Projet programmation 2
Introduction à Git

Mathieu Hilaire & Stefan Schwoon

8 février 2024

Table des matières

1 Introduction

Git est un utilitaire qui aidera à l'élaboration d'un projet collectif, que ce soit un projet de programmation ou un papier de recherche. Ici, un projet est composé d'un ensemble de fichiers qui évoluent. Dans ce contexte, Git remplit deux fonctions :

— *gestion de versions :* permet de garder l'historique du développement du projet, on peut facilement revenir à n'importe quelle version cohérente du projet et en gérer plusieurs branches (p.ex. version en production, développement de longue haleine, bugfix urgent)

— *développement collaboratif :* plusieurs personnes peuvent coopérer et échanger leurs contributions, en travaillant sur de différentes branches

Ce document a pour objectif de vous mettre en mésure d'utiliser les fonctions les plus importantes de Git. Pour d'autres sources d'information, voir la Section 1.2. La Section 2 traitera l'aspect "gestion de versions". La Section 3 traitera l'aspect "développement collaboratif".

1.1 Installation

Git est déjà installé sur les machines de la salle 411. Sur vos machines personnelles, il convient d'installer le paquet `git-all`, avec une commande de la forme `sudo apt install git-all` ou similaire, selon votre système.

Pour tester si Git est bien installé, il convient de lancer la commande suivante dans un terminal :

```
git --version
```

Au cœur, Git est une collection d'outils sur la ligne de commande, et ce document se concentre là-dessus. Il existe certains outils graphiques pour ceux qui préfèrent ce mode d'interaction, mais c'est avec la ligne de commande qu'on obtient le contrôle le plus précis.

1.2 Documentation

Une version électronique du livre *Pro Git* se trouve sur la page web de Git, sous l'URL suivant : `https://git-scm.com/book/en/v2` (en Anglais). Il est fortement conseillé de suivre ce bouquin pour un traitement plus approfondi.

D'ailleurs, la syntaxe des commandes est expliqué dans le manuel électronique (*pages man*). Généralement, la syntaxe des commandes est de la forme `git` ⟨*cmd*⟩ ⟨*params*⟩, où *cmd* est la commande principale, telle que `init`, commit etc. Il existe une page man par commande sous le nom de `git-init`, `git-commit`, etc.

1.3 Configuration

Avant de faire les premiers pas, il est fortement conseillé de configurer votre nom et mail qui seront utilisés pour identifier vos contributions dans un projet, ainsi que votre éditeur de choix (tel que `emacs`, `vi`, etc) :

```
git config --global user.name "Jean Dupont"
git config --global user.email dupont@ens-cachan.fr
git config --global core.editor vi
```

Ici, le mot clé `-global` veut dire que cette configuration s'applique à tous vos projets.

Pour ce document, il est aussi recommandé de rajouter la commande suivante qui crée un raccourci pour observer l'historique de notre projet :

```
git config --global alias.view "log --graph --oneline --decorate --all"
```

L'ensemble de votre configuration s'affiche avec

```
git config -l
```

2 Gestion de versions

Dans cette section, nous allons faire les premiers pas avec Git et étudier comment on s'en sert pour gérer un projet avec un seul développeur. L'aspect collaboratif sera traité dans la Section 3.

2.1 Les commits

Git gère les fichiers associés avec un projet, ainsi que toute leur histoire, dans un *dépôt*. De temps en temps, le programmeur soumet un *commit*, c'est à dire qu'il sauvegarde l'état actuel de ses fichiers dans le dépôt. Le dépôt contient un arbre acyclique de commits, supportant ainsi un développement non-linéaire

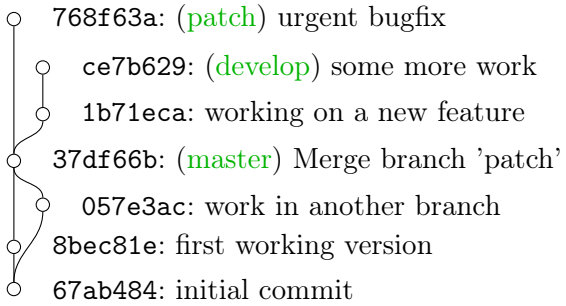

Figure 1 – Exemple de l'histoire d'un dépôt

Figure 1 donne un exemple du dépôt qui raconte son histoire, avec les commit les plus récents en haut. Supposons qu'il s'agit d'une application web.

