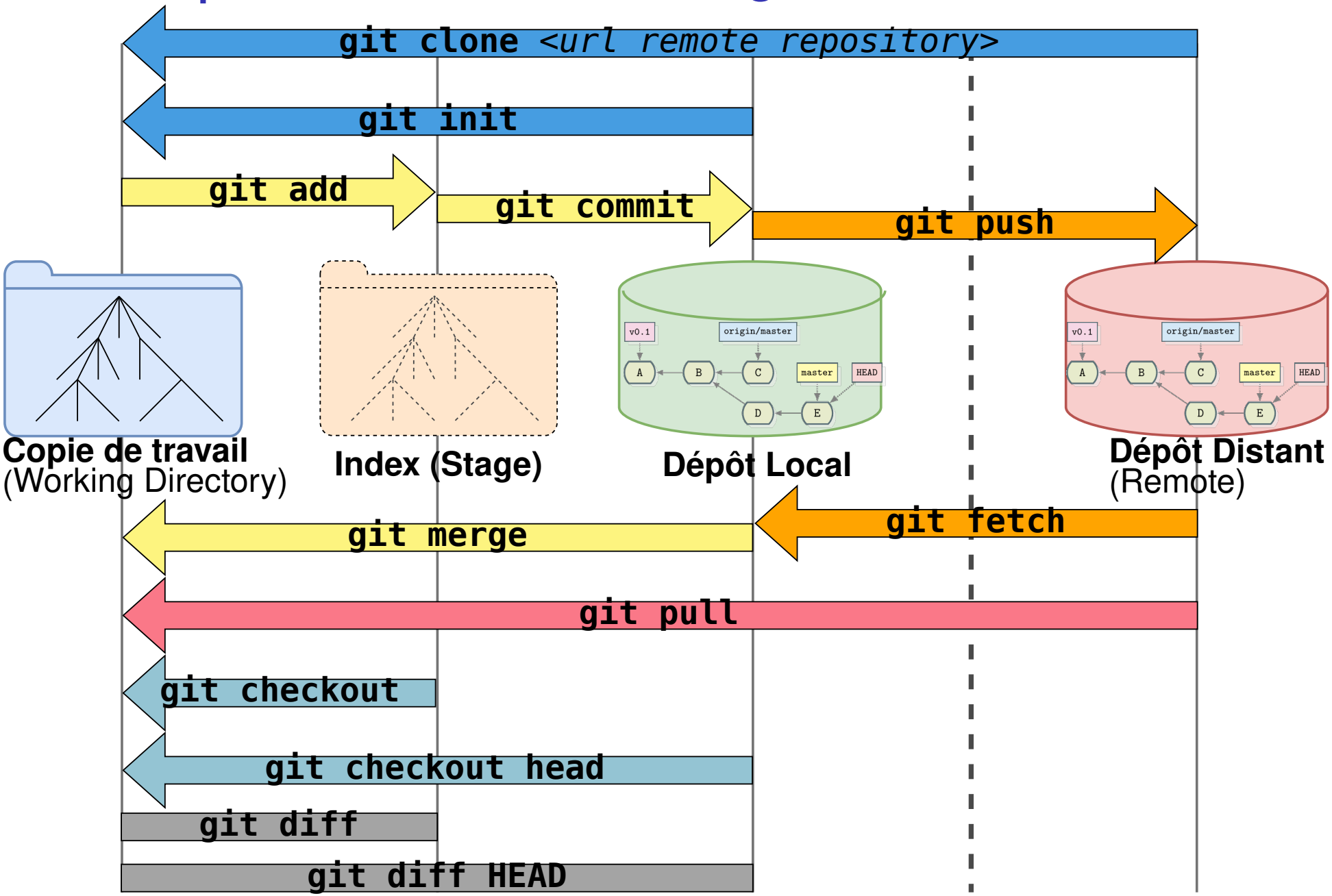
Concepts et commandes `git`



Concepts et commandes `git`



Concepts et commandes `git`



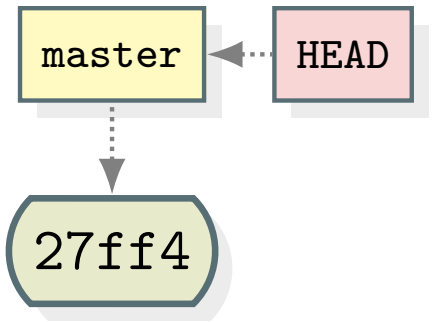
Le *Graphe Orienté Acyclique* de `commits`

(a) Dépôt vide

Dans un terminal . . .

```
mkdir mon_depot ; cd mon_depot
git init .
echo "pomme" >> fruits.txt
git add fruits.txt
git commit -m "Pomme ajouté à la liste de fruits"
  ⇒ ID = 27ff4
```

Le *Graphe Orienté Acyclique* de `commits`



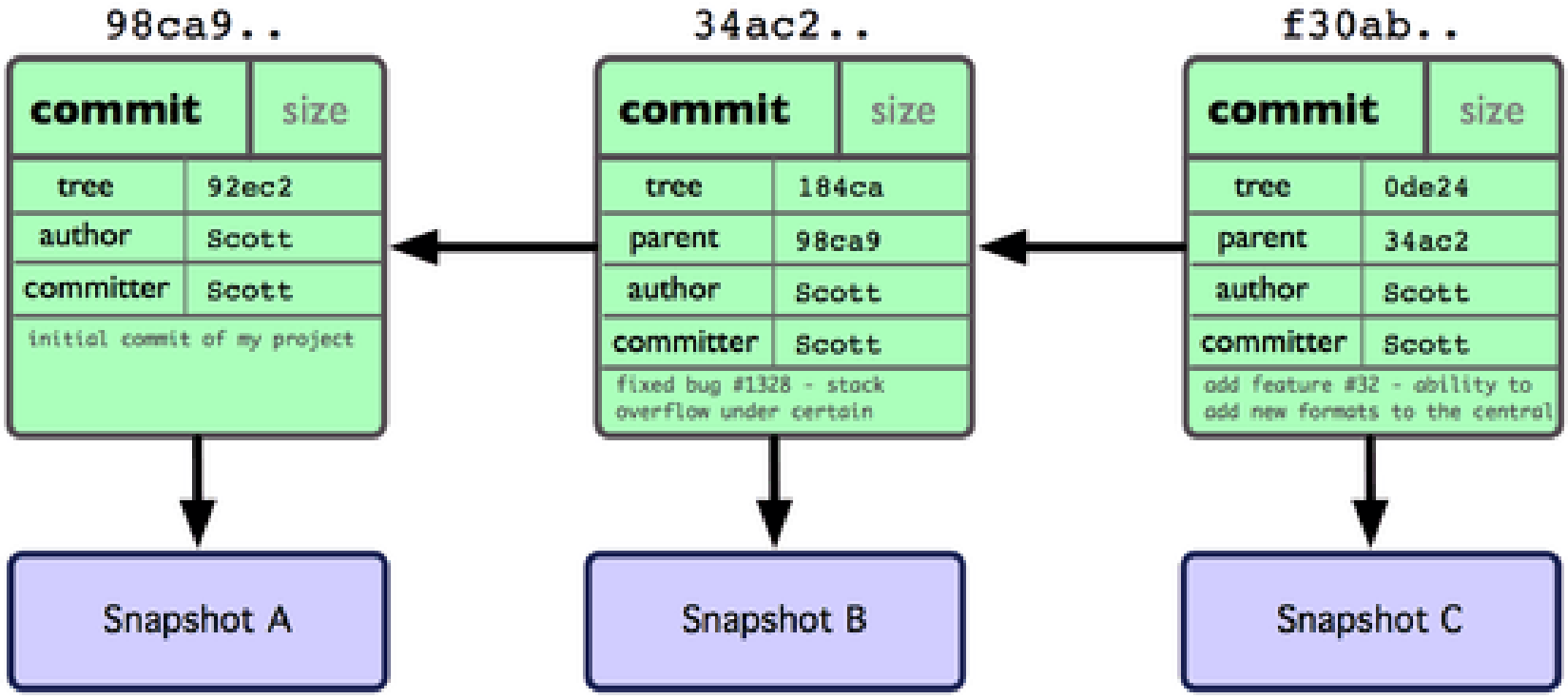
(a) Premier *commit*

Dans un terminal . . .

```
mkdir mon_depot ; cd mon_depot
git init .
echo "pomme" >> fruits.txt
git add fruits.txt
git commit -m "Pomme ajouté à la liste de fruits"
  ⇒ ID = 27ff4
```

Faire **`git status`** et **`git log`** après toute commande!

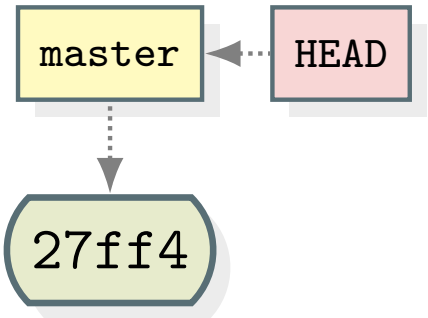
C’est quoi un commit ?



- Le `Commit-ID` est une *empreinte* calculé en utilisant la fonction de hachage `SHA-1` sur
  - **Tout** le contenu du commit + Date + Nom et email du commiteur + Message de log + ID du commit parent + ...

