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 repositoire, 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 repertoire contient alors à la fois une version de travail (dans `monrepo`) et un repositoire 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 commitées.
- Staged qui contient la liste des modifications qui ont été apportées et qui ont été selectionnées pour être commitées.
- 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 commitée ; 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 commiter 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. *Créez une nouvelle branche intitulée developpement dans votre repositoire. Avec Gitg, dans l'onglet History, sélectionnez le dernier commit (attention l'étiquette master représente 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 commitez-les. Observez ce qu'il se passe dans l'onglet History. (Sélectionnez 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 MANOUVELLEBRANCHE`, 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 développement. Est-ce le résultat 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 commitez) des modifications dans la branche developpement et d'autres modifications dans la branche master (attention à ce que ces modifications ne soient pas conflictuelles). Qu'est-ce qui est affiché 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 lequels 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-là ? Essayez d'implémenter ce scénario. Comment Git vous permet-il de résoudre les confits ? Écrase-t-il unilatéralement les modifications effectuées dans une branche ?*

3 Synchronisation de plusieurs repositoires

Jusqu'à présent, nous avons vu quelques fonctionalités 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. *Créez un nouveau dépôt (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 dépôt vers le second avec les commandes (exécutées depuis votre premier dépôt) :*

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

Cette commande va envoyer la branche master du premier dépôt dans une branche appelée 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 résultat en lançant Gitg depuis le second dépôt.

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

Question 3.3. *Créez une nouvelle copie de travail à partir du dépôt* bare *:*
`git clone file:///$PATH_TO_REPO2 copietravail`
qui crée une copie de travail du second dépôt dans le répertoire `copietravail`.

Question 3.4. *Effectuez quelques modifications dans votre première copie de travail. Propagez ces modifications dans votre troisième dépôt :*
- *Envoyez ces modifications dans le second dépôt (avec* `git push file:///PATH_TO_REPO2 BRANCHE:BRANCHE`*)*
- *Puis depuis le troisième dépôt, récupérez avec* `git pull` [1] *ces modifications depuis le second dépôt.*

Nous avons mis en place avec le second et troisième dépôt le schéma de collaboration avec Git le plus courant : il y a un dépôt qui fait office de dépôt maître, et le troisième dépôt qui peut récupérer et envoyer des commits sur le dépôt maître. Nous allons maintenant nous intéresser à l'accès concurrent à ce dépôt maître.

Question 3.5. *Effectuez des modifications dans le premier dépôt, et envoyez ces modifications dans le second dépôt. Sans synchroniser le troisième dépôt avec le second, effectuez (et commitez) des modifications dans le troisième dépôt. Que se passe-t-il maintenant lorsqu'on fait* `git pull` *dans le troisième dépôt ?*

Question 3.6. `git pull` *peut être décomposé en* `git fetch` *suivi de* `git merge`. *Réitérez le scénario de la question précédente mais en faisant* `git fetch` *au lieu de* `git pull`*, observez "toutes les branches" du dépôt 3 dans Gitg.*

Un des aspects fondamental de Git est qu'il est décentralisé. Nous avons ici donné un rôle spécial de dépôt central au dépôt 2, mais s'il venait à disparaître, il serait toujours possible de synchroniser entre eux les dépôts 1 et 3.

Question 3.7. *Dans le dépôt 1, nous allons déclarer l'addresse du dépôt 3, nous allons créer pour cela une* remote *appelée repo3.*
`git remote add repo3 file:///chemin/vers/repo3`
De même, dans le dépôt 3, nous allons créez une remote *appelée repo1 qui pointe vers le premier dépôt.*

Question 3.8. *Effectuez (et commitez) des modifications dans le dépôt 3 et récupérez-les dans le dépôt 1 au moyen de :*
```
git fetch repo3
git checkout master
git merge remotes/repo3/master
```

Nous pouvons simplifier cette démarche en déclarant que la branche master du dépôt 1 "suit" la branche master du dépôt 3, à l'aide de (depuis le dépôt 1) `git branch master --set-upstream repo3/master`.

Question 3.9. *Effectuez des modifications dans le dépôt 1 puis récupérez-les dans le dépôt 3 au moyen de* `git pull`.

Question 3.10. *Synchronisez entre eux les trois dépôts.*

4 Modifications publiées, modifications non publiées

Nous avons vu que les objets que s'échangent les dépôt gits sont des commits. Afin de maintenir une intégrité des arbres de commit, Git utilise des primitives cryptographiques. Chaque commit est en fait signé, en fonction du patch qu'il représente, du nom d'auteur, de la date de création, et aussi de la signature du commit parent (ou des deux parents, dans le cas d'un commit de fusion). Cette signature est un hachage SHA1 de toutes ces informations, et il est possible de se référer à un commit uniquement par cette signature (de la forme f4ccba7ba89d4f6f8f0853056d47912c640a19c1) ou par un préfixe non ambigu de celle-ci (f4ccba7b).

Ainsi Git n'appliquera pas un commit ailleurs que sur son père. L'utilisateur de Git pourra vouloir appliquer un commit ailleurs dans l'arbre de commit (par exemple sur une autre branche), il va pour cela devoir créer un nouveau commit (c'est à dire avec un SHA1 différent) mais qui contient les mêmes modifications.

Supposons que nous clonions un dépôt (qu'on appelera repo et nommons commitA le dernier commit sur ce dépot), et que nous effectuions un commit dans notre copie de travail (commitB dont le père est commitA). Parallèlement, un autre développeur effectue un commit commitC au dessus du commitA et envoie ce commit dans le dépôt repo. Il n'est plus possible d'envoyer notre commit commitB, car le dernier commit du dépot repo n'est pas commitA, mais commitC. La stratégie consiste alors à fusionner notre branche locale avec la branche distante, ce qui va créer un commit de fusion, fils de commitB et commitC et d'envoyer ce commit et le commit commitB vers repo.

Nous allons voir qu'une autre stratégie est possible, de demander (avec la commande rebase) à Git de "modifier" le commitB pour que son père soit commitC

1. Comme le troisième dépôt a été créé à partir du second au moyen de `git clone`, il n'est pas nécessaire de préciser ici où Git doit chercher les commits.

Question 4.1. *Implémentez ce scénario, 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 recréer 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. *Synchonisez vos 3 dépôts. Dans le dépôt 3, effectuez un commit. Publiez-le dans le dépot 2. Dans le dépôt 1 effectuez d'autres commits, récupérez ce commit dans le dépôt 3 à l'aide de* `git fetch repo1` [2]*, 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 présente 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'orthographes. 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. *Créez un commit, puis apportez d'autres modifications, et éditez le précédent commit au lieu d'en créer 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. *Créez 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.