Tutoriel git

Régis Briant, Youngseob Kim & Dmitry Khvorostyanov

Table des matières

Introduction

Git est un gestionnaire de version, libre et très performant. Il possède de nombreux avantages par rapport à svn, notamment, la possibilité de travailler localement. C'est à dire de faire des commits local et de les éditer localement avant de les pousser vers un serveur pour qu'ils soient intégrés au dépôt central.

Voici une liste d'avantage que possède git par rapport à svn :

- Pouvoir travailler en local tout en ayant accès au dépôt central.
- L'utilisation de plusieurs branches est simplifiée.
- Pouvoir éditer des commits précédents avec **git rebase**.
- Pouvoir switcher rapidement d'une version à une autre et les comparer sans changer de dossier et sans avoir à communiquer avec le serveur distant.
- Chaque utilisateur possède une copie complète du dépôt central, contrairement à svn ou chaque utilisateur possède uniquement une copie d'une version donnée. Cela implique qu'en cas de problème sur le serveur git on trouvera sans problème une version backup du dépôt ailleurs. Les échanges avec le serveur sont aussi moins importants.
- Git conserve un historique accessible avec la commande **git reflog**. Si un commit ou une branche à été malencontreusement supprimé de l'arbre de développement, on peut aisément la récupérer. Il est donc rare qu'une modification antérieure soit vraiment perdu.
- Il n'y a un seul répertoire .git, contrairement à svn où l'on a un répertoire .svn dans chaque sous-dossier (important lorsque l'on fait des grep dans les dossiers).

Ce tutoriel vous permettra de prendre en main git. La Section 1 vous permettra d'éditer le fichier de configuration, la Section 2 vous montrera comment initialiser un dépôt git. La Section 3 vous montrera comment faire des commits et les éditer localement. Enfin, la Section 4 vous montrera comment gérer des branches distantes.

1 Configurer git

Tout comme il existe un fichier de configuration pour svn (~/.subversion/config), il exite un fichier de configuration pour git (~/.gitconfig). Ce fichier contient des informations sur vous : nom, adresse e-mail, éditeur de texte. Votre nom apparaîtra dans le fichier de log lorsque vous ferez des commits et l'éditeur de texte vous servira, par exemple, à écrire des messages de commits. Voici quelques commandes pour modifier ce fichier (vous pouvez aussi éditer directement le fichier) :

- **git config --global user.name <votre nom>**
- **git config --global user.email <votre adresse mail>**
- **git config --global core.editor <votre éditeur de texte>**
- **git config color.ui auto** : permet d'avoir une coloration dans le terminal.

2 Initialiser un dépôt git : mettre CHIMERE sous git

La commande **git init** lancée dans un répertoire permet d'initialiser un dépôt git. Par défaut, git se place sur une branche appelé "master". Vous pouvez ensuite y ajouter des fichiers avec la commande **git add**. Ces fichiers seront suivi par git et leurs modifications pourront être enregistrées. Les fichiers de compilation (*.o), les fichiers temporaires (*∼), les exécutables (*.e) n'ont pas à figurer dans le dépôt, il n'est pas utile de garder une trace de leur évolution.

La commande **git status** permet de visualiser l'état actuel du dépôt :
- Fichiers modifiés mais non ajouté (ils ne seront pas commités s'ils ne sont pas ajoutés)
- Fichiers ajoutés / supprimés (ce qui sera commités si l'on fait un commit)
- Fichiers non suivi : fichiers dont l'on ne souhaite pas suivre l'évolution dans le dépôt mais qui sont présent dans le dossier (e.g. fichiers de compilations, fichiers exécutables)

La commande **git commit** permet de faire un commit, c'est à dire d'enregistrer les modifications ajoutées au dépôt (commande **git add**). Un message de commit sera demandé (il est possible d'utiliser l'option -m pour passer le message en paramètre). L'état des fichiers tel qu'ils se trouvent actuellement sera donc sauvegardés et il sera donc possible de commencer à développer. L'état initial ainsi que chaque état commités sera accessible à tout moment.

A partir d'une version de CHIMERE (`/data/PLT6/pub/rbriant/git/exercice_1/trunk`), initialisez un dépôt git avec **git init**. Ajoutez l'ensemble des fichiers à suivre (fichiers *.F90, *.sh, Makefile, *.sed). Commitez ces fichiers en spécifiant un message de commit (e.g. version initiale).

Les commandes précédentes sont utile pour créer un dépôt git à partir de zéro. Si l'on veux contribuer à un dépôt déjà existant on utilisera la commande **git clone <adresse du depot>**. La commande va copier le dépôt présent à l'adresse indiqué dans le répertoire courant.