Propriété : **Unicité** quasi-universelle de l’`ID`

Le Graphe : Commit 2

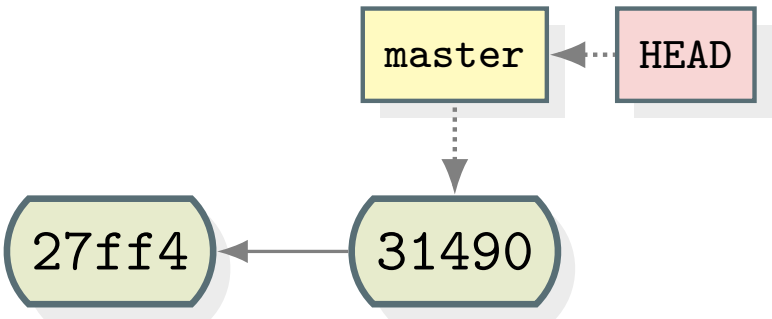


(a) État avant deuxième `commit`

Dans un terminal . . .

```
↪ echo banane >> fruits.txt
  git add fruits.txt
  git commit -m "Ajouté banane à fruits.txt"
    ⇒ ID = 31490
```

Le Graphe : Commit 2



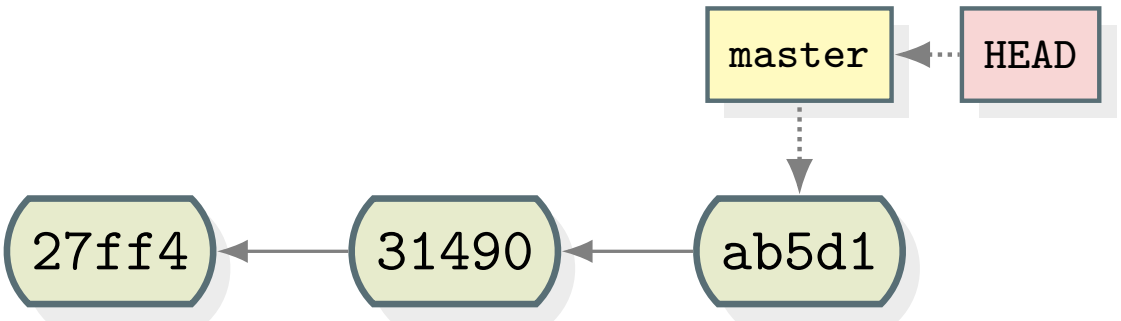
(a) Deuxième *commit*

Dans un terminal . . .

```
echo banane >> fruits.txt
git add fruits.txt
git commit -m "Ajouté banane à fruits.txt"
  ⇒ ID = 31490
```

↪

Le Graphe : Commit 3

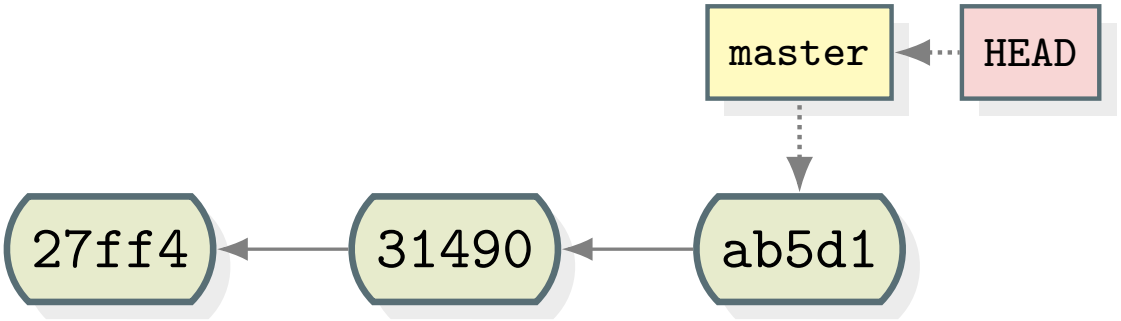


(a) Troisième *commit*

Dans un terminal . . .

```
echo orange >> fruits.txt
git add fruits.txt
git commit -m "Ajouté orange à fruits.txt"
  ⇒ ID = ab5d1
```

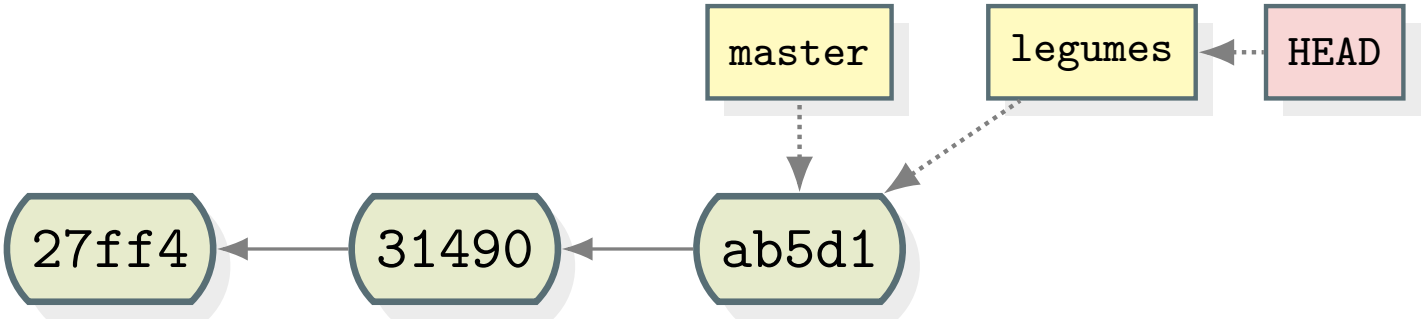
Le Graphe : Branche `legumes`



(a) Avant branche

```
↪ git branch legumes ; git checkout legumes
```

Le Graphe : Branche `legumes`

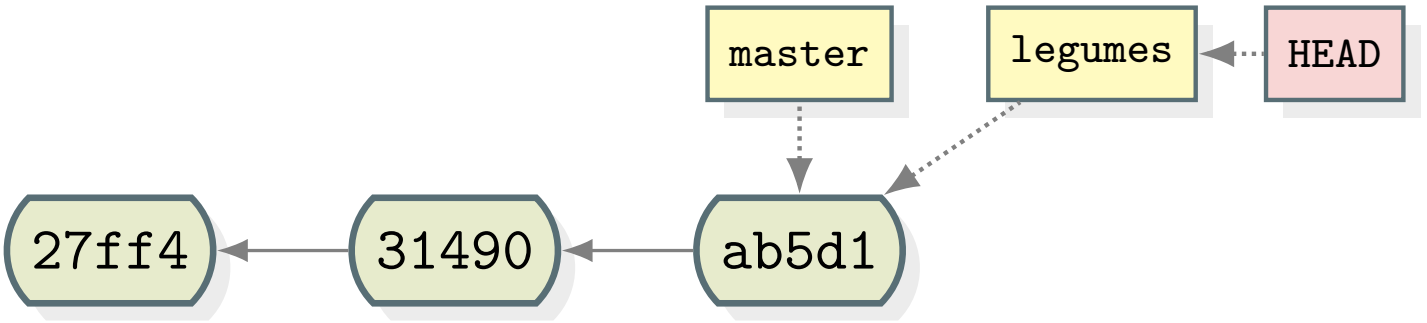


(a) Après branche

$\Rightarrow$ une nouvelle *étiquette* (`legumes`) apparait, elle pointe vers le commit courant (`ab5d1`), et la commande `checkout` fait pointer `HEAD` sur `legumes`

```
↪ git branch legumes ; git checkout legumes
```

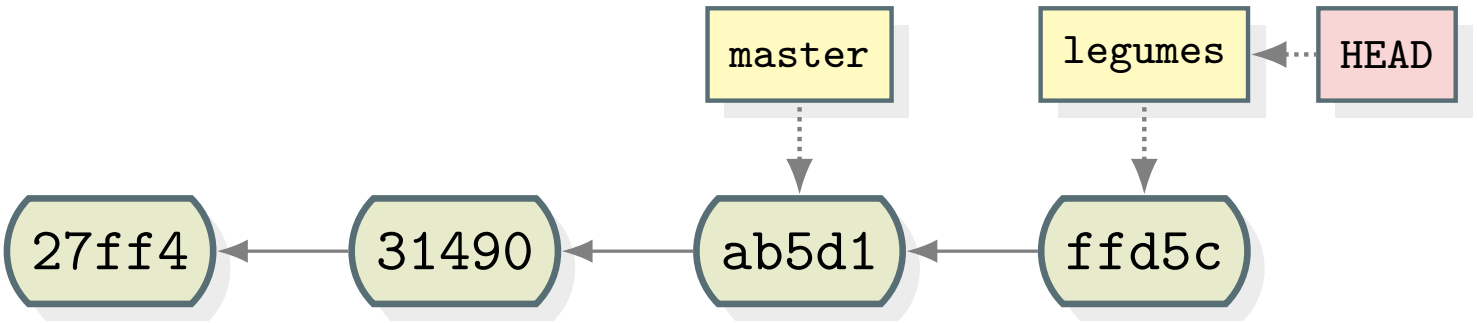
Le Graphe : Branche `legumes`



(a) Après branche

```
git branch legumes ; git checkout legumes
↪ echo aubergine >> legumes.txt ; git add legumes.txt
git commit -m "Ajout aubergine à legumes"
  ⇒ ID = ffd5c
```

