Systèmes de contrôle de version & git

Master-I parcours SSD

Pierre Mahé - bioMérieux & Université de Grenoble-Alpes

Introduction

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.

Système de gestion de version :

- un logiciel permettant de simplifier ce processus
- un élément clé de l'arsenal du data-scientist

Ce cours : une introduction à un système de gestion de version classique, `git`.

1. https://fr.wikipedia.org/wiki/Gestion_de_versions

Plan

1. Systèmes de contrôle de version
2. Systèmes décentralisés : l'exemple de `git`
3. Remarques et conclusions

⇒ TP : prise en main de `git`

Sources principales

- Chacon and Straub (2014) : Pro git : Everything you need to know about Git
- Blischak et al. (2016) : a quick introduction to version control with `git` and `github`

- `svn` quick guide : `http://www.tutorialspoint.com/svn/svn_quick_guide.htm`
- Noble (2009) : a quick guide to organizing computational biology projects

I - Systèmes de contrôle de version

Gestion de version

Wikipedia[2] : 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.

2. https://fr.wikipedia.org/wiki/Gestion_de_versions

Gestion de version

Wikipedia[2] : 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.

La solution archaïque : dupliquer les fichiers

- version 1, version 2, ...
- 2018-11-03, 2018-11-04, ...

2. https://fr.wikipedia.org/wiki/Gestion_de_versions

Gestion de version

Wikipedia[2] : 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.

La solution archaïque : dupliquer les fichiers

- version 1, version 2, ...
- 2018-11-03, 2018-11-04, ...

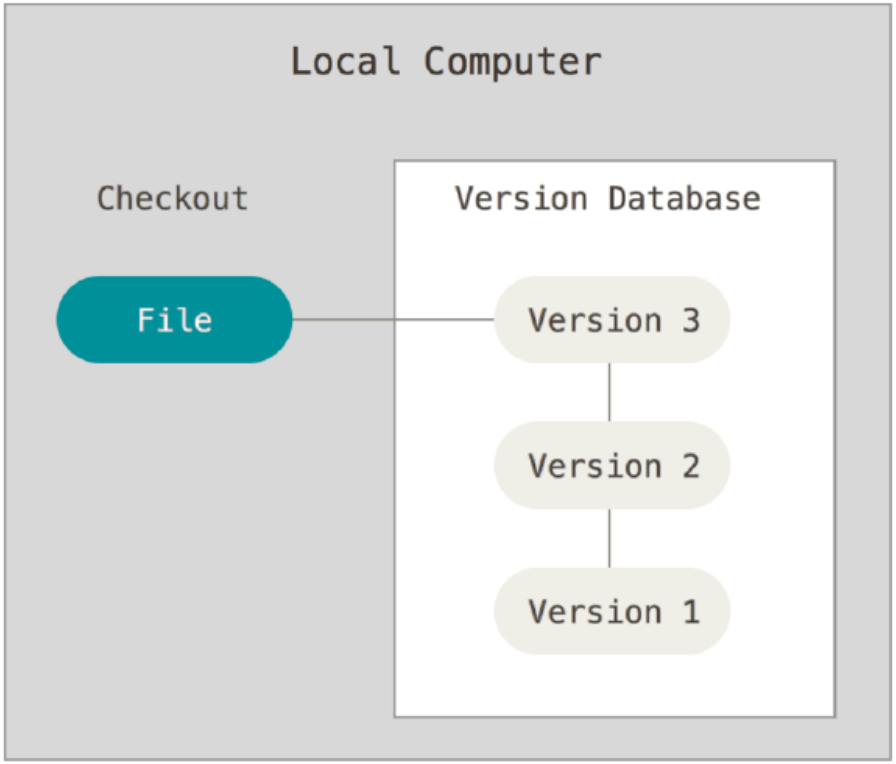
Les systèmes de gestion de version visent à simplifier (et améliorer !) ce processus :

- sauvegarde automatisée, gestion incrémentale
- accès à l'historique et possibilité de restoration.

2. https://fr.wikipedia.org/wiki/Gestion_de_versions

Systèmes de gestion de version - 3 types

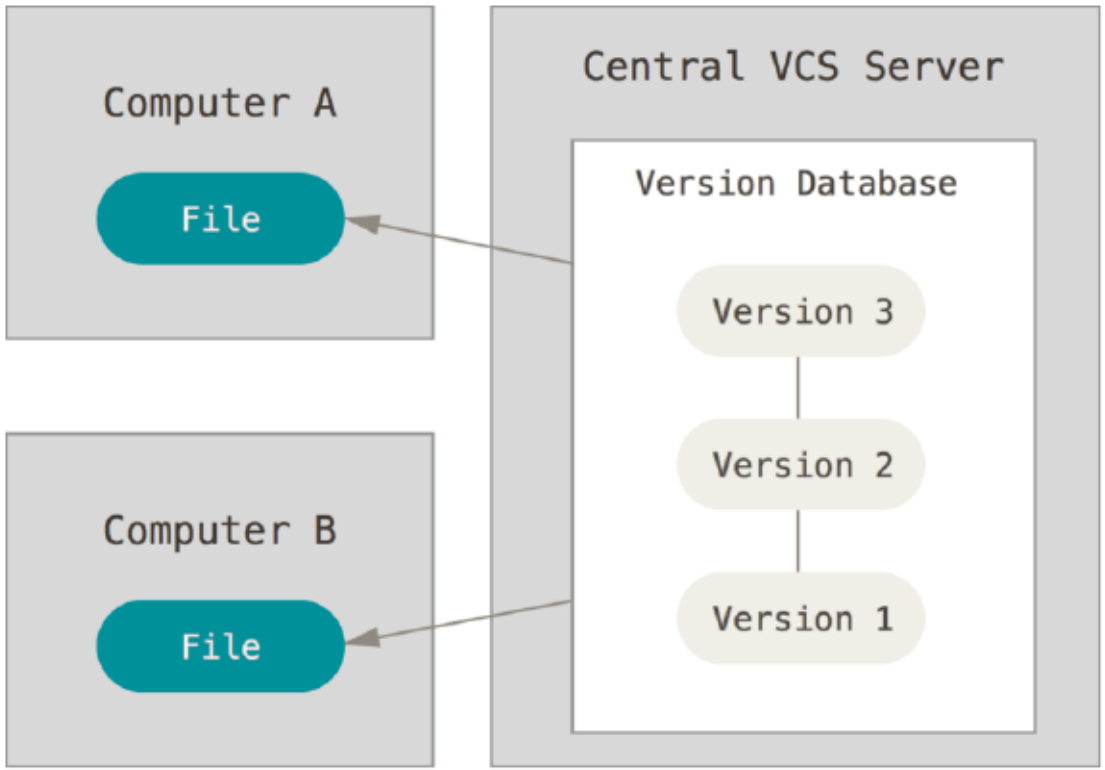
Système local :



- usage personnel
- pas très commun...

Systèmes de gestion de version - 3 types

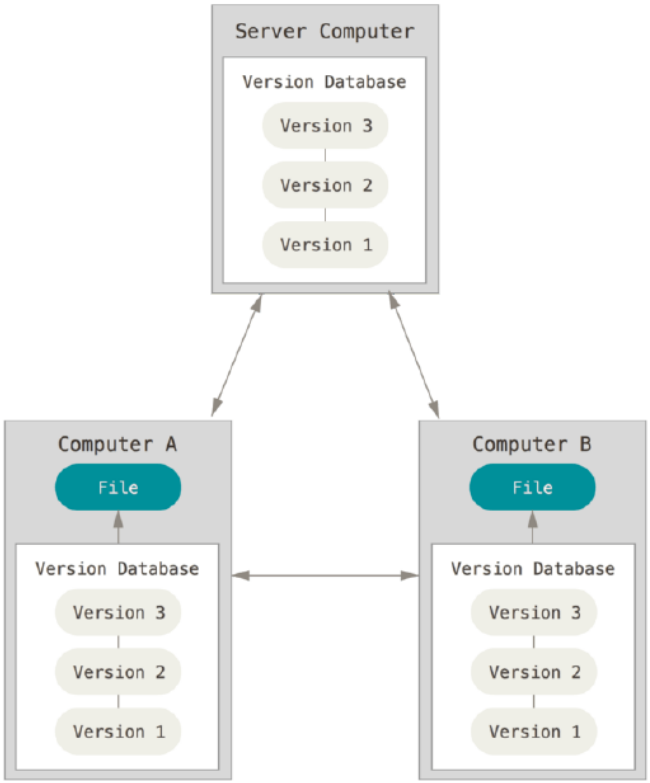
Système centralisé :