3 Votre premier *commit* avec git

Une itération de développement classique se déroule comme suit :
- Modification du code
- Ajout / suppression des fichiers au dépôt avec **git add / git rm**
- Enregistrement des modifications en faisant un commit avec **git commit**

3.1 But de l'exercice CHIMERE

Le but de l'exercice est de rajouter le calcul de *Aerosol Optical Depth* dans le noyau de CHIMERE et son écriture dans le fichier *out...nc*. Il faudra :

1. Rajouter le module de calcul d'AOD dans *src/modules/aod.F90*. Le fichier se trouve à l'adresse : `/data/PLT6/pub/rbriant/git/exercice_1/aodf.F90`
2. Modifier le Makefile dans *src/modules* en ajoutant aodf.F90.
3. Modifier *src/initio/outprint.F90* et *src/model/prep_outprint.F90* pour ajouter le calcul et l'écriture de la nouvelle variable aot(:, :).

3.2 Quelques commandes utiles

Note 1 : toutes les versions dans git sont identifiées par un numéro unique appelé <SHA1> (*Secure Hash Algorithm*) qui ressemble à quelque chose comme dbe2a836720b7a7597ef399e70dc1f9a1cd32941.

Note 2 : quand vous utilisez plusieurs branches, celle par défaut (la première) s'appele *master*.

Note 3 : dans chaque branche la dernière version (le dernier *commit*) s'appele *HEAD*.

- **gitk** (ou **gitk --all** pour voire toutes les branches) : interface graphique qui permet de visualiser l'arbre de développement (branches et commits). Pour chacun des commits on peut voire l'identifiant du commit (<SHA1>), la liste des fichiers modifiés ainsi que les modifications effectuées dans chacun des fichiers.

- **git branch <nom de la nouvelle branche> <SHA1 ou nom de la branche racine de la nouvelle branche>** : Créé une nouvelle branche à partir d'un commit présent de l'arbre de développement.

- **git checkout <nom de la nouvelle branche>** : Changer la branche active. Les modifications suivantes seront appliquées à cette nouvelle branche. En jargon git ca s'appele *placer la tête (HEAD) sur la nouvelle branche*. Attention ! Toutes les modifications en cours doivent être commitées avant de changer de branche !

- **git checkout -b <nom de la nouvelle branche> <SHA1 ou nom de la branche racine de la nouvelle branche>**. Cette commande permet de faire l'équivalent des deux commandes précédentes en une seule.

- **git diff** : permet de visualiser les modifications effectuées par rapport à dernière version commité (HEAD).

– **git diff <SHA1> <autre SHA1>** : permet de visualiser les différences entre les deux commits données en paramètre (fonctionne aussi avec des branches).

– **git commit --amend** : permet de modifier le dernier commit avec les modifications en cours. Cela permet aussi de modifier le message du dernier commit.

– **git blame <nom de fichier>** : permet de visualiser qui a modifié chacune des ligne d'un fichier pour la dernière fois. La commande affiche aussi la date et le SHA-1 concerné.

– **git cherry-pick <SHA1>** : Appliquer à la branche active les modifications du commit <SHA1> (c.a.d. les différences entre la version <SHA1> et la précédente).

Exercice 1

Utilisez les commandes ci-dessus pour l'exercice suivant (n'hésitez pas à utiliser **gitk --all** entre chaque commande pour visualiser ce qui à changé) :

1. Créez une nouvelle branche appelée : "dev".
2. Faites les modifications indiquées dans section 3.1, une par une. Pour l'étape 3, insérez dans *src/model/prep_outprint.F90* (à l'endroit balisé "git TUTORIAL") :

   ```
   call aodf('dust', conc, ifam, nfam, nelem, aod)
   call mpi_send(aod(1:nzonal,1:nmerid), nzonal*nmerid, mpi_real, &
    0, ias_aod, mpi_comm_world, ierr)
   ```

   et dans *src/model/outprint.F90* :

   ```
   call mpi_recv(aod(1:nzonal,1:nmerid), nzonal*nmerid, mpi_real, &
      0, iar_aod, mpi_comm_world,ierr)
   ncstat = nf90_put_var( out_ncid, aot_varid, aod(:,:), &
      (/1,1,iprint+1/), (/nzonal,nmerid,1/) )
   ```

3. Faites *commit* après chaque modification. Spécifiez un message claire pour chaque *commit*. Utilisez **git diff** avant de faire le **git add** afin de visualiser les différences.
4. Vous vous êtes trompé dans le message du dernier *commit* ou vous voulez le rendre plus claire pour vos chers collègues. Corrigez le message, *sans faire de nouveau commit*.
5. Entre temps vous vous souvenez que depuis le week-end dernier vous rêvez de tester l'impact de la constante Von Kármán sur la saltation de dust. Comment satisfaire ce désir passionnant, sans créer de confusion avec les modifications liées à l'epaisseur optique ? Retournez sur la branche *master*. Ensuite modifiez

le fichier *src/modules/chimere_consts.F90*, ligne 29 : vkarm = 4.0d-1 au lieu de 4.1d-1.