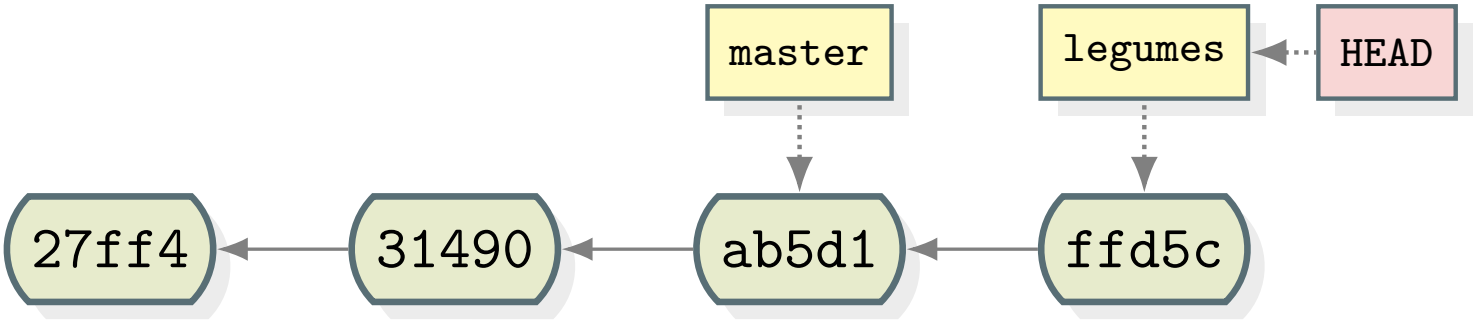
Le Graphe : Branche `legumes`



(a) Après un premier commit dans la branche `legumes`

```
git branch legumes ; git checkout legumes
echo aubergine >> legumes.txt ; git add legumes.txt
git commit -m "Ajout aubergine à legumes"
  ⇒ ID = ffd5c
```

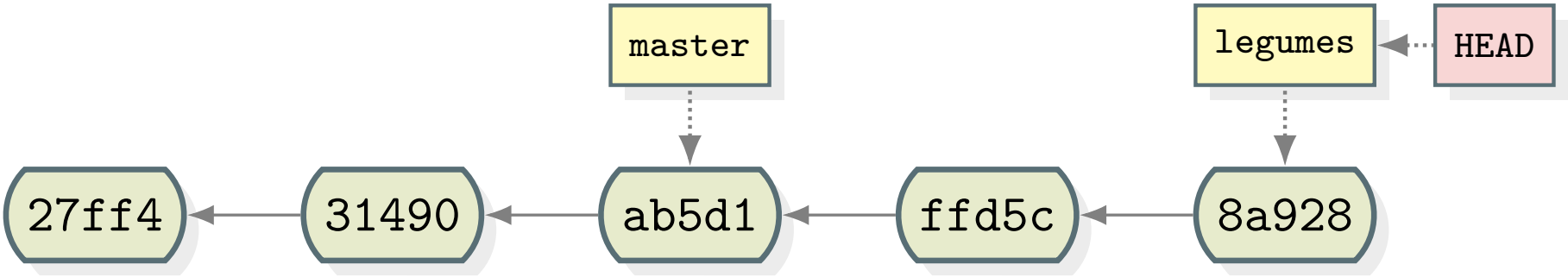
Le Graphe : Branche `legumes`



(a) Après un premier commit dans la branche `legumes`

```
git branch legumes ; git checkout legumes
echo aubergine >> legumes.txt ; git add legumes.txt
git commit -m "Ajout aubergine à legumes"
  ⇒ ID = ffd5c
↪
echo courgette >> legumes.txt ; git add legumes.txt
git commit -m "Ajout courgette à legumes"
  ⇒ ID = 8a928
```

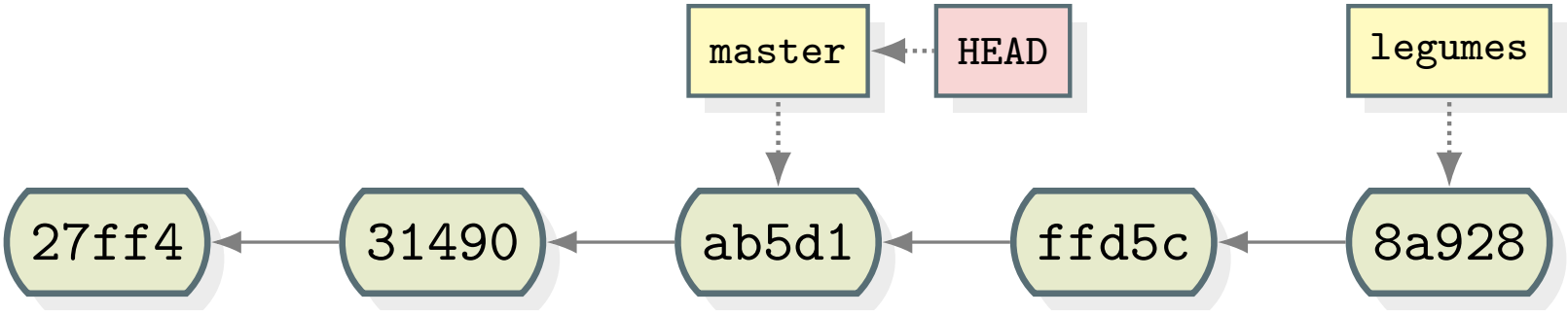
Le Graphe : Branche `legumes`



(a) Après un deuxième commit dans la branche `legumes`

```
git branch legumes ; git checkout legumes
echo aubergine >> legumes.txt ; git add legumes.txt
git commit -m "Ajout aubergine à legumes"
  ⇒ ID = ffd5c
echo courgette >> legumes.txt ; git add legumes.txt
git commit -m "Ajout courgette à legumes"
  ⇒ ID = 8a928
```

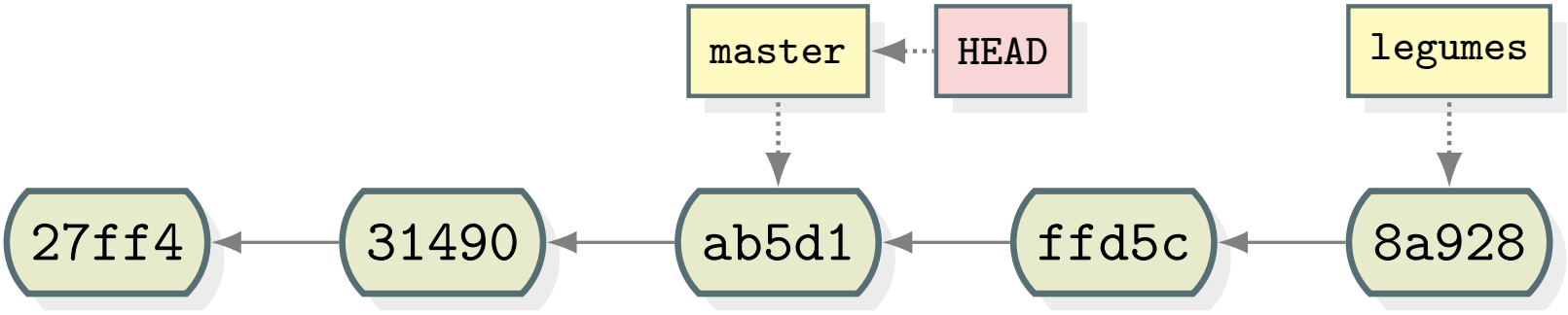
Le Graphe : Branche `master`



(a) Travaillons sur `master`

```
    ...
    git checkout master
↪
```

Le Graphe : Branche `master`

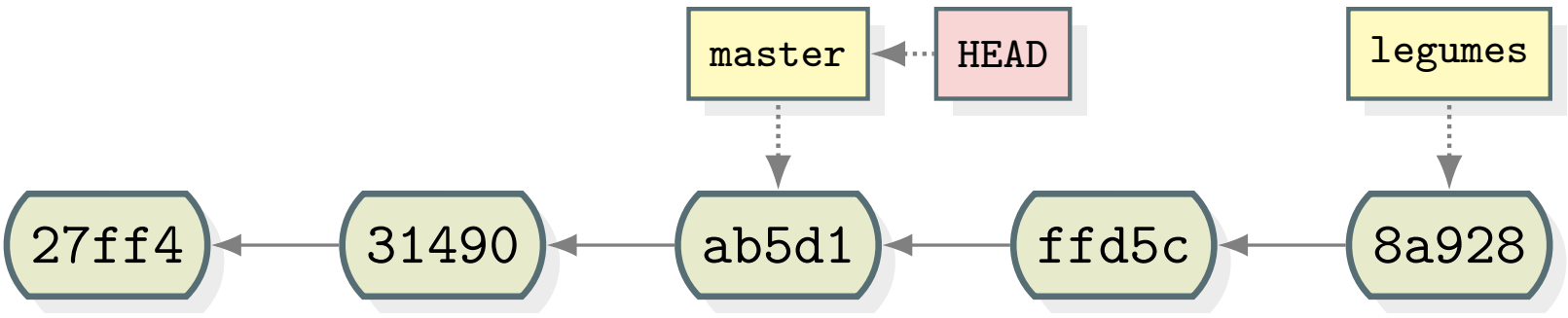


(a) Travaillons sur `master`

$\Rightarrow$ `legumes.txt` n'existe plus dans la Copie de Travail (*Working Directory*)

```
    ...
    git checkout master
↪
```