- Le code actuellement en production est représenté par le commit dénommé `master`.
- Le programmeur a commencé à rajouter quelques fonctionnalités qui se trouvent dans la branche `develop`.
- Or, le programmeur se voyait contraint d'interrompre son travail sur ce développement par la découverte d'un problème dans la version en production qui nécessite son attention immédiate. Le programmeur est donc revenu sur `master`, et il a créé une nouvelle branche `patch` pour résoudre ce problème.
- Une fois que le patch fonctionne, la branche `master` sera fusionnée avec `patch`, et le programmeur peut continuer son travail sur `develop`. La partie basse montre que dans le passé, `master` a été par une telle fusion de branches.

Tout commit est représenté par une chaîne de 40 chiffres hexadécimaux (en effet, une somme de contrôle du type SHA-1 sur tout le contenu), mais souvent il suffit de n'utiliser que les sept premiers chiffres pour identifier un commit. En plus, un commit peut porter un ou plusieurs noms plus lisibles choisis par l'utilisateur (comme `master`, `develop` ou `patch`).

Commençons maintenant les premier pas en créant un nouveau dépôt dans un dossier vide :

```
mkdir test-git
cd test-git
git init
ls -la
```

La dernière command devrait afficher un dossier "caché" dénommé `.git`. Ceci contient en effet toutes les données qu'utilise Git pour gérer le dépôt. N'y touchez pas !

Le nouveau dépôt créé ainsi est dans un état un peu spécial : son graph de commits est entièrement vide. On va donc faire un premier commit. Créez un fichier `toto` avec quelques lignes de texte, et ensuite :

```
git add toto
git commit -m "mon premier commit"
git view
```

Voilà, notre dépôt désormais contient un commit comme le confirme la dernière commande (en fait, un alias qu'on a créé dans la Section 1.3) qui devrait produire à peu près le suivant :

```
* 1d6f14b (HEAD -> master) mon premier commit
```

— Cette ligne contient l'identifiant (`1dff14b`), qui va être différent, en fonction du contenu de votre fichier.
— Votre commit est l'état actuel de la branche `master`. [1]
— `HEAD` signifie que la version de votre projet actuel est considéré comme une évolution de la branche `master`.

▶ Faites quelques commits en plus pour créer une histoire linéaire. (Éditer `toto` et répéter les commandes ci-dessus.) Rajoutez aussi un deuxième fichier. Observez l'évolution de `HEAD` et `master`.

Note : Dans `git commit`, on peut omettre la partie après `-m`, dans ce cas Git vous demandera de saisir un message dans votre éditeur de texte préféré (qui peut s'étendre sur plusieurs lignes si vous le souhaitez).

En pratique, quand est-ce qu'on soumet un commit ? Il n'y a pas de règle précise, mais voici quelques recommandations :
— On commit peut servir comme un copie de sauvegarde, du coup il est recommandé de les faire fréquemment.
— On peut revenir sur n'importe quel commit plus tard, du coup il est préférable que le code soit dans un état cohérent lors du commit.
— Le message après `-m` est censé décrire la contribution apportée par ce commit. Il est donc recommandé de travailler sur un seul objectif bien délimité à la fois et les inclure dans un commit dès que cet objectif est atteint.

2.2 L'espace de travail

Que signifient les lignes `git add toto` et `git commit` précisement ? Rappelons qu'un projet consiste de plusieurs fichiers, et que Git a initialement crée un dossier `.git` qui contient l'historique du dépôt. Le reste en dehors de `.git` s'appelle *espace de travail* (working directory). Les fichiers et dossiers là-dedans peuvent être dans quatre états différents :
— *non versionné / untracked* : un fichier n'étant pas géré par Git ;
— *non modifié / unmodified* : la version actuelle du fichier est identique à celle sauvegarde dans le dernier commit ;
— *modifié / modified* : le fichier a subi des modifications depuis le dernier commit ;
— *indexé / staged* : idem, mais le fichier a été retenu pour être inclus dans le prochain commit.