- usage collaboratif et/ou individuel sur plusieurs sites
- mode client / serveur
- exemples : `cvs`, `subversion`

Systèmes de gestion de version - 3 types

Système décentralisé (distributed) :



- usage collaboratif et/ou individuel sur plusieurs sites
- ~ mode client / serveur à deux étages
- exemples : `git` (le nouveau standard)

Systèmes de gestion de version - 3 types

Avantages et inconvénients :

	local	centralisé	distribué
gestion des versions	+	+	+
travail individuel inter-sites	-	+	+
travail collaboratif	-	+	++
sécurité de sauvegarde	-	+/-	++

⇒ travail collaboratif & systèmes distribués :

- + de flexibilité → meilleure gestion de projets complexes
  - plusieurs équipes sur plusieurs sites
  - contribuer à un projet externe (type open-source)

⇒ sécurité de sauvegarde & systèmes centralisés :

- sauvegarde de son ordinateur...mais dépend du serveur

Versionner quoi ?

Utilisation principale : développement logiciel

- historique de l'évolution : "releases"
- travail collaboratif : synchronisation
- développements en parallèle : "branches"

Versionner quoi ?

Utilisation principale : développement logiciel

- historique de l'évolution : "releases"
- travail collaboratif : synchronisation
- développements en parallèle : "branches"

Mais en soi : tout type de document !

- rapport (tex, word ou autres)
- graphismes
- données (si pas trop grosses)

Versionner quoi ?

Utilisation principale : développement logiciel

- historique de l'évolution : "releases"
- travail collaboratif : synchronisation
- développements en parallèle : "branches"

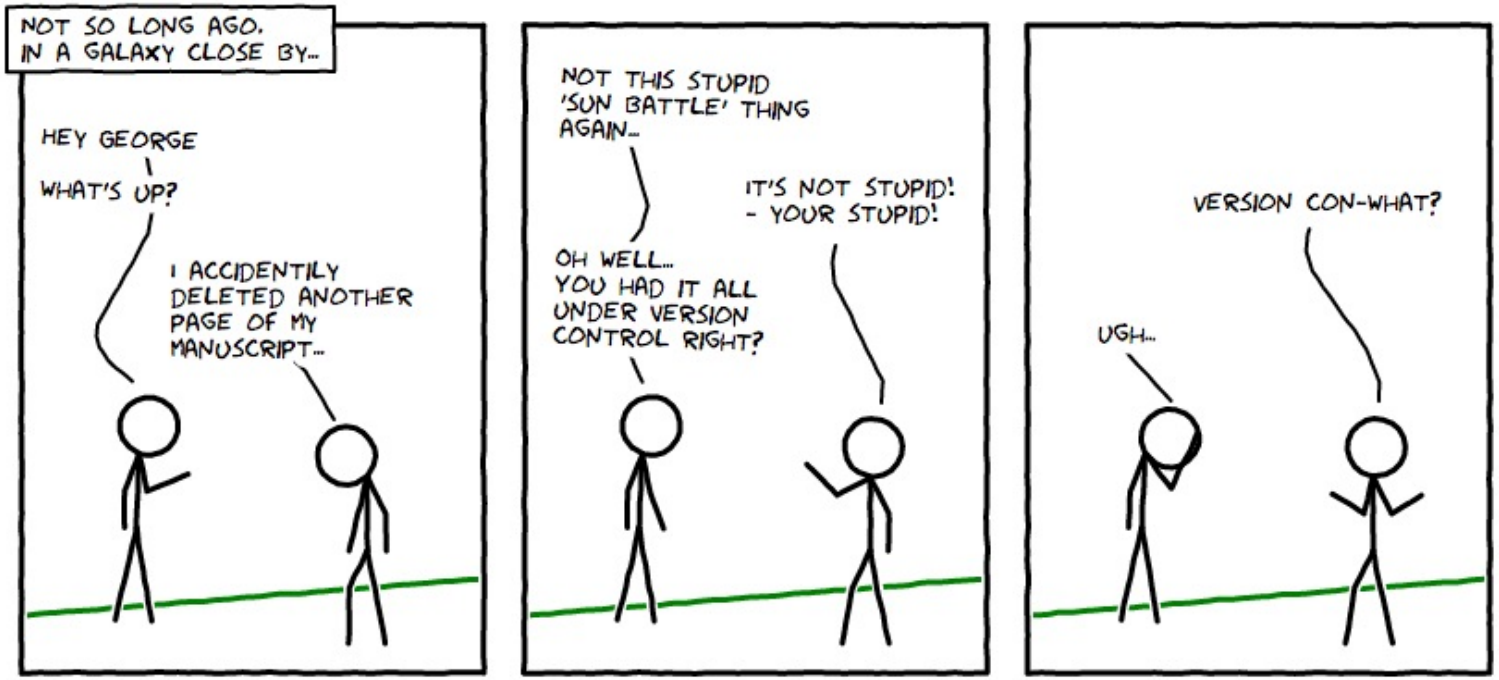
Mais en soi : tout type de document !

- rapport (tex, word ou autres)
- graphismes
- données (si pas trop grosses)

⇒ pour tout fichier "précieux", qui évolue et/ou qui est partagé : **pourquoi ne pas le "versionner" ?**

Versionner quoi ?

Versionner quoi ?

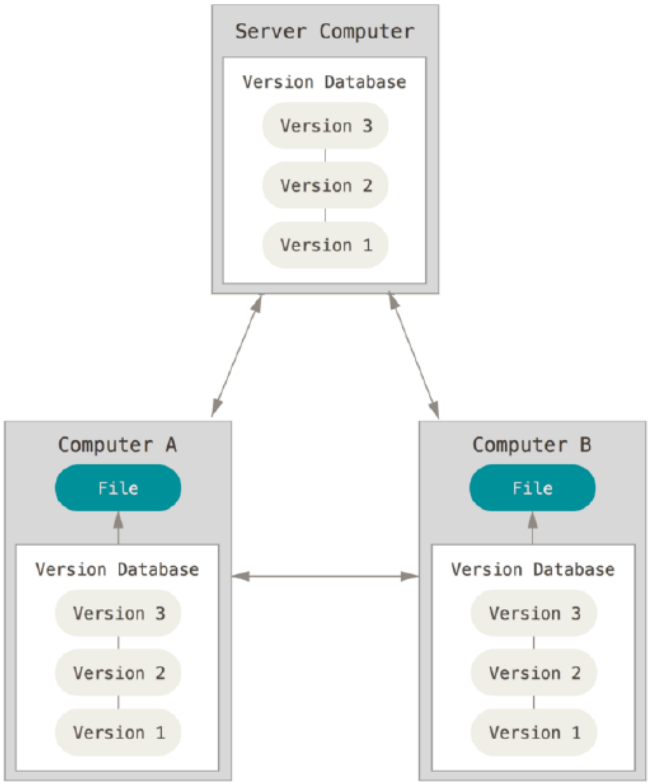


III - Systèmes décentralisés & git

- principes de base
- opérations de base
- travailler un sur serveur distant
- git & branches

git

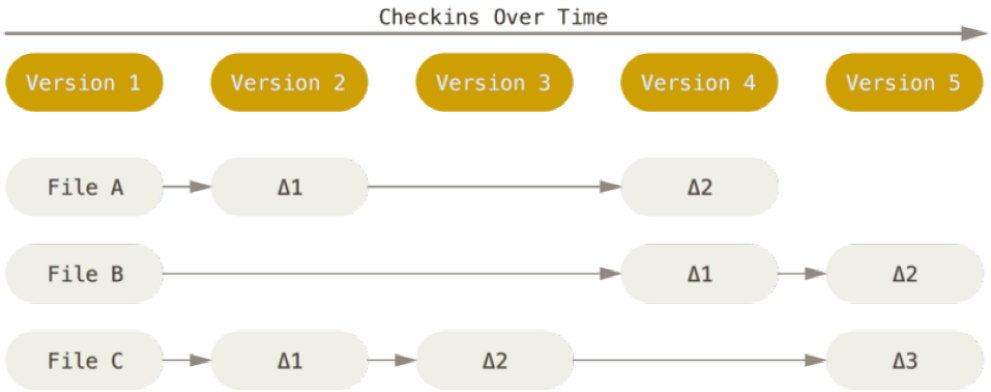
Un système de contrôle de version décentralisé / distribué :



- Créé en 2005 par Linus Torvalds (créateur de Linux).
- Le nouveau standard de la gestion de version.