Le Graphe : Branche `master`

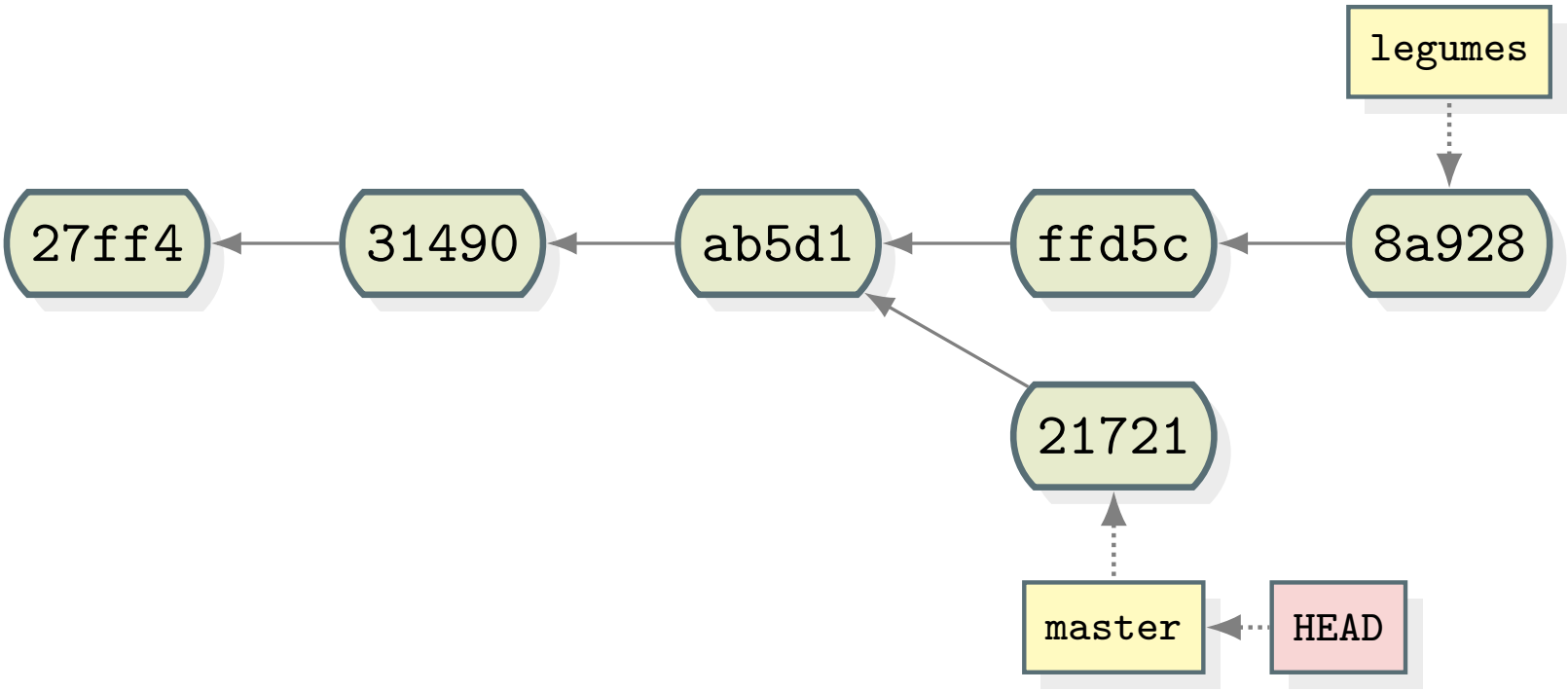


(a) Et si on commite sur `master` ?

```
    ...
    git checkout master
↪

    echo poire >> fruits.txt ; git add fruits.txt
    git commit -m "Ajouté poire à fruits.txt"
        ⇒ ID = 21721
```

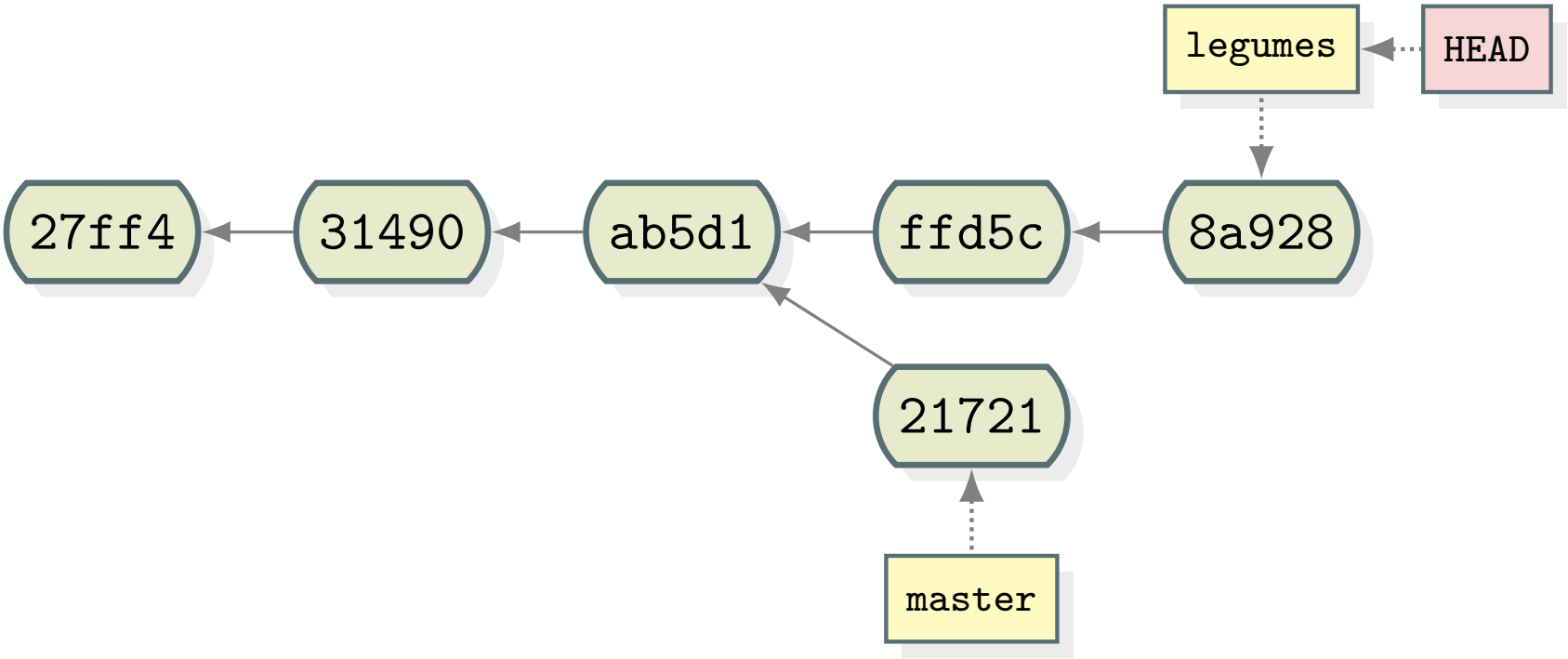
Le Graphe : Branche `master`



(a) Après un nouveau *commit* sur `master`

```
...
git checkout master
echo poire >> fruits.txt ; git add fruits.txt
git commit -m "Ajouté poire à fruits.txt"
    ⇒ ID = 21721
↪
```

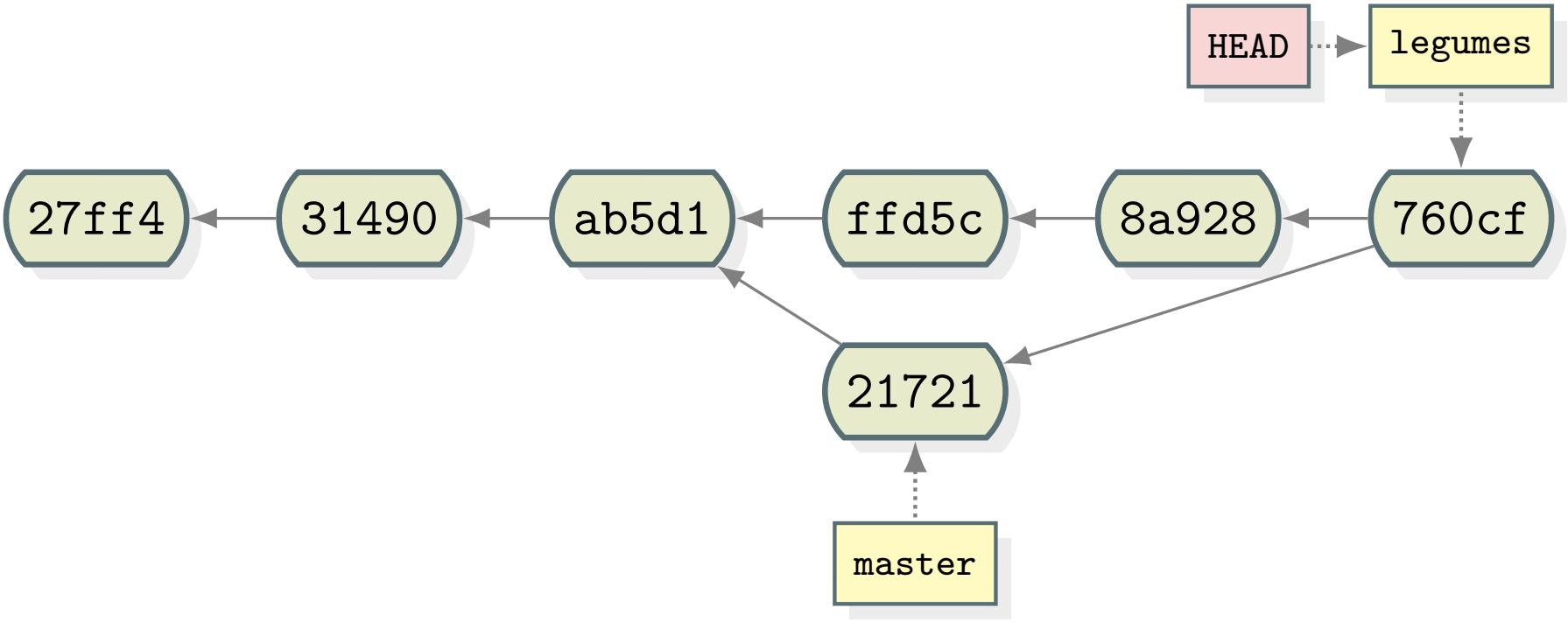
Le Graphe : Merge `master`⇒`legumes`



(a) Allons sur `légumes`, regardons les différences avec `diff`

```
git checkout legumes
git diff master
↪ git merge master
```