6. Vous êtes content(e) du test de la constante Von Kármán et voulez maintenant récupérer vos modifications de l'epaisseur optique. Après avoir commité vos modifications sur la constante de Von Kármán, utilisez **git cherry-pick** pour les récupérer de la branch "dev".

4 Utiliser git avec un dépôt distant pour CHIMERE

4.1 Dépôt distant simple

Les commandes présentées précédemment permettent de gérer un dépôt local dans lequel le développeur possède tous les droits en lecture et en écriture. Il est possible d'y associer des dépôts distants. Ces dépôts distants peuvent être accessible en lecture uniquement (équivalent du trunk de svn). Une manière de travailler est d'avoir un dépôt distant par développeur (en plus des dépôts locaux), lui permettant de publier à tous ses commits, et un dépôt distant root qui sera géré par un administrateur. La figure 1 montre la séquence générale des événements pour utiliser git avec un dépôt distant simple.

Pour gérer un dépôt distant il y a quelques commandes supplémentaires à connaître :

- **git fetch** : permet de mettre à jour l'historique de ses branches distantes.
- **git pull <nom du serveur> <nom de la- branche distante>** : permet de récupérer les commits d'une branche distante sur une branche locale
- **git push <nom du serveur> <nom de la branche locale>:<nom de la branche distante>** : permet de pousser ses commits locaux sur une branche distante
- **git remote add <nom du serveur> <adresse du dépôt>** : permet de suivre une nouvelle branche distante.

Exercice 2 : Que se passe t-il quand deux personnes modifient le même fichier ?

Vous trouverez à l'adresse :
`/data/PLT6/pub/rbriant/git/exercice_2_depot_distant/depot_distant`
un répertoire git contenant un fichier chimere.par. Vous allez maintenant récupérer une version locale de ce dépôt distant, y faire plusieurs commits (localement) que vous pousserez à chaque fois vers le dépôt distant. Il faudra à chaque fois récupérer les modifications de vos voisins, qui poussent dans le même dépôt distant et résoudre les conflits.

[1] http://git-scm.com/book/en/Distributed-Git-Contributing-to-a-Project

1. Dans un nouveau répertoire, récupérez le dépôt distant en utilisant :
   git clone /data/PLT6/pub/rbriant/git/exercice_2_depot_distant/depot_distant <**mon depot local**>
2. Visualiser votre branche distante (origin/master) et la branche root (depot_root/master) avec gitk.
3. Modifier le fichier **chimere.par** et ajouter sur la première ligne :
   #commit numero 1 fait par <votre nom>
4. Faites le commit.
5. Poussez ensuite ce commits vers le serveur origin (nom par défaut du serveur) de votre branche locale (master) vers votre branche distante (master) avec la commande :
   git push origin master:**master**
   Si quelqu'un a fait une modification dans le dépôt distant, git refusera d'envoyez vos modifications au dépôt et vous demandera de récupérer ces modifications d'abord. Pour cela utilisez **git pull** (les instructions apparaissent à l'écran), puis résolvez les conflits en éditant les fichiers concernés. Refaites ensuite le **git push origin master**:**master**.
6. Utilisez gitk pour vérifier que votre commit a bien été poussé.
7. Recommencez, plusieurs fois en faisant un autre commit et en le poussant sur le dépôt distant.

4.2 Dépôts distants multiples

De nombreux groupes passent de svn à git en raison de la possibilité d'avoir plusieurs équipes travaillant en parallèle, en fusionnant les différents axes de travail dans le processus. La capacité des sous-groupes à collaborer via des dépôts distants sans devoir impliquer ou entraver toute l'équipe est un avantage énorme de git. La figure 3 montre comment s'organise les relations entrent les différents dépôts.

Exercice 3

Vous trouverez à l'adresse :
/data/PLT6/pub/rbriant/git/exercice_3_depot_distant/depot_root
un répertoire git contenant un fichier chimere.par. Ce dépôt est l'équivalent du trunk de svn et uniquement l'administrateur du dépôt peut y faire des commits. Vous allez maintenant vous créer un dépôt distant sur le serveur (ici PLT6) ainsi qu'un dépôt local sur votre compte. Vous allez y faire 3 commits (local) et les pousser vers votre dépôt distant. Enfin vous allez récupérer les commits de vos voisins à partir de leurs dépôts distants. N'hésitez pas à utiliser **gitk –all** entre chaque commande pour vérifier que vous obtenez la même chose que la Figure 3.