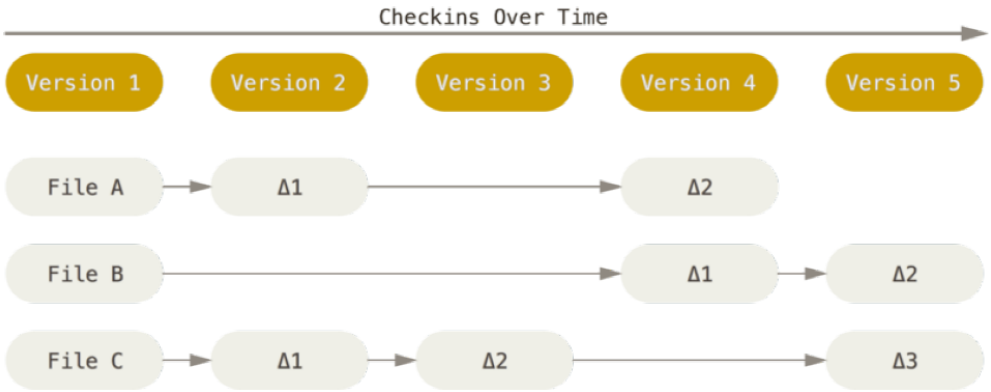
git & "snapshots"

En général (e.g., svn) : 1 version = {différences} ("delta")

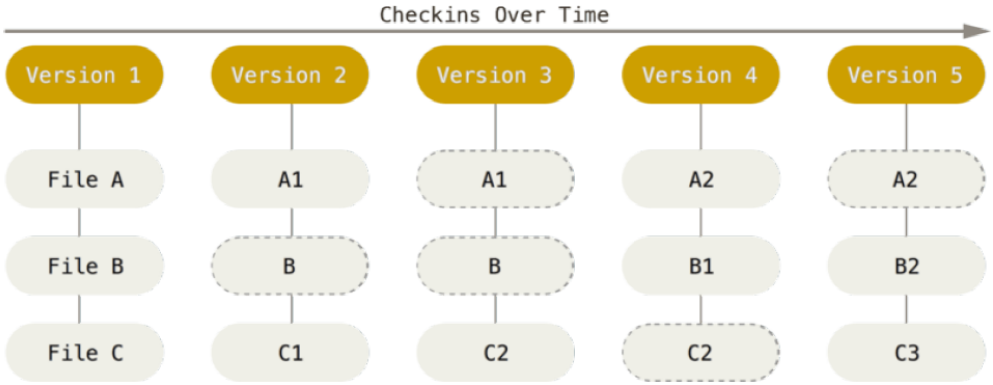


git & "snapshots"

En général (e.g., svn) : 1 version = {différences} ("delta")



Avec git : 1 version = 1 "snapshot" des fichiers



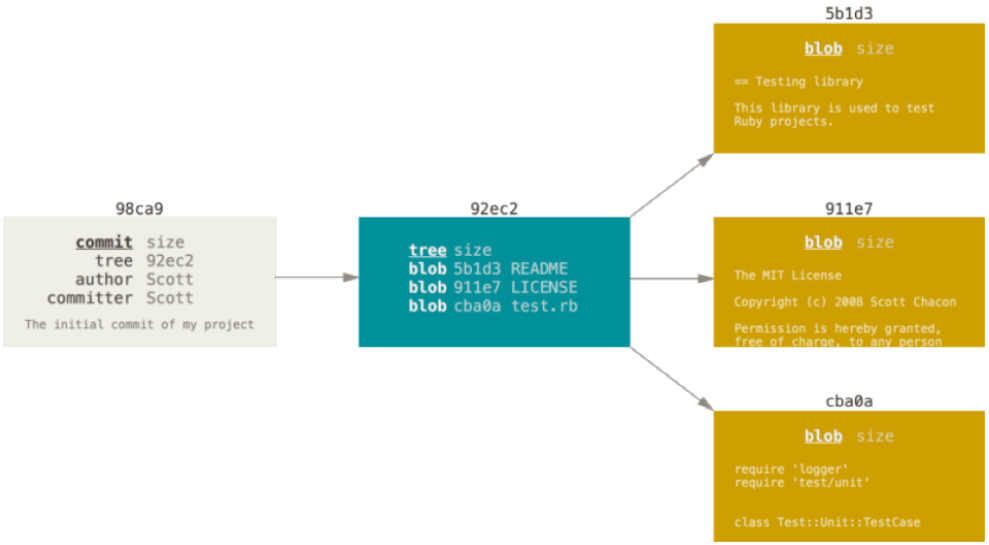
⇒ + rapide : beaucoup d'opérations réalisées en local

⇒ + flexible (systèmes de branches)

Versions & "commits"

Avec `git` : 1 version = 1 "commit"

- un auteur + une date + un message
- un pointeur vers un "snapshot" : "working tree"



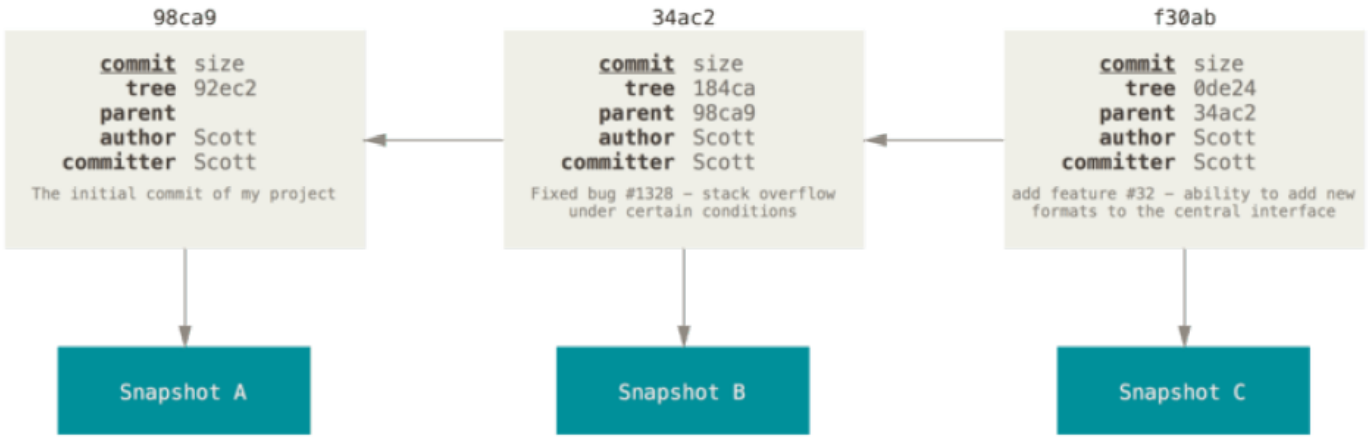
Un "working tree" : des pointeurs vers du contenu ("blobs")

⇒ identifiants = checksums obtenus par fonction de hachage

- garantie d'intégrité

Historique & "commits"

Historique = des pointeurs entre "commits" :

