

# TD 6: Développement collaboratif décentralisé

ED6 — Licence 3 — Université Paris Diderot

Séances du 2 et 9 mai 2012

Git est un logiciel de contrôle de versions décentralisé. Contrairement à Subversion, qui se base sur un unique dépôt avec lequel se synchronisent une ou plusieurs copies de travail (les copies de travail ne peuvent se synchroniser entre elles : elles doivent passer par le dépôt), chaque copie de travail d'un projet versionné avec Git joue aussi le rôle de dépôt, et il est possible de synchroniser entre elles n'importe quelles copies de travail. De plus, Git permet d'utiliser une ou plusieurs *branche de développement* et de fusionner entre elles ces branches de développement.

## 1 Publication de révisions

Nous allons tout d'abord nous intéresser à l'aspect gestionnaire de version de Git : comment enregistrer l'historique des modifications apportées à un projet. Pour initialiser un repository, il convient d'invoquer la commande `git init monrepo`. Cette commande initialise un dépôt Git dans le répertoire `monrepo` (qui est créé si celui-ci n'existe pas). Ce répertoire contient alors à la fois une version de travail (dans `monrepo`) et un repository Git (dans `monrepo/.git`). Bien que Git ait été conçu pour gérer du code source, nous allons nous en servir dans ce TP pour gérer des fichiers textes simples, pour nous concentrer sur le fonctionnement de Git plutôt que sur du code.

**Question 1.1.** *Initialiser un dépôt Git, et créez le fichier `burger.txt` qui contient la liste des ingrédients d'un burger (un ingrédient par ligne).*

```
steak
salade
tomate
cornichon
fromage
```

Git a plusieurs interfaces utilisateur, la plus complète étant l'interface en ligne de commande (CLI), mais nous allons aussi utiliser Gitg qui est une interface graphique à Git.

Depuis le répertoire de votre dépôt, lancez Gitg (ou lancez Gitg puis ouvrez votre dépôt). Gitg a deux onglets *History* et *Commit*. Dans l'onglet *Commit*, on remarque 4 cadres :

- *Unstaged* qui contient la liste des modifications qui ont été apportées dans le dépôt et qui n'ont pas été sélectionnées pour être committées.
- *Staged* qui contient la liste des modifications qui ont été apportées et qui ont été sélectionnées pour être committées.
- *Changes* qui affiche une modification
- *Commit message* qui contient le message du commit courant

**Question 1.2.** *Sélectionnez votre fichier `burger.txt` comme modification à être committée ; éditez un message de commit, puis commitez. Retournez dans l'onglet *History* pour observer votre commit.*

**Question 1.3.** *Rajoutez un ingrédient dans `burger.txt`, puis créez quelques autres `sandwich.txt` et commitez toutes ces modifications. Regardez l'onglet *History*, votre deuxième commit doit apparaître.*

**Question 1.4.** *Créez un nouveau sandwich, et modifiez un sandwich existant. Nous allons committer ces changements avec l'interface en ligne de commande. Commençons par taper `git commit`, que se passe-t-il ? Lisez le paragraphe *DESCRIPTION* de la page de manuel `git-commit(1)` (ce que l'on peut faire soit en invoquant `man git-commit`, soit `git help commit`). Commitez ces changements. Observer l'arbre de commit dans l'onglet *History* de Gitg (Attention, il faudra sans doute rafraîchir avec `Ctrl+R`).*

## 2 Branches de développement

La partie 1 présentait l'utilisation simple de Git pour créer un historique des modifications. Nous allons maintenant nous concentrer sur la notion de *branche*. Lors du développement d'un projet, il peut arriver que l'on veuille introduire

une nouvelle fonctionnalité dans le projet, sans “casser” le projet. Nous voudrions donc pouvoir basculer instantanément de la version stable du projet à sa version “en développement”. C’est ce que nous permettent de faire les branches.

**Question 2.1.** *Creez une nouvelle branche intitulee developpement dans votre repository. Avec Gitg, dans l'onglet History, selectionnez le dernier commit (attention l'etiquette master represente la branche master), clic-droit puis Create New Branch.*

Nous avons donc créé une nouvelle branche, qui est pour l’instant la même que la branche principale. Il est possible de basculer d’une branche à l’autre en cliquant droit sur la branche et sélectionnant Checkout working copy.

**Question 2.2.** *Dans la branche developpement effectuez quelques modifications (modifications dans les fichiers ou ajout/suppression de fichier), puis committez-les. Observez ce qu’il se passe dans l'onglet History. (Selectionnez l’affichage de toutes les branches).*

En ligne de commande, pour afficher la branche courante, il suffit d’invoquer `git branch`. Une branche est créée à partir d’une autre au moyen de `git branch MANOUELLEBRANCHE`, et le passage d’une branche à l’autre se fait au moyen de `git checkout MABRANCHE`.

**Question 2.3.** *Constatez les différences dans chacune des deux branches.*

Jusqu’à présent nous avons suivi le scénario simple où il y a une branche de développement, et une branche stable. Supposons que les modifications dans la branche de développement soient finies et que nous voulions nous lancer dans de nouvelles modifications, il convient de synchroniser les deux branches. Cela s’appelle une fusion de branches (*merge* en anglais).