1. Créez votre dépôt distant sur le serveur à partir du dépôt root :
   git clone /data/PLT6/pub/rbriant/git/exercice_3_depot_distant/depot_root /data/PLT6/pub/rbriant/git/exercice_3_depot_distant/depot_depot_distant_<votrenom>
   Votre dépôt distant est publique, vous y êtes l'administrateur mais les autres développeurs y auront un accès en lecture. Vous pouvez y publiez les commits à intégrer au dépôt central (par l'administrateur) ou des commits temporaire à partager avec d'autres développeurs.
2. Dans un nouveau répertoire (sur votre compte), clonez votre dépôt distant pour créer votre dépôt local en utilisant :
   git clone /data/PLT6/pub/rbriant/git/exercice_3_depot_distant/depot_depot_distant_<votrenom> <**mon depot local**>
   Votre dépôt local est privée, seulement vous y avez accès et vous y êtes l'administrateur. Si vous devez partager un commit avec quelqu'un, poussez les sur votre dépôt distant (Figure 3a).
3. Déplacez vous dans le dépôt et, afin de suivre le dépôt root, utilisez (Figure 3b) :
   git remote add -f depot_root /data/PLT6/pub/rbriant/git/exercice_3_depot_distant/depot_root
4. Modifiez le fichier **chimere.par** et ajoutez sur la première ligne :
   #commit numero 1 fait par <votre nom>
5. Faites le commit (Figure 3c).
6. Poussez ensuite ce commit vers le serveur (origin) de votre branche locale (master) vers votre branche distante (master) avec la commande (Figure 3d) :
   git push
7. Pour suivre le dépôt distant de quelqu'un d'autre utilisez la commande :
   git remote add depot_distant_<**NOM**> /data/PLT6/pub/rbriant/git/exercice_3_depot_distant/depot_depot_distant_<nom>. Vous devez ensuite mettre à jour vos branches distantes, avec la commande **git fetch** –**all**, pour la faire apparaître en faisant gitk (Figure 3e).
8. Enfin, rapatriez les commits de vos voisins sur votre branche locale en utilisant la commande (Figure 3f) :
   git pull depot_distant_<**NOM**> **master**
   Si les commits de vos voisins rentrent en conflits avec les votre, git vous demandera de résoudre ces conflits en éditant les fichiers concernés. Lancez ensuite les commandes **git add** et **git commit**.
9. Utilisez ensuite **git push** pour poussez ces nouveaux commits vers votre dépôt distant (Figure 3g).
10. Recommencez en faisant d'autres commits, en les poussant vers votre dépôt distant et en récupérant les commits des autres.

Conclusion

Git est un outil très performant qui demande un effort d'adaptation pour le maîtriser, mais qui permet de développer en équipe efficacement. Voici quelques conseils utiles pour le développement :

- **Ne pas développer dans sa branche master** afin de pouvoir garder sa branche master synchronisée avec le dépôt central, cela permet d'éviter les conflits lors de **git pull**.
- **Faire plusieurs petits commits plutôt qu'un seul gros**, c'est plus facile à débuguer car on sait immédiatement à quoi correspond telle ou telle modification dans le code.
- Ne pas hésiter à faire un commit local pour faire une sauvegarde d'une version en cours. Les commits locaux n'apparaissent pas sur le serveur. Les commits qui seront poussé sur le serveur doivent être uniquement les commits testé et validé.
- Git permet de pouvoir facilement revenir en arrière. C'est un atout qu'il faut utiliser pour valider vos commit.
- Ne pas hésiter à utiliser les commandes : **git status**, **git diff**, **gitk --all**, **git log**, **git branch** à tout moment pour savoir sur quelle branche on se trouve, quel fichier à été modifié...
- Garder une fenêtre gitk ouverte lorsque l'on fait un rebase. Si l'on fait une erreur, la fenêtre gitk permettra de retrouver les SHA-1 des commits originaux et donc de restaurer une ancienne version (sinon il y a toujours la possibilité d'utiliser **git reflog**).
- Ne garder dans un commit que les modifications nécessaires (pas de modification d'espace, de saut de ligne, de tabulation). Cela permet de visualiser plus facilement ce qui a été modifié dans le commit. Si des modifications "esthétiques" sont nécessaire, il est préférable de faire un commit à part, contenant uniquement cela.

A Liens utiles :

- http://git-scm.com/
- http://try.github.io/levels/1/challenges/1
- http://git-lectures.github.io/

B Modifier l'historique avec git rebase

La commande git rebase permet de modifier l'historique. Concrètement, git va se placer à l'endroit de l'arbre de développement où l'on souhaite modifier quelque chose et nous permettra de faire des modifications. Ensuite git va réappliquer automatiquement les commits suivant un à un. Si il y a des conflits, git s'arrêtera et nous proposera de les résoudre. Voici comment l'utiliser :

Lancer **git rebase -i <SHA-1 du commit précédent celui que l'on souhaite modifier>**. Une fenêtre va alors s'ouvrir où l'on pourra visualiser tous les commits suivant

celui qui est passé en paramètre. Pour chacun de ces commit il faut choisir (en remplaçant dans le fichier) :

- "pick" : le commit sera appliqué sans possibilité de modification.
- "edit" : le commit sera appliqué et l'on pourra le modifier avant que le commit suivant soit appliquer.
- "squash" : le commit sera appliqué et fusionné avec le précédent. Le message de commits pourra alors être modifié.