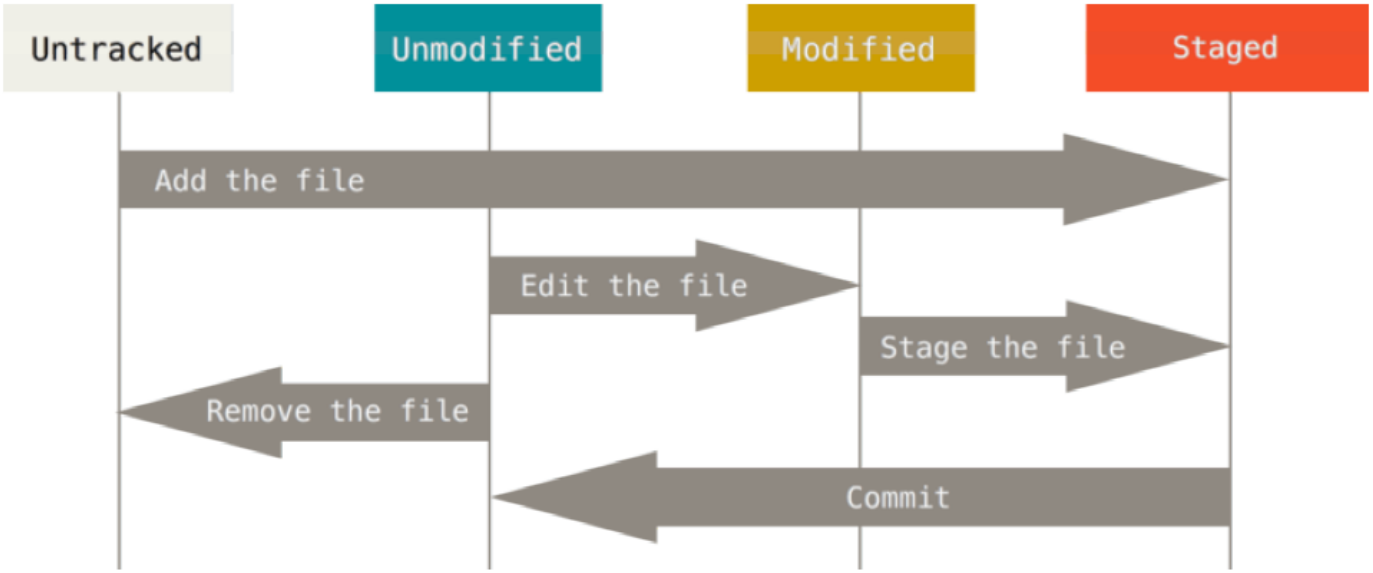


- "commit"initial
- chaque commit pointe vers le commit précédent (parent)

⇒ rend la gestion de "branches" simple et efficace

Staging area

Staging area : ce qui sera pris en compte au prochain commit



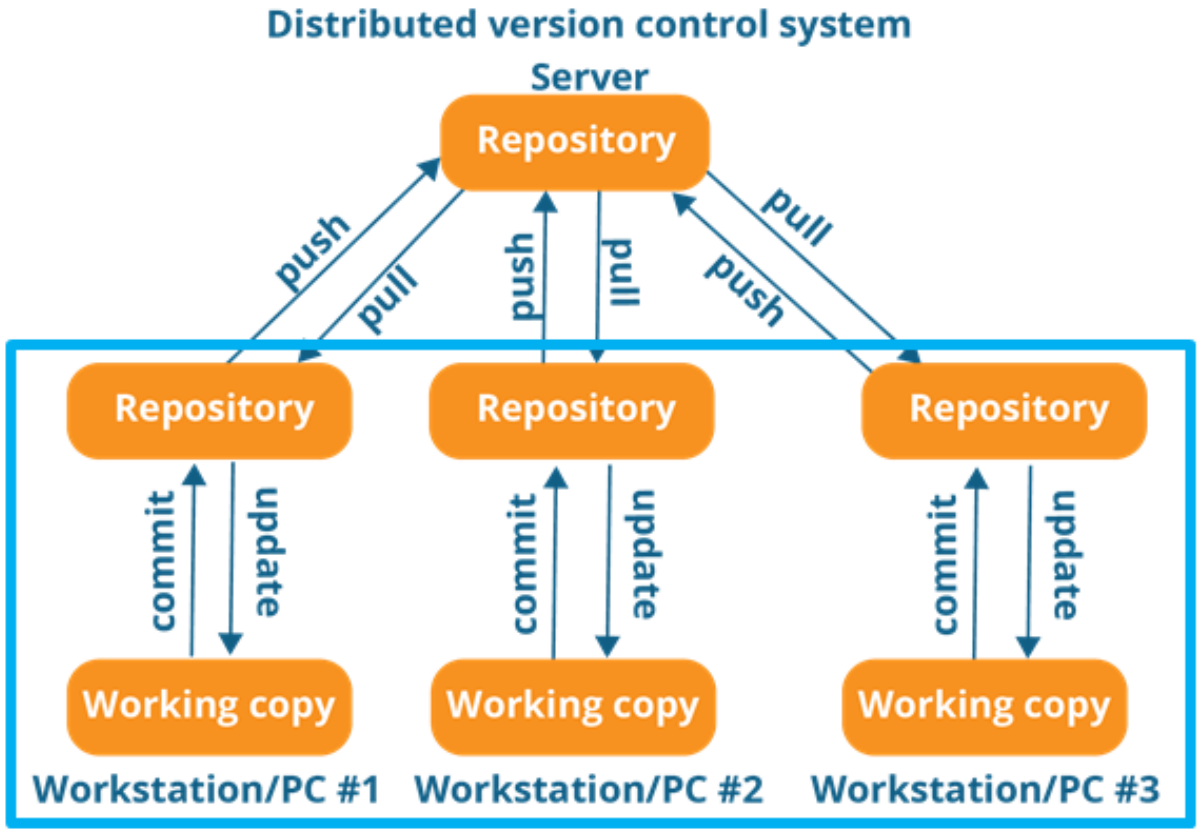
- différence importante par rapport aux systèmes de version classique (e.g., `svn`).
- tracked / untracked : ce qui fait partie du dépôt ou non.

III - Systèmes décentralisés & git

- principes de base
- opérations de base
- travailler un sur serveur distant
- git & branches

Opérations de base ?

"Opérations de base" = ce qui est fait en local :



⇒ ~ "a la `subversion`".

Opérations de base

- créer un projet : `git init`
- ajouter un fichier au projet : `git add`
- prendre en compte la modification d'un fichier : `git add`
  - i.e., l'ajouter à la "staging area"
- faire l'état des lieux : `git status`
- créer une nouvelle version : `git commit`
  - i.e., enregistrer les modifications
- voir les modifications d'un fichier : `git diff`
- voir l'historique des commits : `git log`
- supprimer / déplacer un fichier : `git rm` et `git mv`
- annuler une modification : `git reset` et `git checkout`

⇒ à appeler en ligne de commande.

Créer un projet : `git init`

Pour créer un projet sur votre machine :

1. se rendre dans le répertoire voulu : `$cd my_project`
2. utiliser la commande `git init` : `$git init`

Créer un projet : `git init`

Pour créer un projet sur votre machine :

1. se rendre dans le répertoire voulu : `$cd my_project`
2. utiliser la commande `git init` : `$git init`

⇒ crée un répertoire (caché) nomme `.git`

- dossier interne utilisé par `git`
- pas la peine de savoir précisément ce qu'il contient

⇒ pour l'instant, aucun fichier n'est "tracké"

- même si le répertoire `my_project` n'était pas vide

Ajouter ou modifier un fichier : `git add`

La commande `git add` permet :

1. d'ajouter un fichier au projet : `$git add my_new_file`
   - le fichier devient "tracké"
   - il passe directement dans la "staging area"
2. de prendre en compte une modification apportée à un fichier : `$git add my_old_file`
   - il passe dans la "staging area"

Ajouter ou modifier un fichier : `git add`

La commande `git add` permet :

1. d'ajouter un fichier au projet : `$git add my_new_file`
   - le fichier devient "tracké"
   - il passe directement dans la "staging area"
2. de prendre en compte une modification apportée à un fichier : `$git add my_old_file`
   - il passe dans la "staging area"

⇒ à ce stade, pas de nouvelle version n'est créée.

- `git add` → "staging area"

⇒ cela passera par la commande `git commit`.

Dresser un état des lieux : `git status`

La commande git status permet de dresser un état des lieux :

- fichiers pas encore suivis ("trackés")
  - `"Untracked files"`
- modifications pas encore prises en compte ("stagées")
  - `"Changes not staged for commit"`
- modifications prises en compte
  - `"Changes staged for commit"`
  - fichiers nouveaux ou modifiés

⇒ fournit l'information dans la console
⇒ à utiliser sans modération !