Le Graphe : Merge `master`⇒`legumes`



(a) Merger `master` dans `légumes` : produit un nouveau commit

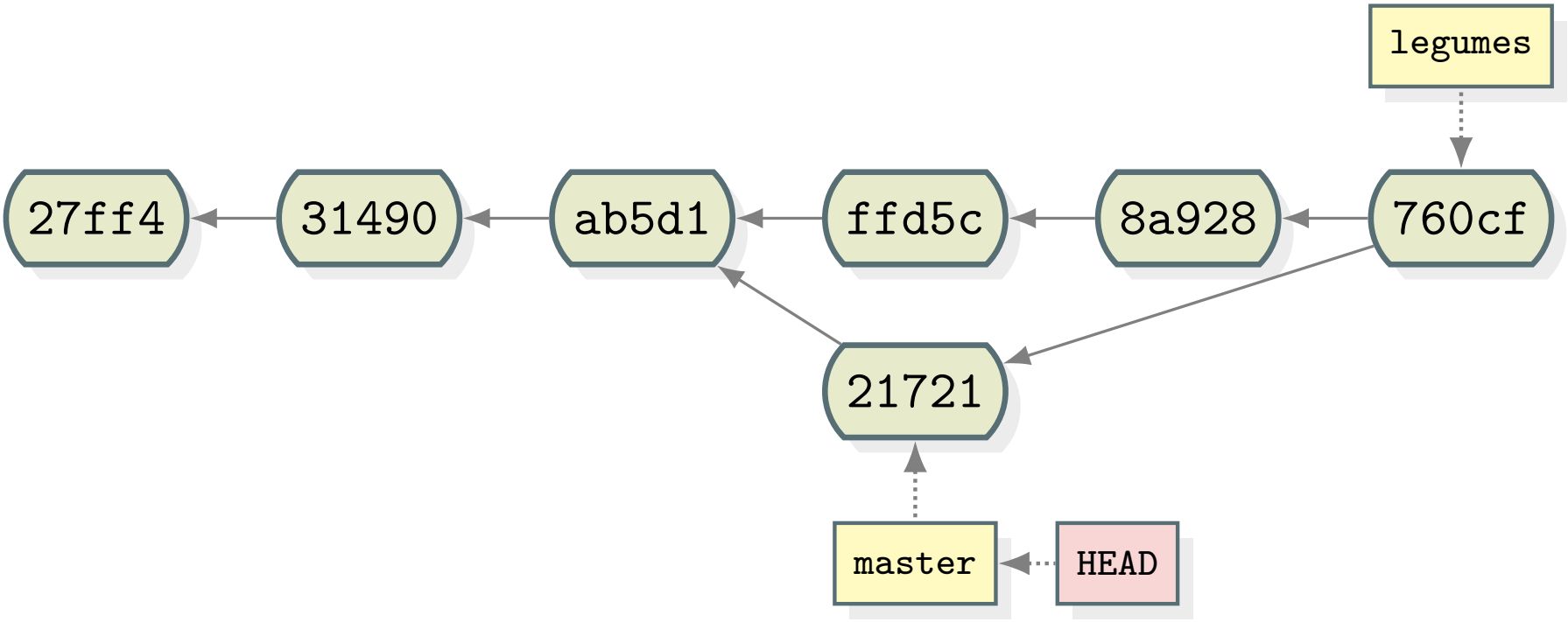
```
git checkout legumes
git diff master
git merge master
```
↪

Merge : Vue dans la console

```
wrudamet@beaner[legumes L|✔] ~/COURS/Git/mon_depot $ git l
*   760cf0e [2017-12-01] (HEAD -> refs/heads/legumes) Merge branch 'master' into legumes [rud
|\
| * 8a928c9 [2017-12-01] (refs/heads/master) Ajouté poire à fruits.txt [rudametw]
* | 1888830 [2017-12-01] Ajout courgette à legumes [rudametw]
* | ffd5c3e [2017-12-01] Ajout de legumes [rudametw]
|/
* ab5d1c0 [2017-12-01] Ajouté orange à fruits.txt [rudametw]
* 3149017 [2017-12-01] Ajouté banane à fruits.txt [rudametw]
* 27ff4c1 [2017-11-30] Pomme ajouté à la liste de fruits [rudametw]
```

`git log --all --graph --oneline --date=short`

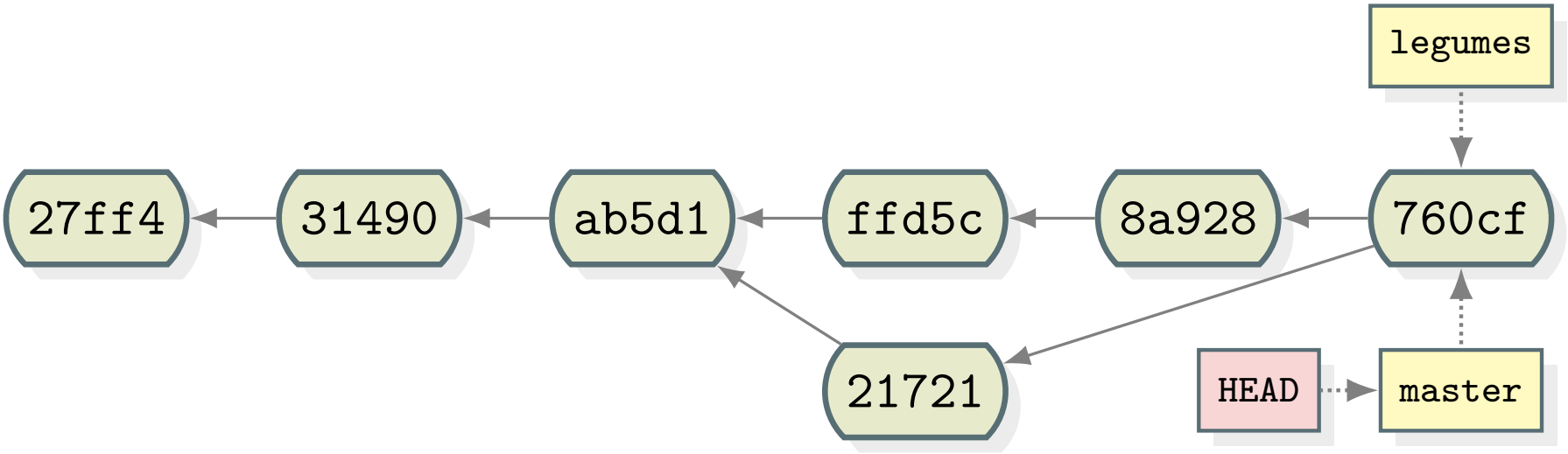
Le Graphe : Merge `legumes`⇒`master`



(a) Allons sur `master`

```
git checkout master
git diff legumes
git merge legumes
git branch -d legumes
```

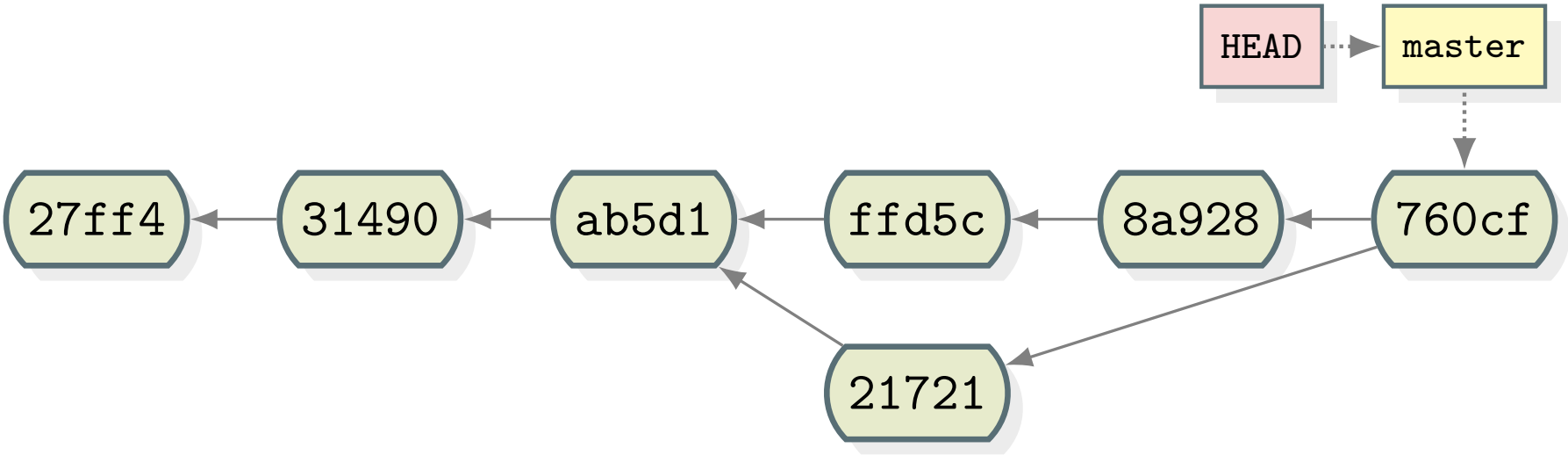
Le Graphe : Merge `legumes`⇒`master`



(a) Merger `légumes` dans `master` : pas de nouveau commit

```
git checkout master
git diff legumes
git merge legumes
git branch -d legumes
```