Ce dernier état intervient typiquement entre une modification et un commit, il permet de sélectionner les fichiers qu'on souhaite inclure dans un commit. Dans ce contexte `add` rajoute `toto` dans l'index, et `commit` sauvegarde le commit.

L'état actuel des fichier peut être inspecté à tout moment avec

```
git status
```

On regardera les états dans le détail :

1. Le nom `master` n'a aucune connotation particulière, c'est simplement le nom que Git choisit normalement pour la toute première branche, mais c'est une branche comme toute les autres.

Fichiers non versionnés Lorsqu'on crée un nouveau fichier, il est à priori non versionné.

— On utilise `git add` pour le rajouter au projet, il devient alors *indexé*. On peut aussi rajouter un dossier entier.

— Si on ne souhaite pas inclure le fichier dans le projet (p.ex. des fichiers automatiquement généré par le compilateur, genre `.class` ou `.o`, il convient de le mettre dans la liste de fichiers à ignorer. Une telle liste peut être mise dans un fichier dénomme `.gitignore` (dans le dossier du dépôt ou dans votre dossier 'home'). En voici un exemple :

```
*.class
*.o
.gitignore
```

▶ Créer des nouveau fichiers, les rajouter au projet ou à la liste à ignorer, et observer l'évolution de `git status`.

Remarque : L'espace de travail est considéré comme *propre* si tous ses fichiers sont soit non modifiés, soit ignorés.

Fichiers non modifiés Revenons sur les fichier `toto` déjà sauvegardé dans le projet dans la Section 2.1.

— Si on modifié le fichier, il devient *modifié* (quelle surprise !). Notons que la distinction entre un fichier modifié et non modifié se fait sur son contenu (ou plutôt sa somme de contrôle) et non pas sur la date de sa dernière modification.

— La commande `git rm toto` détruit le fichier `toto`, et il sera exclu du projet lors du prochain commit. (Ses anciennes versions restent bien sûr dans le dépôt.)

▶ Sauvegarder un fichier versionné sans et avec modification, et utiliser `git rm`, tout en observant `git status`.

Fichiers modifiés Ayant modifié quelques états à satisfaction, on prépare le commit.

— La commande `git add toto` rajoute `toto` à l'index. On n'est pas obligé à y inclure toutes les fichiers modifiés !

— Attention, `git add` rajoute *le contenu actuel* du fichier à l'index. D'éventuelles modifications apportées entre `add` et `commit` ne seront pas inclus dans le commit. (En réalité, la tetrachotomie (oui ce mot est dans le dictionnaire !) d'états n'est donc pas stricte, un fichier peut être à la fois modifié et indexé.)

— Si on n'est pas content de ses modifications, on peut revenir sur l'état du fichier dans le commit actuel, avec `git checkout - toto`.

— Pour abandonner toutes les modifications : `git reset -hard`

▶ Utilisez ces commandes, et comme toujours observez `git status`.

Fichiers indexés Comme l'on a déjà dit, l'index sert pour sélectionner les fichiers que l'on souhaite soumettre dans le prochain commit.

— `git commit` sert à produire un nouveau commit à partir des fichiers indexés. Git vous demandera un message décrivant les modifications qui peut être composé dans votre éditeur préféré ou avec `-m`. Dans un travail collectif, il convient d'agréer quelques conventions sur l'information à inclure dans un tel message.

— Avec l'option `-a`, Git rajoute automatiquement tous les fichiers modifiés à l'index avant de les soumettre.

— Vous avez oublié un fichier dans votre dernier commit ou vous souhaitez améliorer le message ? Pas de problème, il y a `git commit -amend`. (Mais attention à ça si vous travaillez dans un projet collaboratif, voir en Section 3.4.)
— Si jamais on souhaite expulser un fichier de l'index, la commande suivante convient : `git reset HEAD toto`

▶ Exercez ces commandes pour vous y habituer.

Voilà, vous savez maintenant comment gérer une histoire linéaire de commits. Dernier conseil : Si vous trouvez la syntaxe de certains commandes trop obscure, vous pouvez créer des *alias* (voir Section 1.3) (un alias étant tout simplement une substitution de texte).

