Projet PFA

Cours 2

kn@lri.fr

université PARIS-SACLAY

Plan

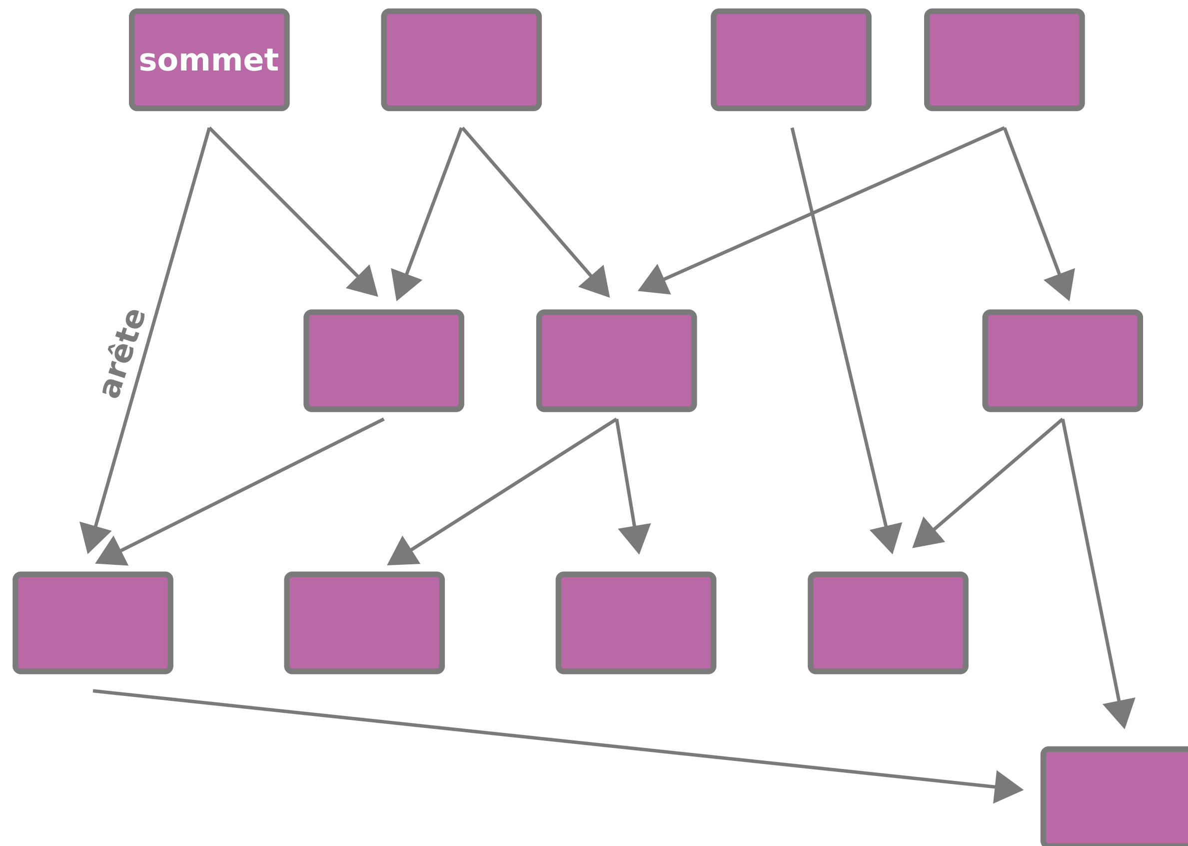
1 Intro Projet ✓

2 Git

2.1 Git

Rappels : DAG

Directed Acyclic Graph : graphe orienté acyclique.



git

`git` est un système de contrôle de version distribué (DVCS) :

- suivre un ensemble de fichiers (ou *projet*)
- suivre les modifications faites à ces fichiers
- savoir qui a modifié quoi et quand et pourquoi
- tout le monde possède tout l'historique (pas de notion de dépôt central ou serveur)

Utilité :

- Pouvoir revenir en arrière dans l'historique d'un projet
- Pouvoir partager ses modifications avec d'autre personnes
- Importer les modifications de quelqu'un et gérer les conflits (modifications incompatibles)
- Travailler de manière asynchrone possiblement sans réseau

Historique

- pré-2005 : Le noyau Linux utilise l'outil propriétaire `BitKeeper` pour gérer les sources du noyau (+22000 fichiers, 12 million de lignes à l'époque)
- Plusieurs développeurs refusent d'utiliser un outil non libre
- 2005 Linus Torvalds décide d'écrire son propre système de contrôle de version avec les critères suivants :
  - Décentralisé/Distribué : pas de concept de dépôt central, possibilité de travailler localement sur sa machine
  - Efficace
  - Robuste (protection contre la corruption de fichier, ...)