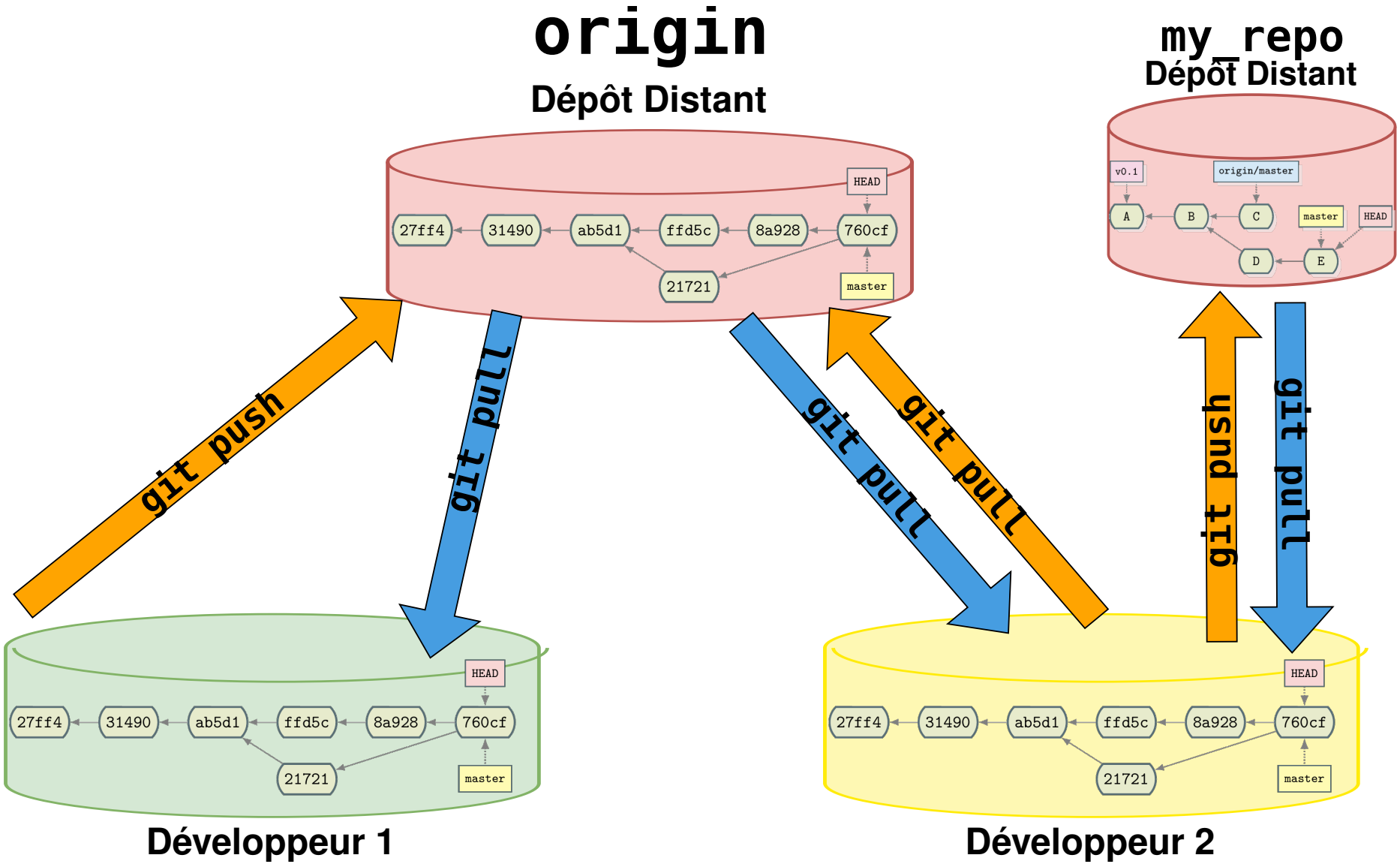
Le Graphe : Merge `legumes`⇒`master`



(a) Effacer la branche `légumes`

```
git checkout master
git diff legumes
git merge legumes
git branch -d legumes
```

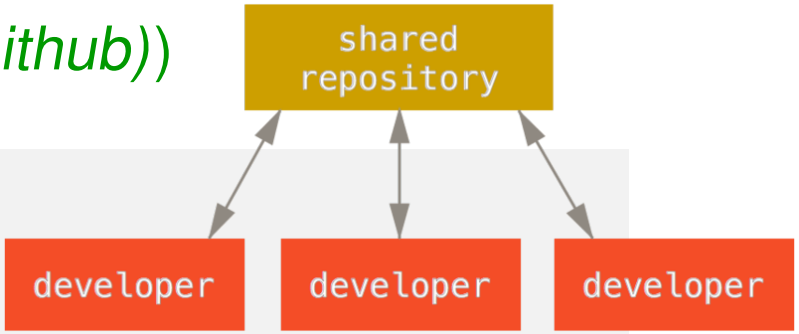
Partager : dépôts distants



Dépôt Centralisée : *initialisation*

Créer le dépôt

(*le dépôt distant doit exister (ici c'est chez Github)*)

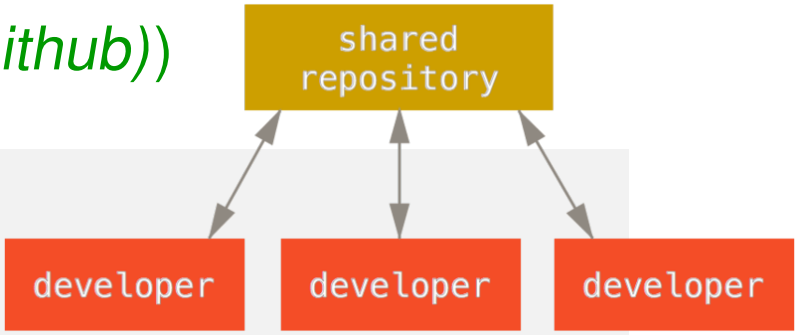



```
1 git init .
2 git add .
3 git commit -m "first commit"
4
5 git remote add origin
  ↪  git@github.com:rudametw/Learning-Git-Test-Repo.git
6 git push -u origin master
```

Dépôt Centralisée : *initialisation*

Créer le dépôt

(*le dépôt distant doit exister (ici c'est chez Github)*)




```
1 git init .
2 git add .
3 git commit -m "first commit"
4
5 git remote add origin
  ↪  git@github.com:rudametw/Learning-Git-Test-Repo.git
6 git push -u origin master
```

Chaque développeur `clone` une seule fois

```
1 git clone https://github.com/rudametw/Learning-Git-Test-Repo.gi
2 cd Learning-Git-Test-Repo/
3 git remote -v #permet de vérifier les addresses
```

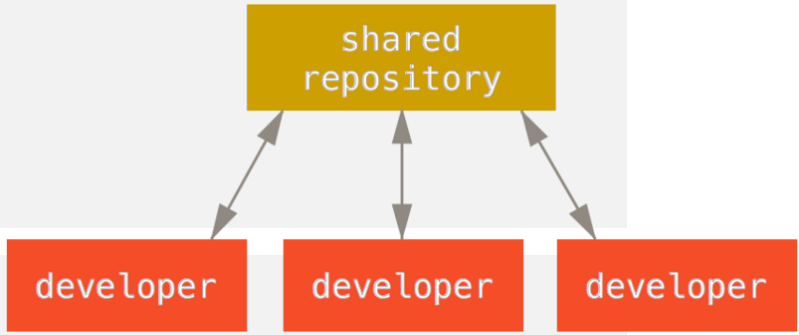
Dépôt Centralisée : *méthode de travail* ***idéal***

Chacun et chaque fonctionalité sur sa branche.
Une fois la fonctionnalité fini, on merge dans `master`.

```
git pull  //update & check work
git branch fonctionalitéX
git checkout fonctionalitéX
```

while (je travaille = vrai)

```
    git diff
    git add <fichiers>
    git commit -m "message"
```




```
git pull --all
git merge master //gérér conflits et TESTER !

//intégrer votre travail
git checkout master
git merge fonctionalitéX
git pull ; git push
```

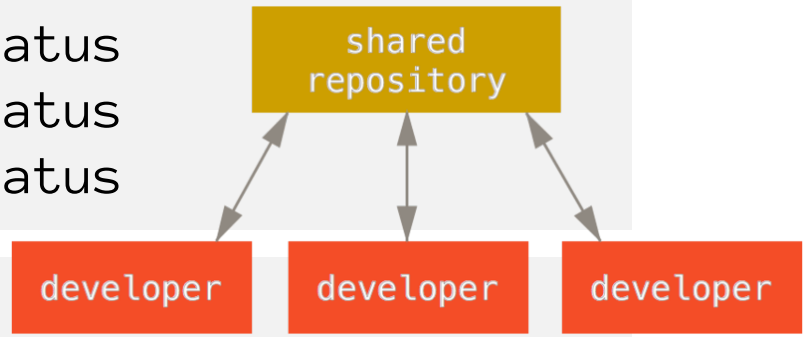
Dépôt Centralisée : *méthode de travail* ***idéal***

En pratique, vérifier l'état de votre dépôt coooonstaaaaament !!!

```
git status ; git pull ; git status //update & check work
git branch fonctionalitéX
git checkout fonctionalitéX
```

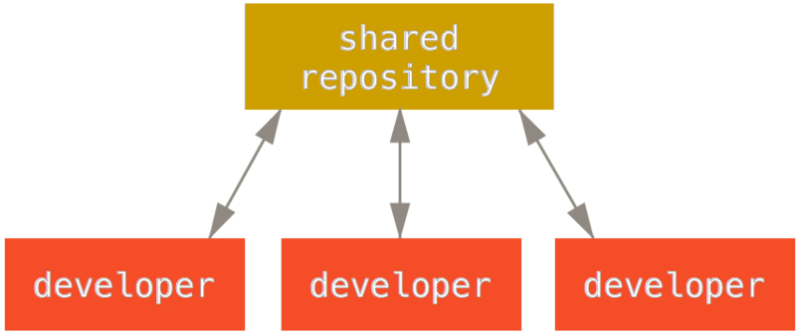
while (je travaille = vrai)

```
    git diff                  ; git status
    git add <fichiers>        ; git status
    git commit -m "message"   ; git status
```




```
git pull --all          ; git status
git merge master //gérér conflits et TESTER !
git status
//intégrer votre travail
git checkout master       ; git status
git merge fonctionalitéX  ; git status
git pull ; git push       ; git status
```

Dépôt Centralisée : *méthode de travail* ***simple***



Sans branches. Commitez souvent.

```
git status
git pull  //mise à jour du dépôt local
git add <fichiers>
git commit -m "message"
git pull --all
git status
git push   //mise à jour du dépôt distant
git status
```

Résolution de conflits

Des conflits vont se produire . . .