Dresser un état des lieux : `git status`

Exemple[3] :

- on ajoute le fichier `README` : `$git add README`

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
```

- on modifie le fichier `CONTRIBUTING.md`

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

⇒ il n'est pas encore "stagé".

3. Tiré de Chacon and Straub (2014)

Dresser un état des lieux : `git status`

Exemple (suite) :

- on modifie le fichier `CONTRIBUTING.md`
- on prend en compte les modifications via `git add` :
  - `$git add CONTRIBUTING.md`

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md
```

⇒ la modification est prise en compte.

⇒ elle sera enregistrée au prochain commit.

Créer une nouvelle version : `git commit`

La commande `git commit` crée une nouvelle version.

Il faut spécifier un message décrivant la nature du commit :

- `$git commit` ouvre un éditeur de texte pour le taper
- `$git commit -m "my message"` le spécifie directement

4. https://xkcd.com/1296/

Créer une nouvelle version : `git commit`

La commande `git commit` crée une nouvelle version.

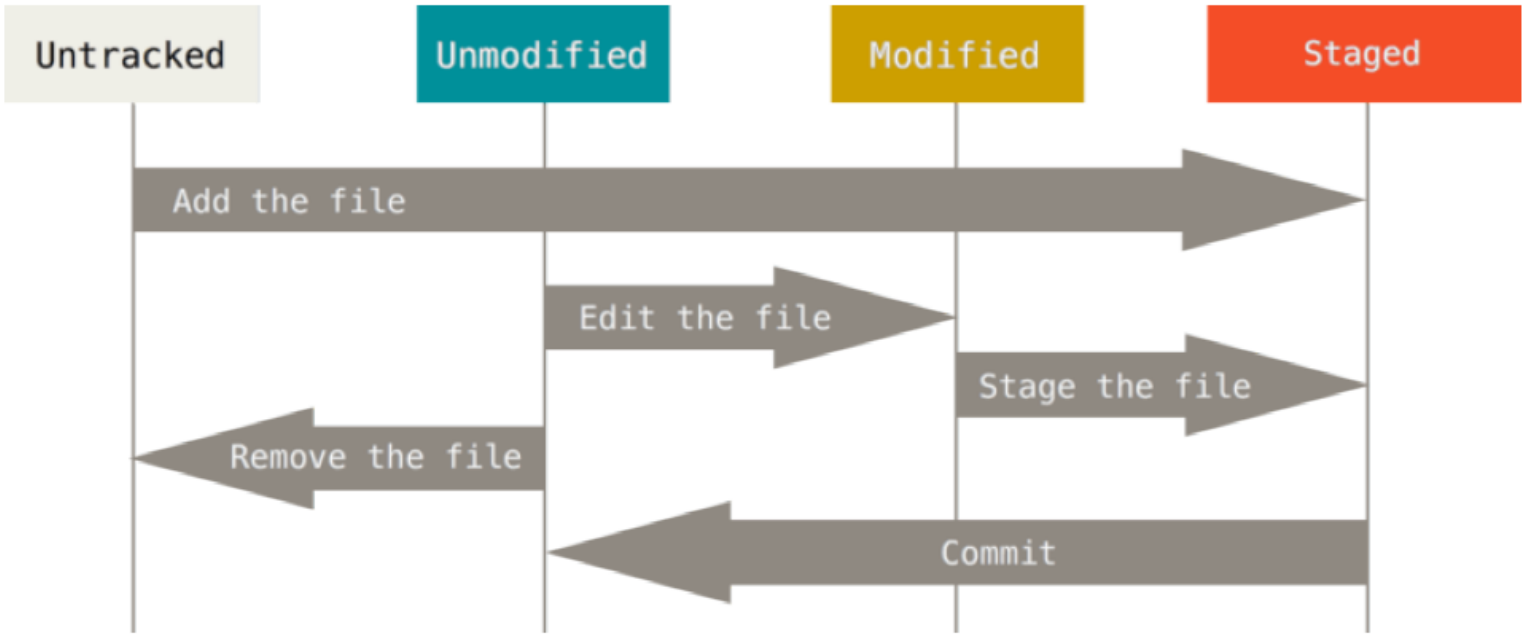
Il faut spécifier un message décrivant la nature du commit :

- `$git commit` ouvre un éditeur de texte pour le taper
- `$git commit -m "my message"` le spécifie directement

⇒ bonne pratique : bien renseigner le message[4]....

4. https://xkcd.com/1296/

Tracked / modified / staged : le cycle de vie d'un fichier



Voir les modifications d'un fichier : `git diff`

La commande `git diff` permet de visualiser les changements apportés aux fichiers :

1. `$git diff` : les modifications non encore "stagées"
   - working dir vs staged
2. `$git diff --staged` : les modifications déjà "stagées"
   - staged vs repository

Voir les modifications d'un fichier : `git diff`

La commande `git diff` permet de visualiser les changements apportés aux fichiers :

1. `$git diff` : les modifications non encore "stagées"
   - working dir vs staged
2. `$git diff --staged` : les modifications déjà "stagées"
   - staged vs repository

⇒ affiche les résultats dans la console

- +/- : ce qui est ajouté/supprimé

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
 Please include a nice description of your changes when you submit your PR;
 if we have to read the whole diff to figure out why you're contributing
 in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if your patch is
+longer than a dozen lines.

 If you are starting to work on a particular area, feel free to submit a PR
 that highlights your work in progress (and note in the PR title that it's
```

Voir l'historique des commits : `git log`

La commande `git log` permet de voir l'historique.

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit
```

⇒ identifiants, date, auteur...et message d'information !

⇒ de nombreuses manières de "customiser"

- stats., concis/détaillé...cf Chacon and Straub (2014)

Supprimer / déplacer un fichier : `git rm` & `git mv`

La commande `$git rm my_file` permet de supprimer un fichier :

1. supprime le fichier en local (via la commande `rm`)
2. enregistre la suppression dans la "staging area"

⇒ il sera supprimé du dépôt au prochain commit.

Supprimer / déplacer un fichier : `git rm` & `git mv`

La commande `$git rm my_file` permet de supprimer un fichier :

1. supprime le fichier en local (via la commande `rm`)
2. enregistre la suppression dans la "staging area"

⇒ il sera supprimé du dépôt au prochain commit.

De la même manière, la commande `$git mv my_file my_new_file` permet de déplacer (ou renommer) un fichier :

1. déplace le fichier en local (via la commande `mv`)
2. enregistre le changement dans la "staging area"

⇒ il sera déplacé dans le dépôt au prochain commit.

Annuler une modification : `git reset` et `git checkout`

Les commandes `git reset` et `git checkout` permettent d'annuler des modifications :

- `git reset` enlève un fichier de la "staging area"
  - la modification ne sera pas prise au prochain commit
  - s'appelle ainsi : `$git reset -- my_file`
- git checkout annule une modification
  - revient à l'état du commit précédent
  - s'appelle ainsi : `$git checkout -- my_file`