`git` est crée en 15 jours. L'architecture interne a peu changé, par contre l'interface utilisateur a évolué.

`git` par l'exemple

- On va explorer au fur et à mesure les différents concepts de `git`
- On va aussi voire toutes les commandes liées à ces concepts et leur utilisation

D'autres références :

- https://www.youtube.com/watch?v=4XpnKHJAok8 talk de Torvalds sur git en 2007
- https://git-scm.com/ : le site de `git`, avec sa documentation
- https://github.com, https://bitbucket.org/, https://about.gitlab.com : des hébergeurs de code basés sur git (propriétaires)
- https://gitlab.u-psud.fr : un hebergement proposé par Paris-Sud (login Adonis)

Dépôt `git` (ou projet)

Un dépôt est une collection de fichiers se trouvant dans un répertoire et dont les modifications sont suivies par `git`

```
$ mkdir my-project

$ cd my-project

$ git init
Initialized empty Git repository in …/my-project/.git/

$ ls -a
. .. .git/
```

Tout les fichier que l'on veut suivre doivent être dans ce répertoire, ou un sous répertoire

La commande *`git status`* permet de connaître l'état du dépôt

commit (1)

Un *commit* est un instané de tous les fichiers du dépôt et de leur contenu à un instant donné, plus des méta-données (date, nom de la personne qui a crée le commit, ...). Les commits sont crées en deux temps :

1. ajout des fichiers modifiés que l'on souhaite au prochain commit
2. validation du commit

```
$ emacs README.md  #ou n'importe quel autre éditeur de texte
$ git add README.md
$ git status
…
Changes to be committed:
    (use "git rm --cached <file>..." to unstage)

new file:   README.md
$ git commit     #lance l'éditeur par défaut
[master (root-commit) 71dda77] Ajout du premier fichier.
1 file changed, 1 insertion(+)
create mode 100644 README.md
```

commit (2)

`git` ne possède pas de commande spéciale pour ajouter un fichier au dépôt. Ajouter un fichier, ou modifier un fichier existant se font de la même manière

`git` ne peut stocker que des *fichiers* et leur chemin. En particulier, on peut pas ajouter un répertoire vide dans un dépôt `git`.

Un commit est identifié de manière unique par son *hash* (somme de contrôle).

`git` utilise l'algorithme SHA-1 : les hash font 160 bits (20 octets ou 40 chiffres hexadécimaux).

La commande *`git show abcde…`* permet de montrer l'objet git (commit mais aussi d'autres) dont le hash est donné. On n'est pas obligé de donner les 40 caractères, un prefixe suffit.

Historique (1)

`git` garde un historique des modifications faites à un dépôt.

Chaque commit pointe vers son parent et le hash du parent est pris en compte dans le calcul du hash du commit.
⇒ Impossible de falsifier un « commit » dans l'historique

```
$ emacs README.md
$ emacs test.ml
$ git add test.ml README.md
$ git commit
[master 30d4a5d] Ajout du fichier principal.
2 files changed, 2 insertions(+), 1 deletion(-)
create mode 100644 test.ml
$ git log   #affiche les commits
```

Historique (2)

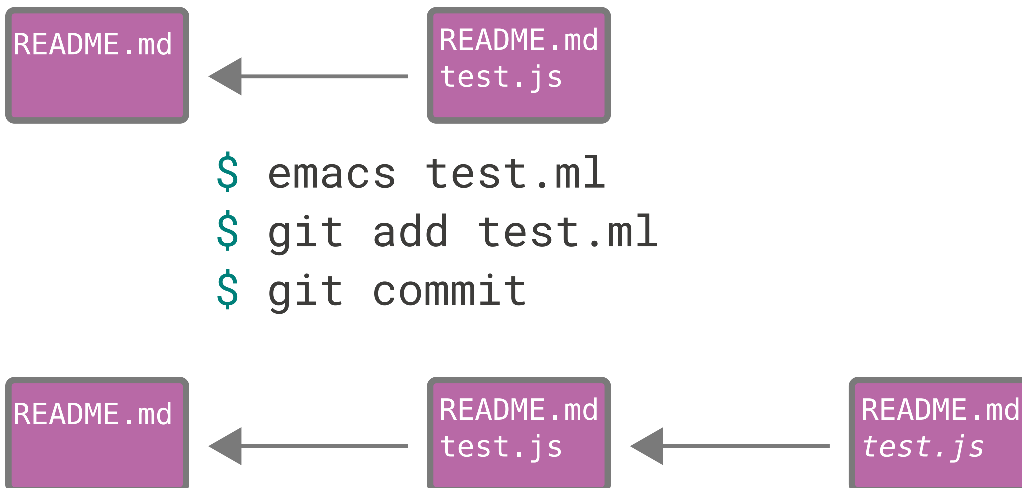
Un commit `git` n'est pas un `diff` d'une version du fichier à l'autre. C'est l'ensemble des fichiers tels qu'ils étaient à un moment donné.