Lorsque le choix des opérations à effectuer est terminé, il faut sauvegarder le fichier et le fermer. Git va alors se placer au niveau du premier commits à et réappliquer un à un les commits suivant en s'arrêtant quand un commit doit être modifié. Des instructions s'afficheront à l'écran à chaque étape (**git rebase --continue**, **git add**, **git commit**...). Il est possible d'abandonner le rebase en lançant la commande : **git rebase --abort**.

La commande **git rebase** est un atout de git par rapport à svn. En effet, avec svn il n'est en aucun cas possible de modifier l'historique. Modifier l'historique est dangereux, et il faut faire très attention et être sur de ce que l'on souhaite faire. Par exemple, modifier l'historique peut parfois créer des bug dans des versions qui marchaient auparavant car les modifications que l'on va faire se répercuteront dans les commits suivant. Il n'est donc pas conseillé de le faire dans le dépôt central. En revanche cela peut s'avérer très pratique localement, lorsque l'on souhaite "nettoyer" un commit (e.g. modifier un message de commit, enlever les print, les saut de ligne inutile).

Utiliser la commande **git rebase** pour éditer le "commit 1" de la Section 3 (faire une nouvelle modification dans un fichier par exemple) et fusionner les commit 2 et 3. Vérifier ensuite que les modifications ont bien été prises en compte avec **gitk --all**.

C Git cheat sheet

Git Cheat Sheet

Overview

When you first setup Git, set up your user name and email address so your first commits record them properly.

```
git config --global user.name "My Name"
git config --global user.email "user@email.com"
```

About Git, GitHub and Heroku.
Git is a free & open source, distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

GitHub is the best way to collaborate around your code. Fork, send pull requests and manage all your public and private git repositories.

Heroku is a cloud application platform that supports a number of different programming languages including Java, Ruby, Node.js, and Clojure - it's a new way of building and deploying web apps.

Basic Git Workflow Example

Initialize a new git repository, then stage all the files in the directory and finally commit the initial snapshot.

```
$ git init
$ git add .
$ git commit -m 'initial commit'
```

Create a new branch named featureA, then check it out so it is the active branch. then edit and stage some files and finally commit the new snapshot.

```
$ git branch featureA
$ git checkout featureA
$ (edit files)
$ git add (files)
$ git commit -m 'add feature A'
```

Switch back to the master branch, reverting the featureA changes you just made, then edit some files and commit your new changes directly in the master branch context.

```
$ git checkout master
$ (edit files)
$ git commit -a -m 'change files'
```

Merge the featureA changes into the master branch context, combining all your work. Finally delete the featureA branch.

```
$ git merge featureA
$ git branch -d featureA
```

Setup & Init

Git configuration, and repository initialization & cloning.

`git config [key] [value]`	set a config value in this repository
`git config --global [key] [value]`	set a config value globally for this user
`git init`	initialize an existing directory as a Git repository
`git clone [url]`	clone a Git repository from a URL
`git help [command]`	get help on any Git command

Stage & Snapshot

Working with snapshots and the Git staging area.

`git status`	show the status of what is staged for your next commit and what is modified in your working directory
`git add [file]`	add a file as it looks now to your next commit (stage)
`git reset [file]`	reset the staging area for a file so the change is not in your next commit (unstage)
`git diff`	diff of what is changed but not staged
`git diff --staged`	diff of what is staged but not yet committed
`git commit`	commit your staged content as a new commit snapshot
`git rm [file]`	remove a file from your working directory and unstage
`git gui`	tcl/tk GUI program to make all of these commands simpler

Branch & Merge

Working with Git branches and the stash.

`git branch`	list your branches. a * will appear next to the currently active branch
`git branch [branch-name]`	create a new branch at the current commit
`git checkout [branch]`	switch to another branch and check it out into your working directory
`git checkout -b [branch]`	create a branch and immediately switch to it
`git merge [branch]`	merge another branch into your currently active one and record the merge as a commit
`git log`	show commit logs
`git stash`	stash away the currently uncommitted modifications in your working directory temporarily
`git stash apply`	re-apply the last stashed changes

Share & Update

Fetching, merging and working with updates from another repository.

`git remote add [alias] [url]`	add a git URL as an alias
`git fetch [alias]`	fetch down all the branches from that Git remote
`git merge [alias]/[branch]`	merge a branch on the server into your currently active branch to bring it up to date
`git push [alias] [branch]`	push the work on your branch to update that branch on the remote git repository
`git pull`	fetch from the URL tracked by the current branch and immediately try to merge in the tracked branch

Inspect & Compare

Examining logs, diffs and object information.