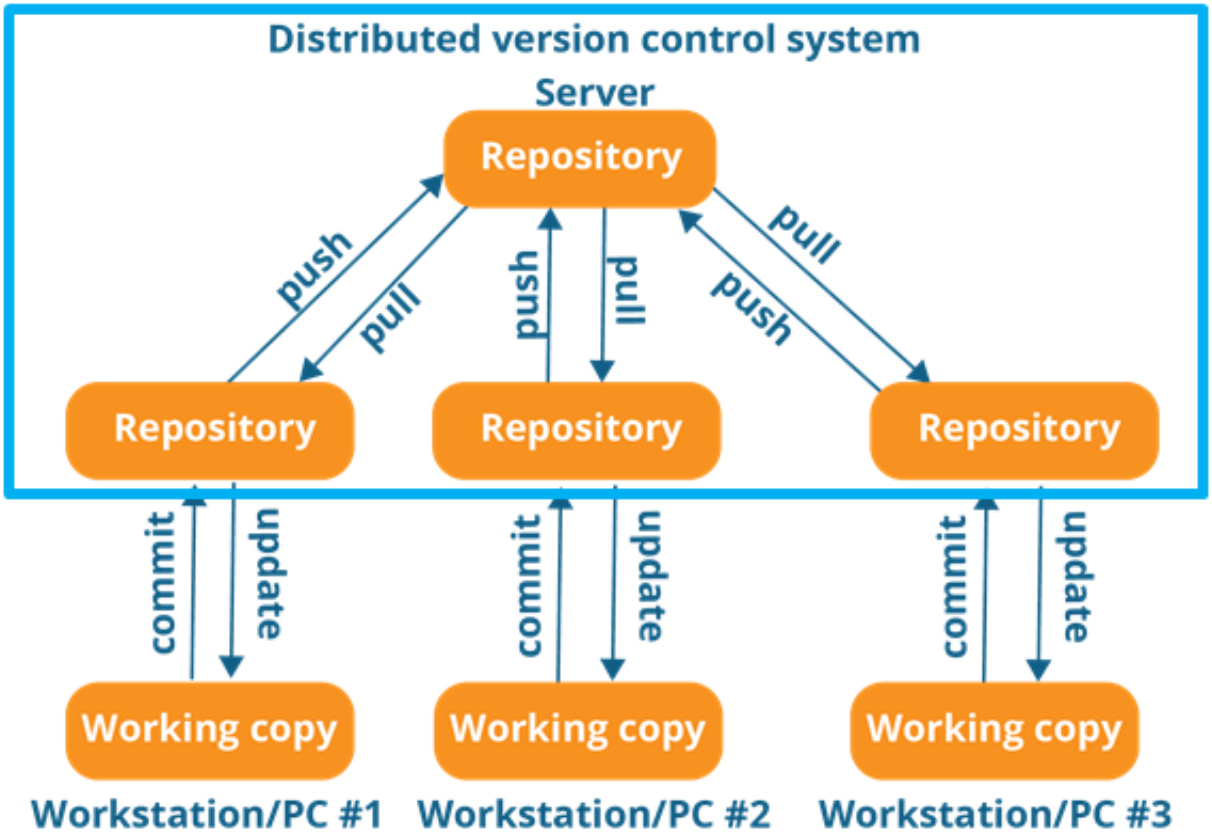
⇒ à noter que `git status` vous rappelle tout ça !

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

III - Systèmes décentralisés & git

- principes de base
- opérations de base
- travailler un sur serveur distant
- git & branches

Travailler sur un serveur distant



⇒ les "remote" de `git`

⇒ commandes clés : `git push` et `git pull`

Serveurs distants ?

Serveur distant ?

- hebergé / accessible via internet
- hebergé / acessible au sein de votre entreprise / institut

⇒ on y accède en général via les protocoles `https` ou `ssh`.

Plateformes en ligne classiques : GitHub, GitLab, Bitbucket

- hébergement de projets `git`
- accès +/- gratuit (+/- de fonctionalités)
- hébergent de nombreux projets open-source
- permettent d'y accéder facilement (voire d'y contribuer)
  - "forks" et "pull requests"

⇒ TP : créer un projet sur GitHub et le partager.

Travailler sur un serveur distant

Travailler sur un serveur distant = se synchroniser (à plusieurs) par rapport à une version centrale.

On y accède en créant une connection "remote".

Pour voir la liste de ses "remotes" :

- `$git remote` : liste leur identifiants
- `$git remote -v` : donne en plus leur adresse

```
$ git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
```

Se connecter à un serveur distant

Pour ajouter un "remote" : `git remote add <id> <url>`

```
$ git remote
origin
$ git remote add pb https://github.com/paulboone/ticgit
$ git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
pb  https://github.com/paulboone/ticgit (fetch)
pb  https://github.com/paulboone/ticgit (push)
```

⇒ on peut avoir plusieurs "remote" sur un même projet.

- e.g., pointant vers différents collègues développeurs

Se connecter à un serveur distant

Pour ajouter un "remote" : `git remote add <id> <url>`

```
$ git remote
origin
$ git remote add pb https://github.com/paulboone/ticgit
$ git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
pb  https://github.com/paulboone/ticgit (fetch)
pb  https://github.com/paulboone/ticgit (push)
```

⇒ on peut avoir plusieurs "remote" sur un même projet.

- e.g., pointant vers différents collègues développeurs

Mais en général, on commence par se synchroniser avec un projet existant via `$git clone <url>` :

1. crée en local un "clone" ("mirroir") du projet
   - contenu + **historique**
2. initialise la remote `"origin"` (pointant vers `<url>`)

Mettre à jour le serveur distant

On met à jour le serveur distant via la commande `git push`.

Typiquement : `$git push origin master`

- on "pousse" sur la remote `origin` le contenu de la branche (locale) `master`

Mettre à jour le serveur distant

On met à jour le serveur distant via la commande `git push`.

Typiquement : `$git push origin master`

- on "pousse" sur la remote `origin` le contenu de la branche (locale) `master`

Branche `master` ?

- on est toujours sur une "branche" avec `git`
- la branche `master` : celle par défaut (toujours présente)
- mais on peut très bien pousser une autre branche...

⇒ à suivre !

Mettre à jour le serveur distant

On met à jour le serveur distant via la commande `git push`.

Typiquement : `$git push origin master`

- on "pousse" sur la remote `origin` le contenu de la branche (locale) `master`

Branche `master` ?

- on est toujours sur une "branche" avec `git`
- la branche `master` : celle par défaut (toujours présente)
- mais on peut très bien pousser une autre branche...

⇒ à suivre !

Remarque : on ne peut "pousser" à distance que si on a fait un/des commit(s) en local.

Se synchroniser par rapport au serveur distant

Pour se synchroniser par rapport au serveur distant : `git pull` et `git fetch`.

Typiquement : `$git pull origin master`

- on "tire" sur la branche (locale) `master` le contenu de la remote `origin`

⇒ met effectivement à jour la version locale.

La commande `$git fetch origin` :

- récupère les modifications faites à distance
- **sans** les implémenter en local

⇒ permet d'inspecter les modifications [5] ... avant de les implémenter via `$git merge origin/master`

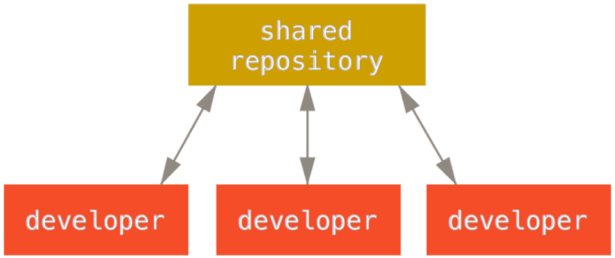
⇒ `git pull` = `git fetch` + `git merge`

5. e.g., via `$git diff master..origin/master`

Vers des workflows réellement distribués...

Workflow centralisé :

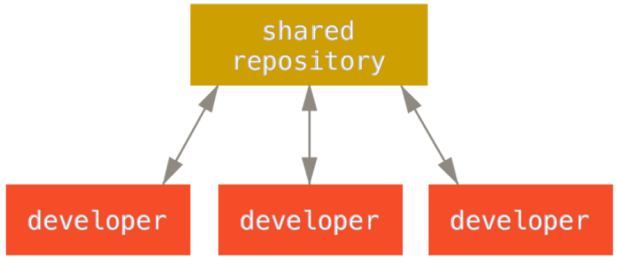
- 1 dépôt central (~ `subversion`)
- totalement partagé



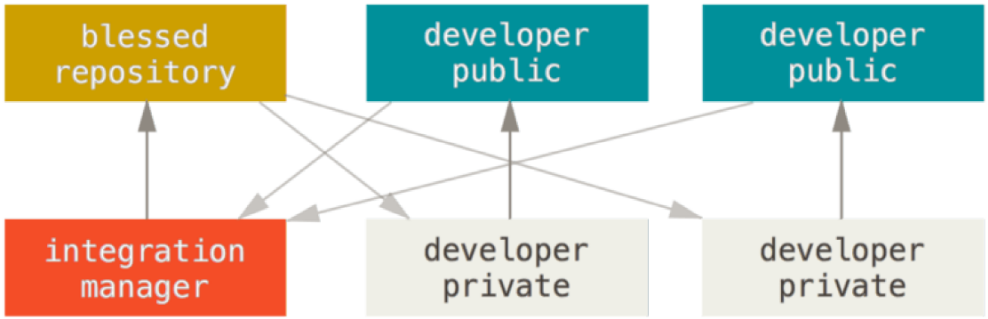
Vers des workflows réellement distribués...