2.3 Inspecter l'historique

La commande `git log` est un outil assez puissant pour inspecter le graphe de commits. La page man pour `git-log` en donne une impression. En fait, `git view` est un alias qu'on vient de créer qui en regroupe certains options.

Sans entrer dans tous les détails, `git log` sert à afficher l'histoire de la branche actuelle ou de toutes les branches (avec `-all`). Plusieurs options règlent le format d'affichage et les commits à y inclure.

▶ Utilisez `git log` sans arguments. Comment limiter l'affichage aux trois derniers commits ?

Revenons sur la Figure 1 qui contient un commit avec l'identifiant `1b71eca`. La commande `git show 1b71eca` affichera toutes les informations sur ce commit : la date de soumission, l'identité du contributeur (voilà pourquoi on a configuré notre nom et mail), la liste des fichiers et le détail des modifications.

Finalement, on peut comparer n'importe quelle paire de commits avec `git diff`, p.ex. :
— `git diff` (tout seul) affiche les modifications apportées aux fichiers modifiés (i.e. pas encore indexés).
— `git diff 1b71eca` affiche les différences entre la version actuelle et le commit `1b71eca`.
— `git diff 8bec81e 1b71eca` compare les deux commits donnéés.

2.4 Les branches

On va maintenant étudier l'un des aspects les plus puissants de Git, la possibilité de développer un projet de façon non-linéaire. Dans un projet non-trivial il convient de garder une branche qui contient la version stable/en production, typiquement on l'appelle `master`. Créer un branche supplémentaire peut s'avérer utile dans pas mal de circonstances, par exemple (voir aussi la Figure 1) :

— Vous souhaitez expérimenter avec une idée "folle" qui n'a peut-être pas vocation d'être retenu dans le projet final. Vous ouvrez donc une nouvelle branche qui contient le travail relatif à cette idée. Si jamais l'idée s'avère inutile, vous pouvez revenir sur la branche de départ.
— Vous embarquez sur un développement de longue haleine qui prendra un peu de temps (et plusieurs commits) avant d'être suffisamment complet et stable pour être inclus dans `master`. Ce développement se fait donc dans une branche différente.
— Vous devez interrompre votre travail sur un tel développement pour apporter une correction urgente à `master`. Vous créez donc une nouvelle branche depuis `master` qui ne contient que cette correction.
— Plusieurs développeurs travaillent en parallèle sur des aspects différents, chacun dans sa branche.

Comment alors créer une nouvelle branche ? C'est facile, on tape

```
git branch develop
```

où `develop` est un nom librement choisi. (Il convient d'avoir un espace de travail propre avant de le faire.) On vérifie (avec `git view`) que la branche a bien été créée.

Attention, même si `git branch` crée une nouvelle branche, on reste toujours dans l'ancienne branche. Pour basculer dans la nouvelle branche, il convient de dire

```
git checkout develop
```

Dans l'historique, `HEAD` devait désormais pointer vers `develop`.

▶ Basculez entre `master` et `develop` et faites au moins deux commits dans chacune des branches. Observez l'évolution de l'historique.

On peut combiner les deux opérations : La commande `git checkout -b patch` va créer une nouvelle branche dénommée `patch` et y basculer.

Désormais, on comprend mieux le fonctionnement de `HEAD` et comment Git identifie les commits :

— Comme l'on a déjà dit, chaque commit est identifiable par sa somme de contrôle hexadécimale.

— En plus, certains commits peuvent être étiquettés avec un nom tel que `master` etc (qui s'affiche en vert dans `git view`). On appelle ça une *branche*, il s'agit d'un pointer sur un commit qui évolue lorsqu'on ajoute des commits. [2]

— `HEAD` est un pointeur vers une telle branche. Disons que `HEAD` pointe vers `develop` qui lui pointe vers `abc1234`. Lorsqu'on crée le prochain commit (disons `def5678`, Git en fait un fils de `abc1234` et fait avancer `develop` à `def5678`.