**Question 2.4.** *Essayez sous Gitg, de fusionner la branche master avec la branche developpement. Est-ce le resultat attendu ?*

Ce scénario simple ne s’applique pas toujours : pendant que des modifications sont effectuées dans la branche de développement, il peut falloir aussi effectuer des correctifs mineurs dans la branche stable.

**Question 2.5.** *Effectuez (et committez) des modifications dans la branche developpement et d'autres modifications dans la branche master (attention a ce que ces modifications ne soient pas concurrentes). Qu'est-ce qui est affiche dans l'onglet History de Gitg ?*

Il est bien entendu possible de fusionner deux branches avec l’interface en ligne de commande. Lisez le paragraphe DESCRIPTION dans la page de manuel de `git-merge`(1)

**Question 2.6.** *Avec l'interface en ligne de commande, fusionnez les branches master et developpement.*

Jusqu’à présent, nous n’avons envisagé que des scénarios dans lesquels la fusion des branches est simple, mais il peut y arriver qu’il y ait des conflits, par exemple un même bogue corrigé de manière sensiblement différente dans deux branches différentes.

**Question 2.7.** *Que se passe-t-il dans ce cas-la ? Essayez d'implementer ce scenario. Comment Git vous permet-il de resoudre les conflits ? Ecrase-t-il unilateralement les modifications effectuees dans une branche ?*

### 3 Synchronisation de plusieurs repositories

Jusqu’à présent, nous avons vu quelques fonctionnalités de Git sans nous intéresser son aspect collaboratif. Git permet un travail collaboratif sur un dépôt. C’est-à-dire qu’il est possible de synchroniser entre elles des branches de deux dépôts différents.

**Question 3.1.** *Creez un nouveau depot (avec `git init --bare`).*

Ceci initialise un dépôt Git sans copie de travail. Il y a deux façons de synchroniser entre eux deux dépôts :

- soit en récupérant les commits du dépôt distant (*pull*)
- soit en envoyant des commits vers le dépôt distant (*push*).

Dans le deuxième cas, il faut que le dépôt distant soit un dépôt *bare*, c’est à dire sans copie de travail.

**Question 3.2.** *Envoyez les commits de votre premier depot vers le second avec les commandes (executees depuis votre premier depot) :*

```
git push file:/// $PATH_TO_REPO2 master:master
```

Cette commande va envoyer la branche *master* du premier dépôt dans une branche appelee *master* dans le second dépôt. La seconde va effectuer la même chose avec la branche *developpement*.

```
git push file:/// $PATH_TO_REPO2 developpement:developpement
```

Observez le resultat en lançant *Gitg* depuis le second depot.

Pour rendre la synchronisation plus intéressante, nous allons utiliser une deuxième copie de travail.

**Question 3.3.** *Creez une nouvelle copie de travail a partir du dépôt bare :*

*git clone file:///PATH\_TO\_REPO2 copietravail*

*qui cree une copie de travail du second dépôt dans le repertoire copietravail .*

**Question 3.4.**

**Question 4.1.** *Implementez ce scenario, constater que `git push` renvoie une erreur, puis au lieu d'invoquer `git merge`, invoquez `git rebase origin/master` (*origin/master* étant le nom de la branche distante avec laquelle on voudrait normalement effectuer un merge). Git va recreer les commits de votre branche master qui ne sont pas dans la branche master du dépôt repo et va les placer au dessus du dernier commit de la branche master de repo).*

Attention, il est très fortement déconseillé de rebase des commits qui ont déjà été publiés, c'est à dire présent sur un autre dépôt. La question suivante va donc vous montrer ce qu'il faut éviter de faire.

**Question 4.2.** *Synchronisez vos 3 dépôts. Dans le dépôt 3, effectuez un commit. Publiez-le dans le depot 2. Dans le dépôt 1 effectuez d'autres commits, recuperez ce commit dans le dépôt 3 a l'aide de `git fetch repo1`<sup>2</sup>, puis (dans le dépôt 3) rebasez votre branche master au dessus de la branche `repo1/master` avec `git rebase repo1/master`.*

*Que se passe-t-il si on essaie de merger cette branche avec la branche presente dans le dépôt 2 ?*

Un autre cas de modification de commit est avec la sous-commande `amend` de Git. `amend` permet d'éditer, de modifier le contenu d'un commit. Supposons qu'on vienne de commiter un commit intitulé orthographe et qu'il corrige des fautes d'orthographe. Supposons qu'une faute ne soit pas corrigé par ce commit, et qu'on ne veuille pas créer un autre commit par dessus, il est possible de modifier le dernier commit avec `git commit --amend`.

**Question 4.3.** *Creez un commit, puis apportez d'autres modifications, et editez le precedent commit au lieu d'en creer un nouveau.*

Là encore, il est impératif de ne pas éditer un commit qui a déjà été publié, c'est ce que la dernière question vous demande de faire et qu'il faut éviter de faire :

**Question 4.4.** *Creez un commit, envoyez-le vers un dépôt distant, puis amendez votre commit, synchronisez votre dépôt avec le dépôt distant.*

---

2. car nous avons déclaré une remote appelée `repo1` dans le dépôt 3 qui pointe vers le premier dépôt.