Workflow centralisé :

- 1 dépôt central (~ `subversion`)
- totalement partagé



Workflow distribué (un exemple) :



- chacun travaille sur son propre "fork" du dépôt central
- "integration manager" responsable du dépôt central
  - le met à jour après des "**pull requests**"

⇒ plus de flexibilité, meilleure gestion des conflits.

III - Systèmes décentralisés & git

- principes de base
- opérations de base
- travailler un sur serveur distant
- git & branches

Branches ?

Branche = ligne de développement parallèle

- permet de travailler sans perturber la ligne principale
- développer une nouvelle fonctionalité, corriger un bug

Branches ?

Branche = ligne de développement parallèle

- permet de travailler sans perturber la ligne principale
- développer une nouvelle fonctionalité, corriger un bug

Une fonctionnalité clé de `git`

- beaucoup plus souple que sur autres systèmes

Branches ?

Branche = ligne de développement parallèle

- permet de travailler sans perturber la ligne principale
- développer une nouvelle fonctionalité, corriger un bug

Une fonctionnalité clé de `git`

- beaucoup plus souple que sur autres systèmes

⇒ des chaînes de commit différentes :

- vouées à se rejoindre ou non

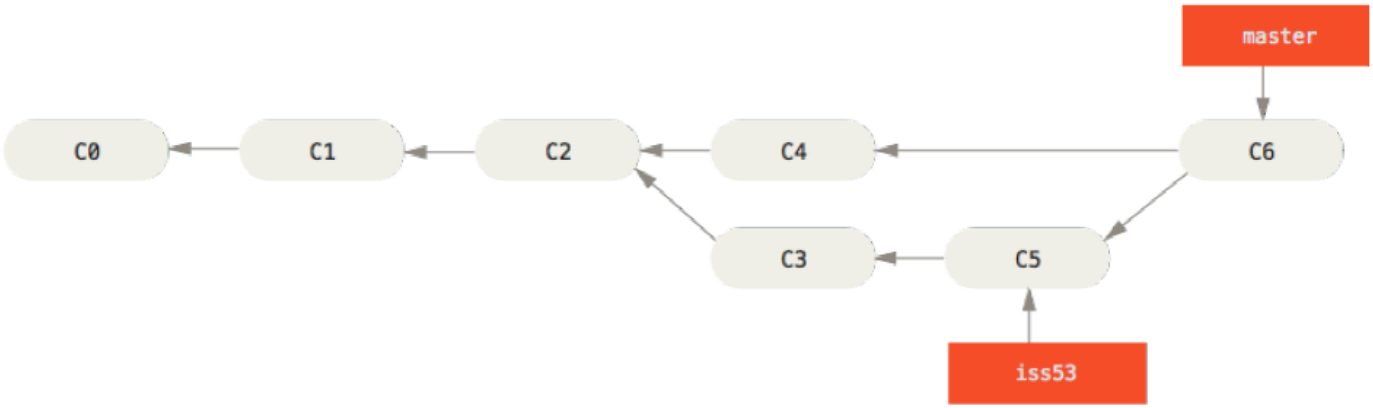
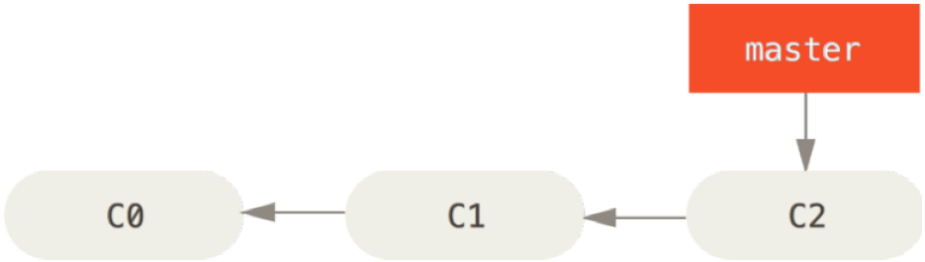


Illustration tirée de Chacon and Straub (2014)

Etape 0 : on est sur notre branche maitre

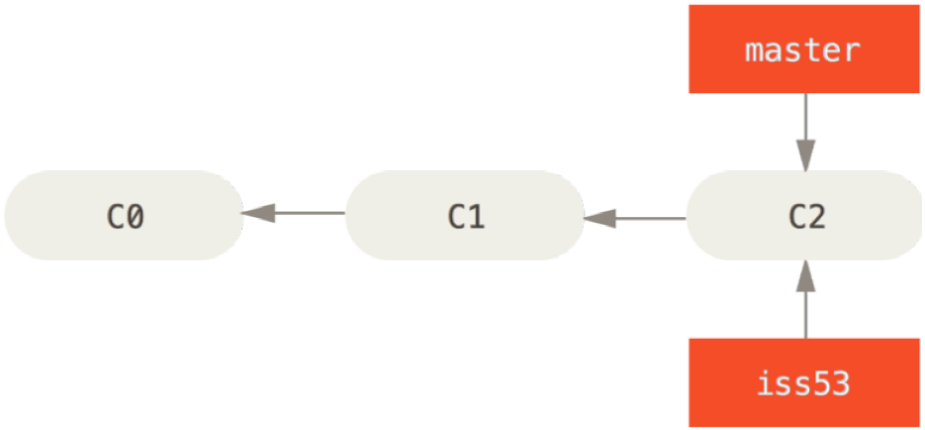
- la branche **master**, toujours présente



⇒ on veut développer une nouvelle fonctionalité

Illustration tirée de Chacon and Straub (2014)

Etape 1 : on créé une nouvelle branche



⇒ pointe sur le dernier commit de la branche `master`

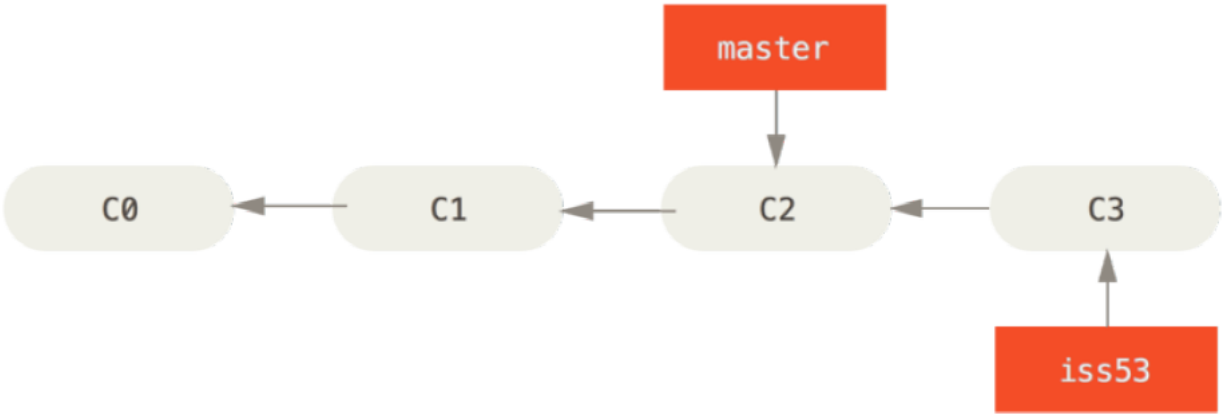
Commandes `git` :

- `$git branch <branch>` : crée la branche
  - `$git branch` : donne la liste des branches existantes
- `$git checkout <branch>` : bascule sur la branche

(`$git checkout -b <branch>` : fait les deux d'un coup)

Illustration tirée de Chacon and Straub (2014)