`git checkout` marche aussi avec des commits qui ne sont pas pointé par une branche, p.ex. `git checkout abc1234`. Or, Git vous affiche un avertissement :

```
You are in 'detached HEAD' state. [...]
```

Ce n'est pas dangereux en tant que ça, mais il faut bien savoir ce qu'on fait à ce moment car `HEAD` ne pointe sur aucune branche. Si jamais on soumet un nouveau commit dans cet état, ceci n'est accessible depuis aucune branche et n'apparaîtra donc pas dans `git view`. D'ailleurs, Git considère un tel commit comme abandonné, il peut donc devenir victime des ramasse-miettes que Git conduit de temps en temps.

▶ Basculez vers un ancien commit et créez-y une nouvelle branche. Comment peut-on supprimer une branche ?

Autre que les branches, Git connaît un autre type d'identifiant, les *étiquettes* (tags). Celles-ci sont des pointeurs *fixes* qui identifient un commit précis. P.ex., ça sert servir pour identifier la version 1.0 d'un projet ou la version soumis pour une soutenance... La syntaxe en est

```
git tag v1.0
```

(où v1.0 est choisi librement). Ensemble, tous ces identifiants (hexadécimaux, branches, tags) sont appellés *références* (refs tout court en anglais).

2. Le terme *branche* est donc un peu trompeur – techniquement, une branche est simplement un pointeur sur un commit particulier, même si en pratique celui-ci est souvent une feuille dans le graphe des commits, et donc la pointe d'une branche.

2.5 Fusionner des branches

La fusion des branches permet de réunir les contributions de deux[3] branches dans un seul commit. Revenons par exemple sur le cas de Figure 1 : ayant complète le bugfix dans la branche `patch`, le programme veut l'intégrer à la fois dans la version de production `master` et dans `develop`.

La fusion n'est pas une opération tout à fait symmétrique : étant donné deux branches, `A` et `B`, l'opération suivante fait avancer la branche `A`, avec un nouveau commit qui intègre les contributions de `B`.

```
git checkout A
git merge B
```

Si `A` est un descendant de `B`, cela ne change rien. Si `A` est un ancêtre de `B`, alors on deplace le pointer de `A` vers `B`. Sinon, on cherche le plus récent ancêtre commun de `A` et `B`, disons `C`. On prend alors la différence de `C` et `B` et l'applique à `A`.

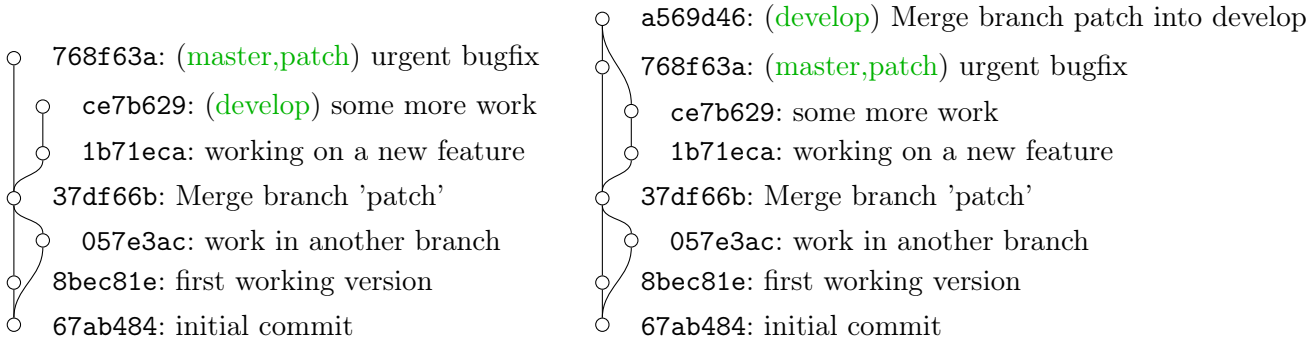

FIGURE 2 – À gauche : merge de `master` avec `patch`. A droite : merge de `develop` avec `patch`.

La Figure 2 illustre les deux cas intéressants, par rapport au graphe de la Figure 1 :
— `merge` est un ancêtre de `patch`, du coup la fusion fait avancer `merge` ;
— Le plus récent ancêtre commun de `develop` et `patch` est `37df66b`, du coup la fusion rajoute la différence entre celui-ce et `patch` dans un nouveau commit de `develop`. Ce commit est considéré comme ancêtre et de `patch` et de (l'ancien) `develop`.