`git log`	show the commit history for the currently active branch
`git log branchB..branchA`	show the commits on branchA that are not on branchB
`git log --follow [file]`	show the commits that changed file, even across renames
`git diff branchB...branchA`	show the diff of what is in branchA that is not in branchB
`git show [SHA]`	show any object in Git in human-readable format
`gitx`	tcl/tk program to show the commit log in a GUI

Contributing on GitHub

To contribute to a project that is hosted on GitHub you can fork the project on github.com, then clone your fork locally, make a change, push back to GitHub and then send a pull request, which will email the maintainer.

fork project on github

```
$ git clone https://github.com/my-user/project
$ cd project
$ (edit files)
$ git add (files)
$ git commit -m 'Explain what I changed'
$ git push origin master
```

go to github and click ‘pull request’ button

Deploying to Heroku with Git

Use the heroku command-line tool to create an application and git remote:

```
$ heroku create
```

```
[Creating glowing-dusk-965... done, stack is bamboo-mri-1.9.2
http://glowing-dusk-965.heroku.com/ <http://glowing-dusk-965.
heroku.com/> | git@heroku.com:glowing-dusk-965.git <x-msg://536/
git@heroku.com:glowing-dusk-965.git> Git remote heroku added]
```

Use git to deploy the application.

```
$ git push heroku master
```

Create an additional Heroku app for staging, and name the git remote "staging".

```
$ heroku create my-staging-app --remote staging
```

Use git to deploy the application via the staging remote.

```
$ git push staging master
```

github
SOCIAL CODING

http://github.com

heroku™

http://heroku.com

082011

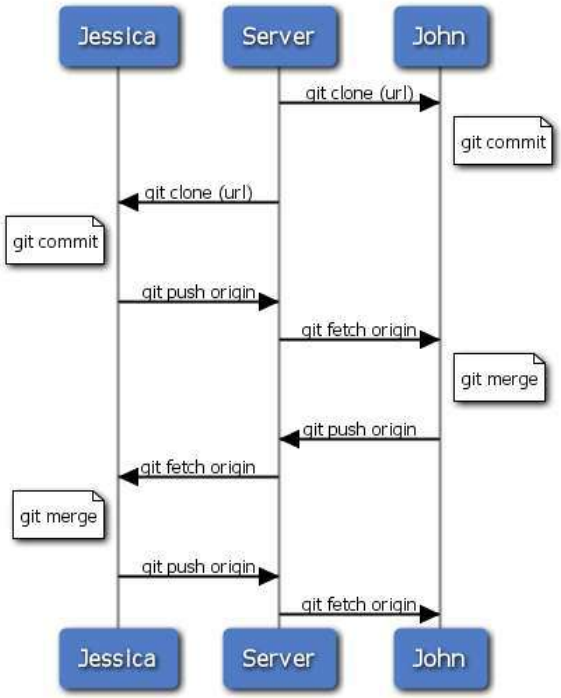
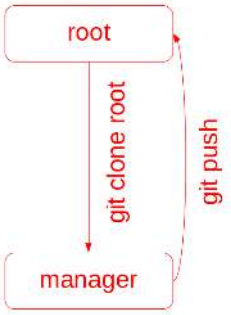
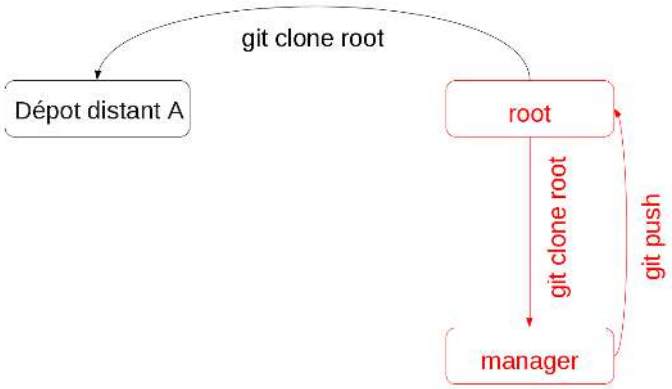


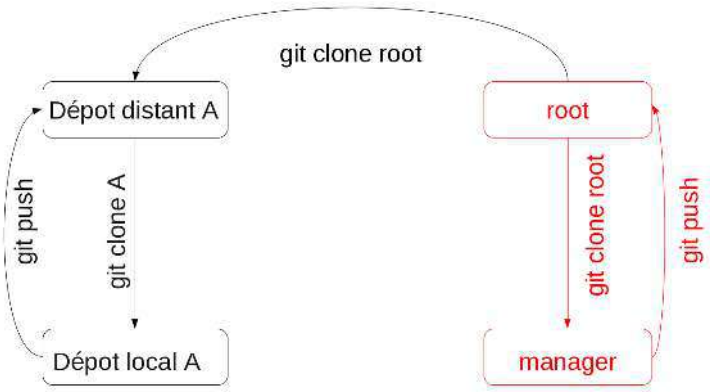
FIG. 1 – *General sequence of events for a simple multiple-developer Git workflow* [1]