`git` stocke ces « images instantanées » de manière compacte.

Historique (3)

L'ensemble des commit d'un projet git forme un DAG :

- Il y a un commit initial
- Chaque commit pointe vers son (ou ses) parent(s)
- Un chemin entre un commit et le commit initial est appelé une branche (`branch`)



Branches (1)

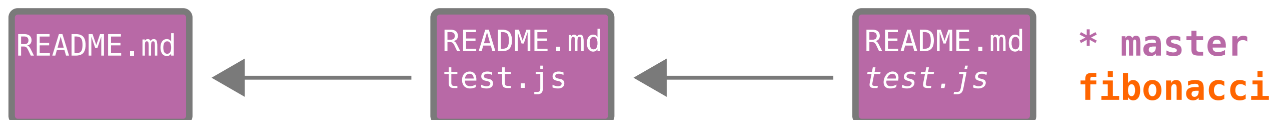
Sauf cas particulier, on est toujours sur une branche, i.e. sur un chemin nommé entre le commit courant et le commit initial

La branche principale, crée par défaut se nomme `master`

```
$ git branch
* master
```

Il est possible d'avoir plusieurs branches (pour faire des essais, corriger des bugs, …) sans toucher à la branche principale.
La commande `git branch foo` crée une nouvelle branche appelée `foo` à partir du commit courant.

```
$ git branch fibonacci
$ git branch
fibonacci
* master
```

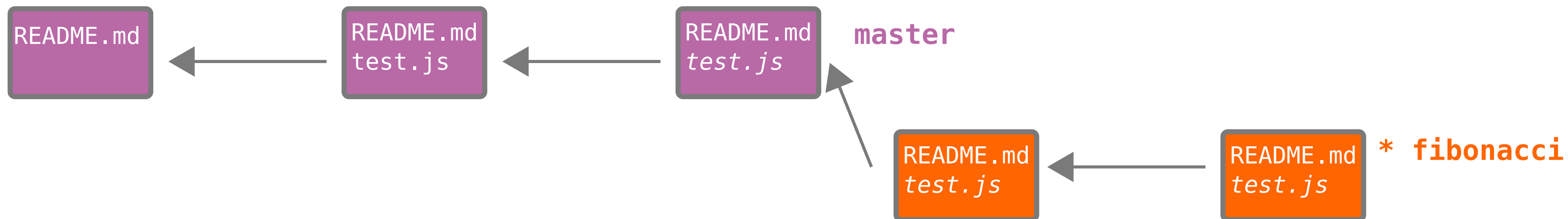


Branches (2)

Chaque branche se souvient de son dernier commit

On peut changer de branche avec `git checkout ma_branche`

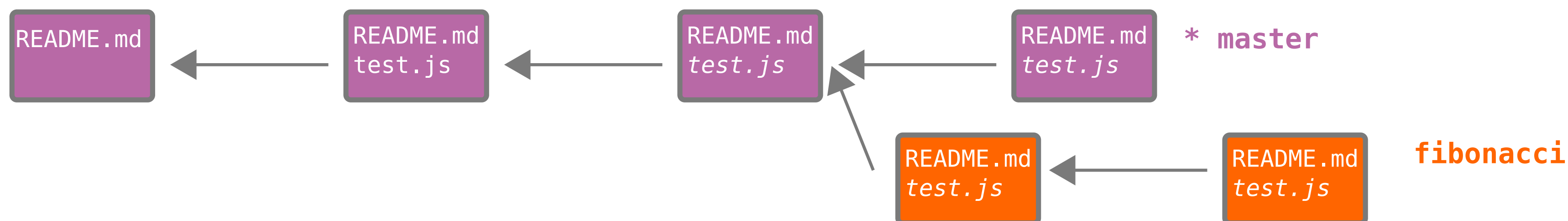
```
$ git checkout fibonacci
Switched to branch 'fibonacci'
$ emacs test.ml
$ git add test.ml
$ git commit
$ emacs README.md
$ git add README.md
$ git commit
```



Branches (3)

Deux branches sont complètement indépendantes et peuvent évoluer en parallèle:

```
$ git checkout master
Switched to branch 'master'
$ emacs README.md
$ git add README.md
$ git commit
```



merges

Une opération de *merge* (fusion) consiste à importer les modifications d'une branche dans une autre. Pour cela :

1. On se place dans la branche de destination (avec `git checkout ma_branche`)
2. On effectue `git merge branche_source`

```
$ git checkout master
Switched to branch 'master'
$ git merge fibonacci
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
```

`git` essaye de fusionner les deux branches automatiquement. La plupart du temps il y arrive. Si un même fichier est modifié de deux manière différentes, il y a un conflit.