Résoudre les conflits Dans le deuxième cas, si les deux branches en question ont travaillé sur des fichiers différentes ou sur des endroits différents dans un même fichier, GIT arrive normalement à bien fusionner les branches automatiquement. Si ce n'est pas le cas (p.ex. parce qu'une même ligne a subi deux modifications différentes), la commande `git merge` n'aboutit pas à créer une nouvelle version. Elle crée plutôt une nouvelle version de ce fichier qui a la forme suivante, en présentant les deux choix.

```
contenu commun à develop et patch
«««« HEAD
contenu de develop
=======
contenu de patch
»»»» patch
```

3. En réalité, on peut fusionner un nombre arbitraire de branches d'un seul coup, mais ici on se limite à deux.

```
contenu commun à develop et patch
```

Dans ce cas, `git status` donne le nom des fichiers en conflit. GIT attend de vous de mettre ces fichiers dans la forme souhaité, et de procéder avec `git add` / `git commit` pour compléter la fusion (ou alors de l'abandonner avec `git reset -hard`).

▶ Créez une structure similaire à celle dans la Figure 1, et opérez une fusion sans conflit et une avec conflit. Contrôlez-en le résultat avec `git view`. Supprimez ensuite la branche `patch` qui ne sert plus à rien.

Déplacement des branches Dans la section précédente, la fusion a servi pour intégrer le bug-fix de `patch` dans `develop`. Le mème effet peut être obtenu avec une autre opération. Supposons que dans le cas de la Figure 2 (gauche), on faisait ceci

```
git checkout develop
git rebase patch
```

Le résultat en est revenir au plus récent ancêtre entre `develop` et `patch` (encore `37df66b`) et de créer, pour chaque commit entre les deux, créer un nouveau commit par-dessus de `patch`. On déplace alors le pointer de `develop` vers la pointe des cette nouvelle série de commits. La nouvelle structure est illustré dans la Figure 3 : deux commits ont été déplacés.

FIGURE 3 – Déplacement d'une branche par rapport à Figure 2 (gauche).

Les commandes à utiliser en cas de conflit sont légèrement différents :
— pour poursuivre l'opération, utiliser `git add` et `git rebase -continue` ;
— pour abandonner l'opération, utiliser `git rebase -abort`.

▶ Dans la Figure 3, les deux commits de `develop` ont changé d'identifiant – pourquoi ? Opérez un rebase sans et avec conflit.

Que l'on utilise `rebase` ou `merge`, le résultat pour `develop` est identique, mais avec deux historiques différentes. Pourquoi donc préférer `rebase` à `merge` ? Là aussi, il n'y a pas de règle précise : `rebase` crée une historique plus linéaire, donc plus simple à lire. Or, certains préfèrent de garder leur historique intacte. D'ailleurs, le déplacement de branches peut être dangereux dans un contexte collaboratif, voir Section 3.4.

2.6 La pile d'états sales

Dans le cours d'un développement, on est parfois ramené à interrompre son travail pour s'occuper d'une autre tâche urgente. Disons qu'on est sur `develop`, en train de travailler sur le

prochain commit, mais qu'on doit immédiatement créer un bugfix dans une autre branche. Dans ce cas, on souhaite sauvegarder le travail effectué ; mais si l'état du projet est "sale" (code qui ne compile pas, éditions incomplètes, bogues non résolus) on n'a peut-être pas envie d'en faire un commit qui "salirait" l'historique.