. . . comment faire pour les résoudre ?

Provoquer un conflit dans `fruits.txt`

Branche ananas

```
git checkout master
git branch ananas
git checkout ananas
awk 'NR==3\{print "ananas"\}1'
 ↪  fruits.txt > fruits.txt
git add fruits.txt
git commit -m "+ananas"
```

Branche kaki

```
1 git checkout master
2 git branch kaki
3 git checkout kaki
4 awk 'NR==3\{print kaki\}1'
   ↪  fruits.txt | grep -v
   ↪  orange > fruits.txt
5 git add fruits.txt
6 git commit -m "+kaki -orange"
```

Provoquer un conflit dans `fruits.txt`

Branche ananas

```
git checkout master
git branch ananas
git checkout ananas
awk 'NR==3\{print "ananas"\}1'
 ↪  fruits.txt > fruits.txt
git add fruits.txt
git commit -m "+ananas"
```

Branche kaki

```
1 git checkout master
2 git branch kaki
3 git checkout kaki
4 awk 'NR==3\{print kaki\}1'
   ↪  fruits.txt | grep -v
   ↪  orange > fruits.txt
5 git add fruits.txt
6 git commit -m "+kaki -orange"
```

Branche ananas

`fruits.txt` :

```
1 pomme
2 banane
3 ananas
4 orange
5 poire
```

Branche kaki

`fruits.txt` :

```
1 pomme
2 banane
3 kaki
4 poire
```

Merger un conflit dans `fruits.txt`

Branche `ananas`

`fruits.txt` :

```
1 pomme
2 banane
3 ananas
4 orange
5 poire
```

Branche `kaki`

`fruits.txt` :

```
1 pomme
2 banane
3 kaki
4 poire
```

Merger un conflit dans `fruits.txt`

Branche `ananas`

`fruits.txt` :

```
1 pomme
2 banane
3 ananas
4 orange
5 poire
```

Branche `kaki`

`fruits.txt` :

```
1 pomme
2 banane
3 kaki
4 poire
```

Les merges

```
1 git checkout master
2 git merge ananas
```

Sorties console

```
Updating 760cf0e..1711864
Fast-forward
fruits.txt | 1 +
1 file changed, 1 insertion(+)
```

```
3 git merge kaki
```

```
Auto-merging fruits.txt
CONFLICT (content): Merge conflict in fruits.txt
Automatic merge failed; fix conflicts and then
↪  commit the result.
```

`diff` entre `ananas` et `kaki` avant de merger

```
wrudamet@beaner[merge_fruits L|✔] ~/COURS/Git/mon_depot $ git diff 1711864 34dabb6
diff --git a/fruits.txt b/fruits.txt
index e3922ba..5dbddd0 100644
--- a/fruits.txt
+++ b/fruits.txt
@@ -1,5 +1,4 @@
 pomme
 banane
-ananas
-orange
+kaki
 poire
```

Différences entre les *commits* réalisés sur les branches `kaki` et `ananas` qui avaient pour objectif de produire un conflit. En rouge, les lignes qui existent sur la branche `ananas` et pas `kaki`. En vert les lignes qui éxistent sur la branche `kaki` et pas `ananas`.

Résoudre un conflit dans `fruits.txt`

immédiatement après la commande `git merge kaki`

Conflit dans `fruits.txt`

`git` ajoute des guides pour s'y retrouver

```
1  pomme
2  banane
3  <<<<<<< HEAD
4  ananas
5  orange
6  ||||||| merged common ancestors
7  orange
8  =======
9  kaki
10 >>>>>>>
11 poire
```

Résoudre un conflit dans `fruits.txt`

immédiatement après la commande `git merge kaki`

Conflit dans `fruits.txt`

`git` ajoute des guides pour s'y retrouver

```
1  pomme
2  banane
3  <<<<<<< HEAD
4  ananas
5  orange
6  ||||||| merged common ancestors
7  orange
8  =======
9  kaki
10 >>>>>>>
11 poire
```

Solution *(édité à la main)*

```
1  pomme
2  banane
3  ananas
4  kaki
5  poire
```

Résoudre un conflit dans `fruits.txt`

immédiatement après la commande `git merge kaki`

Conflit dans `fruits.txt`

`git` ajoute des guides pour s'y retrouver

```
1  pomme
2  banane
3  <<<<<<< HEAD
4  ananas
5  orange
6  ||||||| merged common ancestors
7  orange
8  =======
9  kaki
10 >>>>>>>
11 poire
```

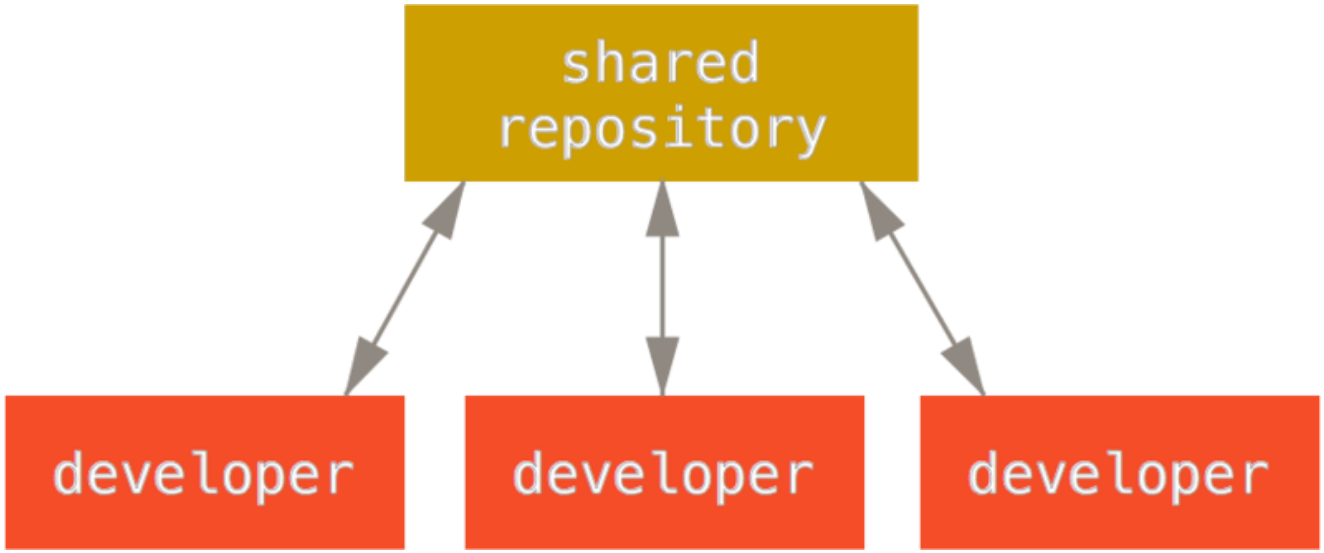
Solution *(édité à la main)*

```
1  pomme
2  banane
3  ananas
4  kaki
5  poire
```

Résolution du conflit (sur terminal)

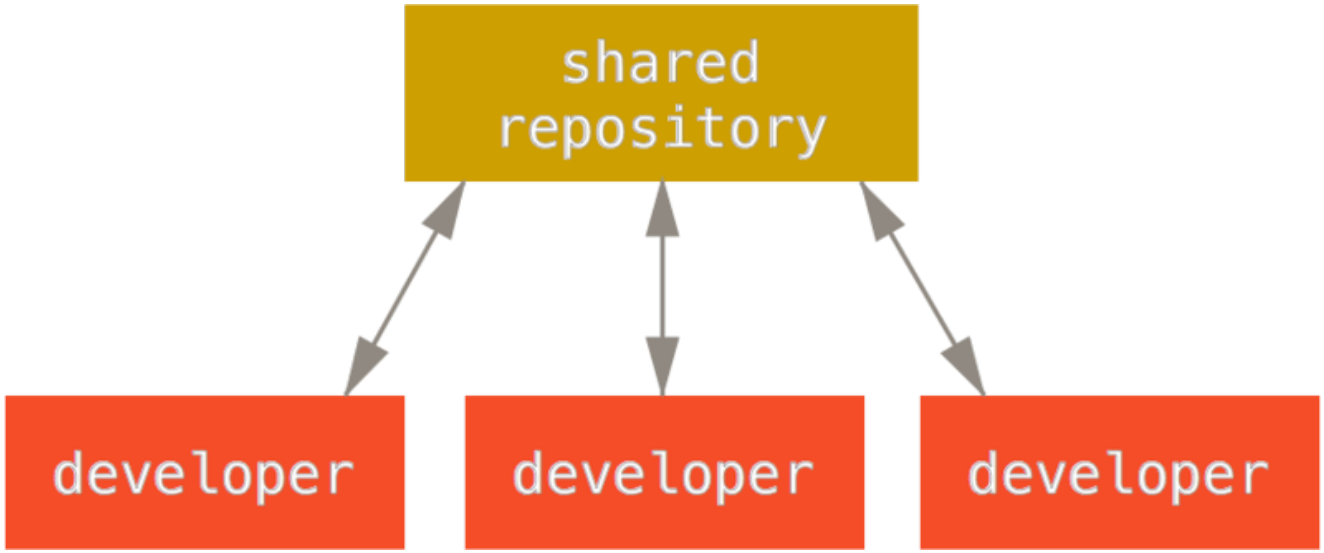
```
1  git add fruits.txt
2  git status
3  git commit -m "Merge branch
   ↪  'kaki' into master"
4  git pull
5  git push
```