Conflits

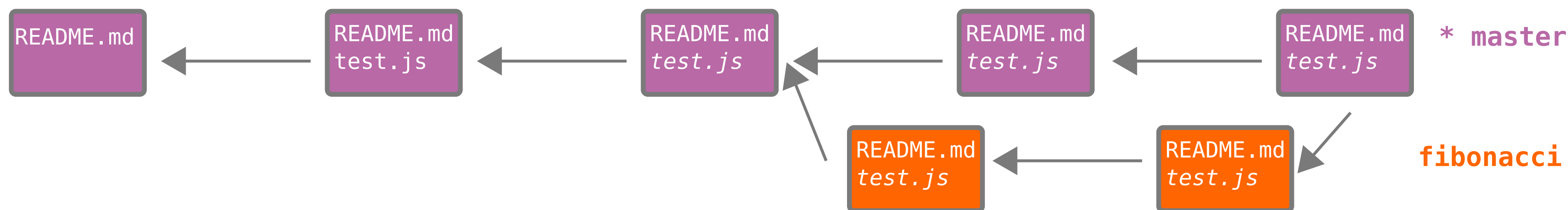
Un conflit dans un fichier est simplement matérialisé par une section :

```
<<<<<<< HEAD
du texte
=======
une autre version
>>>>>>> branche_source
```

La portion du haut est le texte de la branche cible (sur laquelle on est) et celui du bas celui de la branche source.

Il faut retirer toutes les sections qui ont cette forme (en choisissant l'une ou l'autre des versions, ou encore une troisème), puis commiter.

```
$ emacs README.md #on règle le conflit
$ git add README.md
$ git commit
[master e01a960] Merge branch 'fibonacci'
```



Dépôt distant (1)

Pour récupérer un projet git déjà existant, on utilise la commande `git clone` :

```
$ git clone https://gitlri.lri.fr/kim.nguyen/js
```

De telles URL sont appelées « *remote* »
L'URL source initiale est appelée `origin`.

Pour synchroniser le projet local avec un *remote* on utilise `git pull` et `git push`.

`git pull [nom-de-remote-ou-url]` : récupère l'état de la même branche dans la repository distante et la merge dans la branche courante

`git push [nom-de-remote-ou-url]` : merge la branche courante dans la branche de même nom de la repository distante

Dépôt distant (2)

En réalité, un dépôt git (local) possède :

- toutes les branches définie localement
- Une branche pour chaque branche de chaque remote

```
$ git branch -a
* master
remotes/origin/HEAD -> origin/master
remotes/origin/fibonacci
remotes/origin/master
```

La commande `git pull` est en fait un alias pour :

1. `git fetch` : synchronize la branche mirroir locale de chaque remote
2. `git merge` de la branche mirroir locale dans la branche locale

Dépôt distant (3)

En fonction de l'état des dépôt distants, les commandes `git pull` ou `git push` peuvent créer des conflits :

- `git pull` : c'est un simple conflit de merge. On le résout en retirant les `<<…==…>>` dans les fichiers concernés et on commit
- `git push` : le conflit serait stocké « coté remote », la commande échoue. Il faut donc faire un `git pull` (qui va merger le nouveau remote, on peut alors régler les conflits localement) puis `git push`.

Plein d'autres outils ...

`git log` : affiche l'historique

`git log --graph` : affiche l'historique avec le graphe de commits

`git diff` : Affiche la différence entre deux commits donnés

`git bisect` : Permet de trouver une régression introduite par un commit

...

Fichier à ne pas mettre sous git

Certains fichiers peuvent être présents mais ne doivent pas être versionnés :

- Les fichiers générés : `.o`, `.class`, ...
- Les fichiers contenant des informations confidentielles (mots de passe, ...)

On peut créer dans tous lse sous-répertoire du dépôt (et aussi à la racine) un fichier `.gitignore` qui contient les fichiers à ignorer. Git ne se plaindra pas de leur existance et interdira de les commiter par erreur.

`github` et compagnie ?

Il existe de nombreux services autour de `git`. Ils ne sont pas nécessaire à l'utilisation de `git` pour un projet, mais offre des avantages :

- Site web pour le projet
- Système de gestion de bugs (bugtracker)
- Système d'intégration continue (les tests sont rejoués après chaque `git push`, si un test échoue, le `push` est refusé
- Génération d'archives zip à partir du dépôt

On fera cepdant garde, les remarques usuelles sur la souveraineté des données s'appliquent.