Etape 2 : on fait évoluer la nouvelle branche



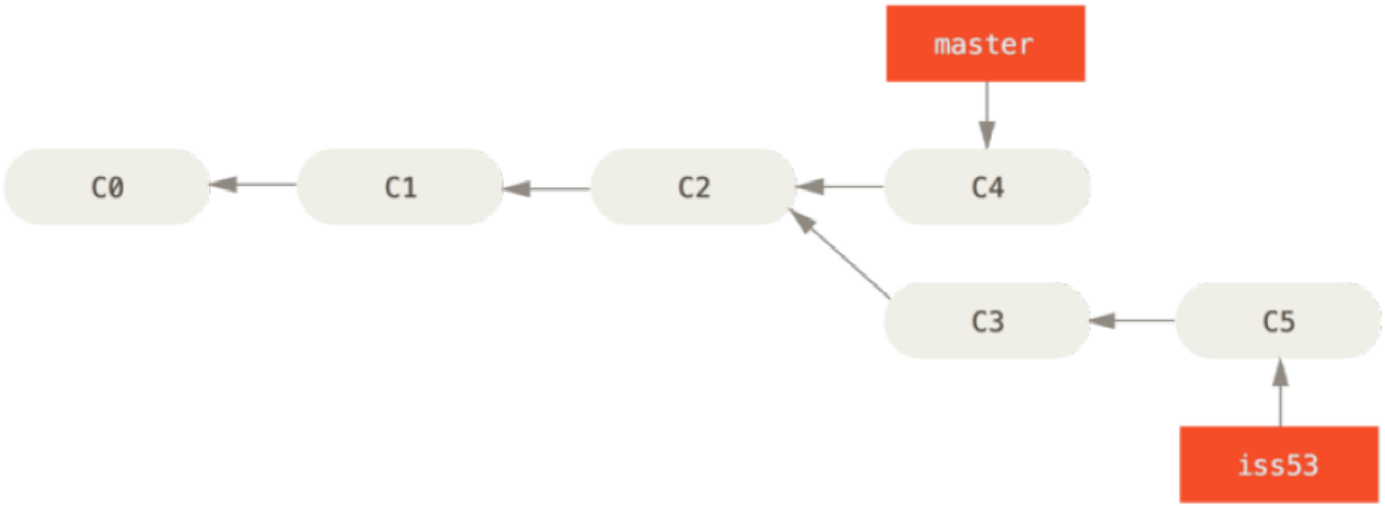
⇒ on crée des nouveaux commit sur la nouvelle branche

- via des `git add` et `git commit`

⇒ la branche `master` pointe sur l'ancien commit.

Illustration tirée de Chacon and Straub (2014)

Etape 3 : on fait évoluer les deux branches en parallèles



⇒ on crée des nouveaux commit sur chacune des branches

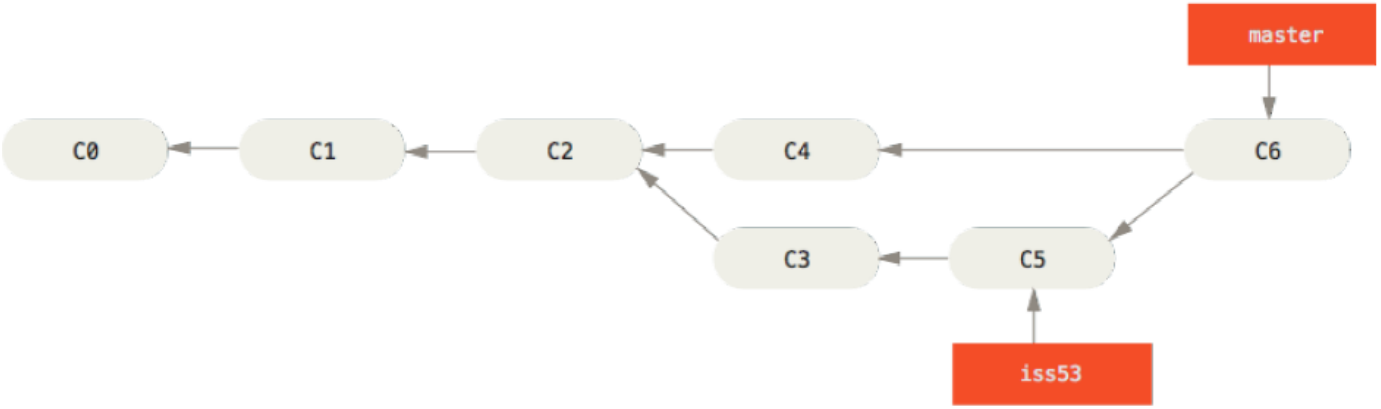
- via des `git add` et `git commit`

⇒ on change de branche via `$git checkout <branch>`

- ici `git checkout master` ou `git checkout iss53`

Illustration tirée de Chacon and Straub (2014)

Etape 4 : enfin, on fusionne les branches



⇒ implémente les modifications sur la branche `master`

⇒ la fusion donne lieu à un nouveau commit

Commandes `git` :

- `$git merge <branch>` : fusionne la branche
  - ici `$git merge iss53`, à partir de la branche `master`
- `$git branch -d <branch>` : supprime la branche
  - le pointeur, pas l'historique des commit

Récapitulatif

Si on récapitule :

- branche principale : branche `master`
- créer une branche : `$git branch <branch>`
  - `$git branch` : donne la liste des branches existantes
- changer de branche : `$git checkout <branch>`
- fusionner des branches : `$git merge <branch>`
  - à lancer à partir de la branche `master`
- supprimer une branche : `$git branch -d <branch>`
  - une fois qu'on n'a plus besoin de la branche
  - pas obligatoire mais recommandé
  - ne supprime pas l'historique des commit

Récapitulatif

Si on récapitule :

- branche principale : branche `master`
- créer une branche : `$git branch <branch>`
  - `$git branch` : donne la liste des branches existantes
- changer de branche : `$git checkout <branch>`
- fusionner des branches : `$git merge <branch>`
  - à lancer à partir de la branche `master`
- supprimer une branche : `$git branch -d <branch>`
  - une fois qu'on n'a plus besoin de la branche
  - pas obligatoire mais recommandé
  - ne supprime pas l'historique des commit

⇒ Point clé :

- branches = plusieurs versions d'un même fichier
- quand on change de branche, le fichier change

Branches et serveur distant

Mêmes mécanismes que précédemment :

- `$git push origin <branch>` pour "pousser" la branche `<branch>` sur la "remote" `origin`
  - précédemment : `$git push origin master`
  - NB : la première fois, crée la branche sur la remote.
- `$git pull origin <branch>` pour mettre à jour la branche `<branch>` à partir de la "remote" `origin`
  - précédemment : `$git pull origin master`

Pour supprimer la branche de la "remote" :

- `$git push origin --delete <branch>`

IV - Remarques et conclusion

Conclusion

Contrôle de version : outil indispensable du data-scientist

- développement collectif
- historique et traceabilité
- recherche reproductible

⇒ ce cours : une introduction à `git`.

Conclusion

Contrôle de version : outil indispensable du data-scientist

- développement collectif
- historique et traceabilité
- recherche reproductible

⇒ ce cours : une introduction à `git`.

`git` : le nouveau standard

- système décentralisé
- développement par "branche"
- plateformes en ligne GitHub, GitLab, Bitbucket

⇒ riche et complexe.
⇒ maitriser les commandes de base ouvre beaucoup de portes

Conclusion

`git` : commandes de bases

- en local : `init, add, commit, status, log`
- serveur distant : `clone, push, pull, fetch`
- branches : `branch, checkout, merge`

TP :

1. créer localement un projet `git`
2. interfacer le projet avec GitHub et le partager
3. créer une nouvelle branche

Références

John D. Blischak, Emily R. Davenport, and Greg Wilson. A quick introduction to version control with git and github. *PLOS Computational Biology*, 12(1) :1–18, 01 2016. doi : 10.1371/journal.pcbi.1004668. URL https://doi.org/10.1371/journal.pcbi.1004668.

Scott Chacon and Ben Straub. *Pro git : Everything you need to know about Git*. Apress, second edition, 2014. URL https://git-scm.com/book/en/v2.

William Stafford Noble. A quick guide to organizing computational biology projects. *PLOS Computational Biology*, 5(7) :1–5, 07 2009. doi : 10.1371/journal.pcbi.1000424. URL https://doi.org/10.1371/journal.pcbi.1000424.