Git distribué : Développements distribués

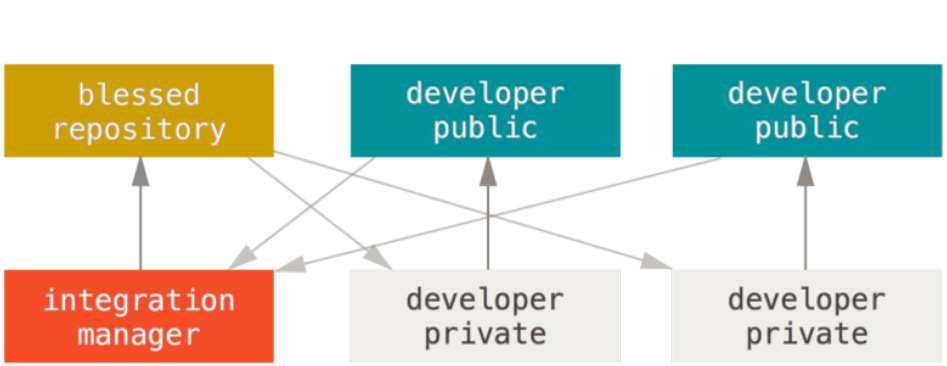


Centralized

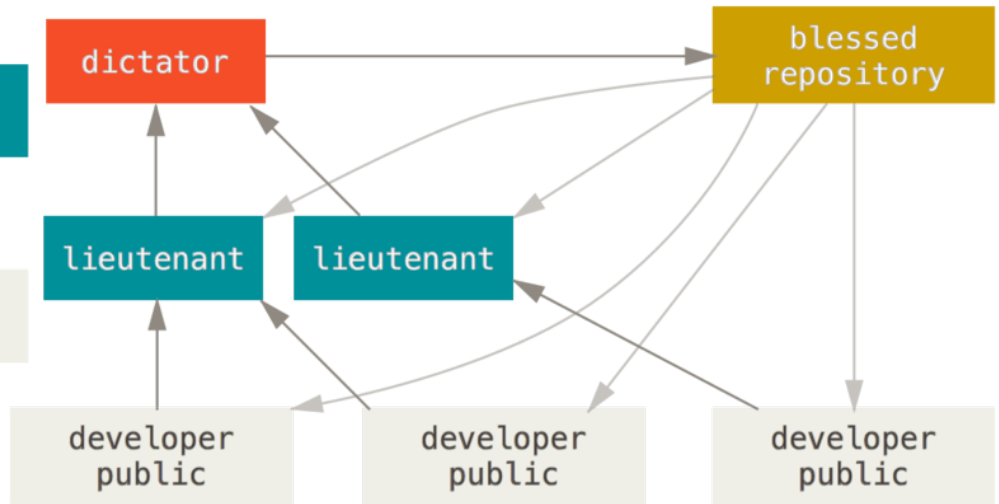
Git distribué : Développements distribués



Centralized



Integration manager



Benevolent Dictator

Premiers pas : *configuration de* `git`

```
git config --global user.name "votre nom"
git config --global user.email nom.prenom@polytech-lille.net
git config --global core.editor 'kate -b' #Par défaut vim
```

- Choix de l'éditeur : `nano`, `vim`, `gedit`, `emacs`, `...`
- À faire **une seule fois** par compte: informations stockées dans `~/.gitconfig`
- Disposez d'un prompt adapté :

  ```
  source ~wrudamet/public/bashrc-students
  ```

  à ajouter dans votre `~/.bashrc`

Quelques astuces (1/4)

Lire, lire et relire la sortie des commandes et les erreurs !!!

Quelques astuces (1/4)

Lire, lire et relire la sortie des commandes et les erreurs !!!

- En cas de doute, vérifiez l'état du dépôt :

```
git status    #Vérifier l'état des fichiers
git status    #Revérifier

ls -lah       #Lister les fichiers du dossier
git remote -v #Lister les dépôts distants
git log       #Regarder vos commits

git status    #Revérifier l'état !
git status    #Re-revérifier
```

Quelques astuces (2/4)

- Afficher un joli log avec graphe et branches
  `git log --graph --oneline --decorate --all`
- Annuler un `merge` en cas de conflit
  `git merge --abort`
- Corriger le dernier commit (avant un push!)
  `git commit --amend`
- Annuler une modification (avant de commiter)
  `git checkout -- <nom_du_fichier>`
- Sauvegarder votre mot de passe (accès `https`, 1h)

```
git config --global credential.helper cache --timeout=3600
```

- Éditer manuellement votre configuration ou créer des alias dans `~/.gitconfig`
- Ne pas mettre un dépôt git dans un dépôt git (effacer le dossier `.git` pour détruire un dépôt)

Quelques astuces (3/4)

- Modifier vos dépôts distants

```
git remote -v #lister tous les dépôts distants
git remote remove <nom_depot> #Effacer un dépôt distant
git remote add    <nom_depot> #Ajouter un nouveau dépôt
git remote rename <vieux_nom> <nouveau_nom> #Renommer
```

Quelques astuces (3/4)

- Modifier vos dépôts distants

```
git remote -v #lister tous les dépôts distants
git remote remove <nom_depot> #Effacer un dépôt distant
git remote add    <nom_depot> #Ajouter un nouveau dépôt
git remote rename <vieux_nom> <nouveau_nom> #Renommer
```

- Par exemple, changement de HTTPS à SSH

```
git clone https://github.com/rudametw/Learning-Git-Test-Repo.git
cd Learning-Git-Test-Repo/
#Mince, je voulais SSH !
git remote -v  #lister les remotes
git remote remove origin
git remote add origin
 ↪  git@github.com:rudametw/Learning-Git-Test-Repo.git
git remote -v  #vérifier le bon changement
```

Quelques astuces (3/4)

- Modifier vos dépôts distants

```
git remote -v #lister tous les dépôts distants
git remote remove <nom_depot> #Effacer un dépôt distant
git remote add    <nom_depot> #Ajouter un nouveau dépôt
git remote rename <vieux_nom> <nouveau_nom> #Renommer
```

- Par exemple, changement de HTTPS à SSH

```
git clone https://github.com/rudametw/Learning-Git-Test-Repo.git
cd Learning-Git-Test-Repo/
#Mince, je voulais SSH !
git remote -v  #lister les remotes
git remote remove origin
git remote add origin
 ↪  git@github.com:rudametw/Learning-Git-Test-Repo.git
git remote -v  #vérifier le bon changement
#Indiquer la branche local↔distant par défaut
git branch -set-upstream-to=origin/master
git pull
```

Quelques astuces (4/4)

- Ne pas commiter des fichiers générés, créer le fichier `.gitignore` à la racine du projet

```
#Exemple de .gitignore
*~
*.o
a.out
build/
bin/
```

- Écrire de la documentation en Markdown
  - Syntaxe simple, propre, comme Wikipédia
  - `README.md` automatiquement converti en HTML
  - Permet de créer tous types de document, très puissant si combiné avec `pandoc`
  - Inspirez vous de `https://gist.github.com/PurpleBooth/109311bb0361f32d87a2`

Conclusion

- Ce cours est une introduction de `git`
- Gestionnaire de versions, élément incontournable du développeur ou équipe de développeurs
- `git` : outil performant et massivement utilisé
- `git` : spécialisé pour le texte et la ligne de commande mais de nombreuses extensions et outils graphiques
  - `gitk`, `smartgit`, tortoise (windows), EGit pour environnement Eclipse, ...

Liens, aides et outils (1/2)

- References bibliographiques
  - Livre “Pro-Git” De Scott Chacon and Ben Straub `https://git-scm.com/book/fr/v2`
  - Git Magic (Stanford) `https://crypto.stanford.edu/~blynn/gitmagic/intl/fr/book.pdf`
  - Présentation “Les bases de GIT” `https://fr.slideshare.net/PierreSudron/diapo-git`

- Où stocker vos projets
  - `https://gitlab.univ-lille.fr/`
  - ~~`https://archives.plil.fr/`~~ $\Leftarrow$ Polytech
  - `https://gitlab.com/`
  - `https://github.com/`
  - `https://bitbucket.org/`
  - Votre serveur perso (e.g., gitea, gitlab)

Liens, aides et outils (2/2)

- Tutoriels
  - http://www.cristal.univ-lille.fr/TPGIT/
  - https://learngitbranching.js.org/
  - https://try.github.io/
  - https://www.mixiumum.fr/blog/enfin-comprendre-git/
- Vidéos
  - https://www.youtube.com/watch?v=OqmSzXDrJBk
  - https://www.youtube.com/watch?v=uR6G2v_WsRA
  - https://www.youtube.com/watch?v=3a2x1iJFJWc
  - https://www.youtube.com/watch?v=1ffBJ4sVUb4
  - https://www.youtube.com/watch?v=duqBHik7nRo