(a) Step 1 : Cloning git repository from **root** to manager′s local

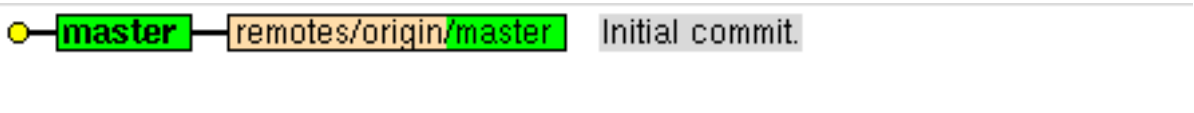


(b) Step 2 : Cloning git repository from **root** to user A′s remote repository

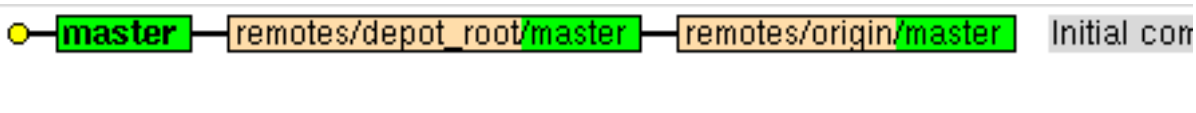


(c) Step 3 : Cloning git repository from user A′s remote repository to user A′s local repository

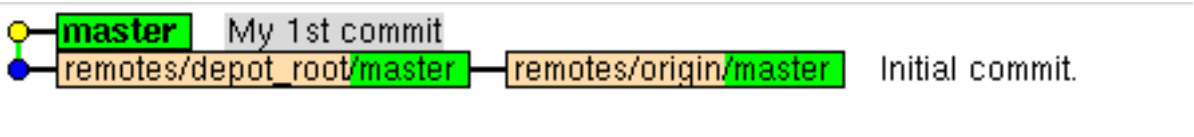
FIG. 2 – *Schéma pour le flux de travail avec les dépôts distants multiples.*



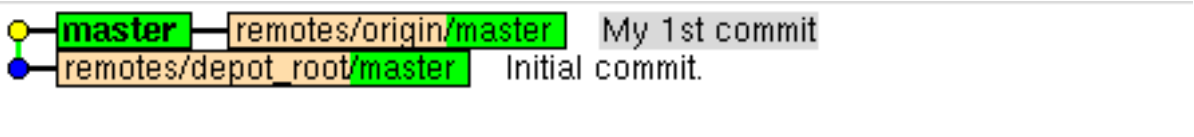
(a) Step 2 : **git clone**. Vous avez récupéré la copie du dépôt distant. Vous avez maintenant votre *branche locale* (**master**) avec des fichiers sur votre disque (en couleur verte sur le dessin), ainsi qu'un *lien* vers le dépôt distant **remotes/origin/master** (en marron). **origin** est le nom du dépôt distant par défaut, et **master** est sa branche par défaut. **master** et **remotes/origin/master** sont au même niveau (connectés avec une ligne horizontale), c.a.d leur contenus sont identiques.



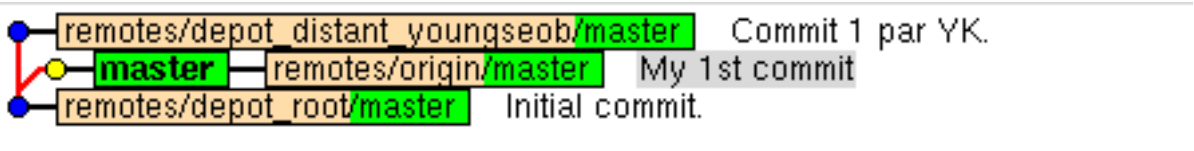
(b) Step 3 : **git remote add depot_root**. Il existe un dépôt distant qui s'appelle **root**. C'est un équivalent de **trunk** de svn. La commande que vous venez de taper vous permet de suivre l'évolution de ce dépôt avec ***gitk***. Sur ce schéma le dépôt **root** est au même niveau que **remotes/origin/master** (= ils sont identiques), car il n'y a pas eu de modifications après leur dernière synchronisation.



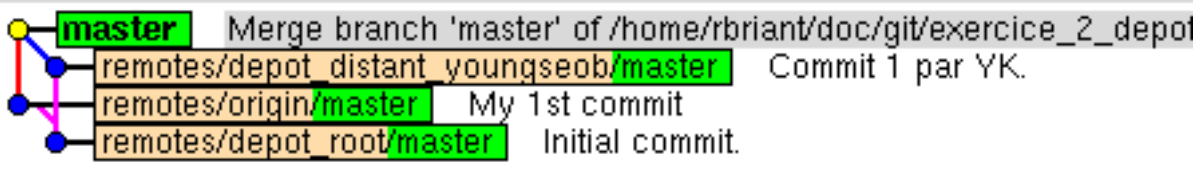
(c) Step 5 : **git commit** : vous avez commité des modifications dans votre branche **master** locale. La branche se trouve donc en avant par rapport aux dépôts distants.



(d) Step 6 : **git push** : vous synchronisez avec le dépôt distant en le mettant à jour par rapport à votre branche **master** locale. Les deux se trouvent maintenant au meme niveau et a l'avance par rapport au **depot_root**.

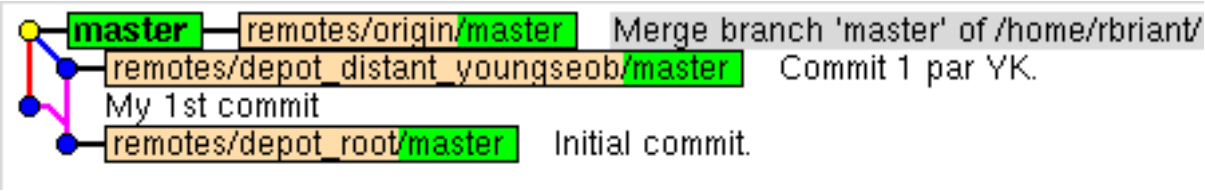


(e) Step 7 : **git fetch**. Qui se passe-t-il dans les dépôts ? Cette commande permet de se mettre à jour par rapport aux dépôts distants associé avec la commande **git remote add**. Comme nous indique le schéma, il y a un nouveau dépôt qui apparaît avec gitk (**depot_distant_youngseob**). Ce dépôt est en avance par rapport au **depot_root**, lui aussi. Il y a donc trois versions différentes du code : la votre, celle de votre collègue, et le dépôt central **root**. Ça demande un *merge* !



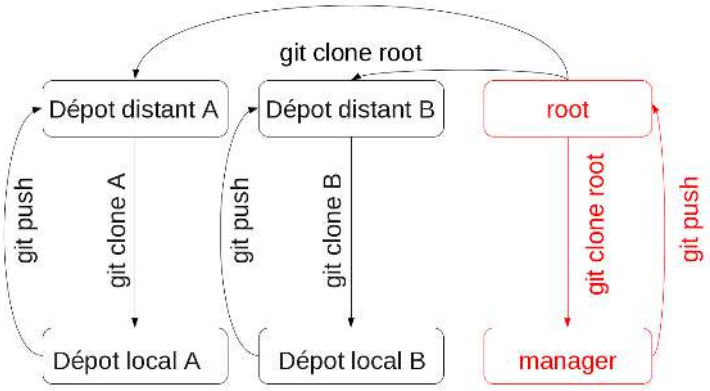
(f) Step 9 : Après avoir connecté le dépôt de votre collègue avec *git remote add*, il est temps de faire **git pull** pour mettre à jour *votre branche* par rapport à lui (= récupérer ses modifications). Cela peut provoquer des conflits si les mêmes endroits du code ont été modifiés. On résout les conflits et on termine notre **git pull**. Comme on le voit sur le schéma de ***gitk***, notre branche *master* est en avance par rapport à notre dépôt distant, car les modifications de votre collègue ont été appliquées localement.

FIG. 3 – *Snapshots de gitk pour le flux de travail avec les dépôts distants multiples.*

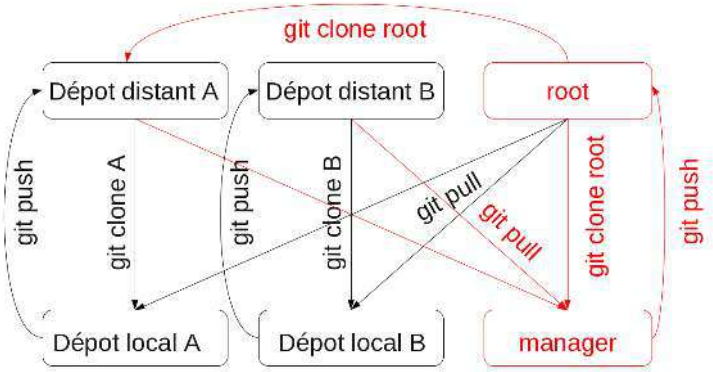


(g) Step 10 : **git push** : on va mettre à jour finalement le dépôt distant pour qu'il soit synchronisé avec la branche **master** locale. **remotes/origin/master** et **master** se trouvent au même niveau sur le schéma de ***gitk***.

FIG. 3 – suite.



(h) Step 4 : Cloning git repository for user B (repeat Steps 2 and 3)



(i) Step 5 : Sharing the users′ remote repositories between users A and B

FIG. 3 – *Schéma pour le flux de travail avec les dépôts distants multiples.*