Pour ce cas, GIT met à disposition un espace dédie, le *stash*. Celui-ci fonctionne comme une pile.

```
git stash
```

stocke les fichiers modifiés (plus précisement : les différences par rapport à `develop`) sur cette pile et revient ensuite dans un état propre, celui de `develop`. On peut désormais s'occuper du bugfix en toute tranquilité, puis revenir sur `develop` et récupérer le travail sauvegardé avec

```
git stash pop
```

Puisque GIT stocke les *différences*, le `pop` peut même être effectué dans une autre branche. Disons que vous commencez à travailler sur un nouveau commit, et après quelques minutes vous vous rendez compte que vous avez oublié de basculer vers la bonne branche : disons, vous êtes sur `master` plutôt que sur `develop`. Aucun problème :

```
git stash
git checkout develop
git stash pop
```

3 Gérer un projet collaboratif

On va finalement tourner notre attention vers l'aspect collaboratif. GIT permet à plusieurs utilisateurs d'échanger leurs contributions. Cette collaboration peut prendre des formes assez distincts, voir le chapître 5 du *Git Book* pour des exemples. Afin de simplifier, on se concentre sur un scénario suffisant pour un petit projet :

- On suppose l'existence d'un dépôt central.
- Les programmeurs participant au projet possèdent chacun une copie locale de ce dépôt. La plupart du temps, chacun travaille sur sa copie locale, avec les opérations décrites dans la Section 2.
- De temps en temps, un programmeur envoie ses nouveaux commits au dépôt centrale, et il reçoit les contributions des autres.

Il est donc important de comprendre que `git commit` est une opération locale qui – techniquement parlant – n'affecte que le contenu de votre dossier `.git`. Le résultat d'un commit n'est donc à priori pas tout de suit visible pour les autres participants. Pour cela, il faut rajouter une opération supplémentaire.

3.1 Création d'un dépôt central

Le dépôt central est un dépôt particulier qu'on appelle *dénudé* (bare), c'est à dire sans espace de travail. Il peut soit être accessible à travers d'un système de fichiers, soit à travers le web. Pour commencer, on va travailler avec la première méthode.

▶ Créez un nouveau dépôt dans un dossier `git-new` avec au moins un commit.

On va convertir ce nouveau dépôt en un dépôt dénudé :

```
cd ..
git clone --bare git-new depot-central
```

```
Fetch URL: /home/schwoon/git/depot-central
Push URL: /home/schwoon/git/alice
HEAD branch: master
Remote branches:
  master tracked
Local branch configured for 'git pull':
  master merges with remote master
Local ref configured for 'git push':
  master pushes to master (up to date)
```

FIGURE 4 – Vue de `origin` depuis `alice`.

Cela crée un nouveau dossier `depot-central` qui contient les fichiers nécessaires, mais vous n'y toucherez jamais directement. On va plutôt créer deux copies "locales" qui communiquent avec ce dernier.

```
git clone depot-central alice
git clone depot-central bob
```

Allez dans le dossier `alice` (ou l'autre) et faites `git view`. Vous devriez voir la même historique qu'auparavant dans `git-new` (vous pouvez désormais supprimer ce dernier), à une exception près : votre nouveau dossier contient deux références supplémentaires, affichées en rouge, qui s'appellent `origin/HEAD` et `origin/master`.

D'un point de vue d'`alice`, le dépôt central est un *dépôt à distance* (remote) qui s'appelle `origin`.[4] L'information en rouge vous dit que `origin` connaît deux références qui, lors de la dernière commmunication avec `origin`, pointaient vers un tel ou tel commit. Ces deux sont des *références à distances*, et vous n'avez pas droit de les déplacer ailleurs (car ils appartiennent au dépôt à distance).

On regardera quelques informations supplémentaires sur l'interaction d'`alice` avec `origin`.

```
git remote show origin
```

Le résultat sera similaire à ce qu'on voit dans la Figure 4. On y reviendra dans le suivant.

3.2 Échange de données

▶ Effectuez d'abord quelques commits sur `master` dans `alice` et `bob`. Vérifiez que ces commits ne sont visibles que localement. Basculez ensuite vers `alice`.

Alice va maintenant envoyer ses contributions au dépôt central :

```
git push
```

Contrôlez l'évolution des références à distance avant et après cette commande.

Si Bob tente d'envoyer ses contributions lui aussi, il se verra confronté avec un message d'erreur car ses références à distance ne sont plus à jour – `origin/master` a évolué dû à Alice (c'est ce que veut dire la ligne `master pushes sur master` dans la Figure 4). Bob doit donc d'abord obtenir l'état actuel du dépôt central, mettant à jour ses références à distance :

4. Tout comme la branche `master`, `origin` n'est pas un mot-clé, c'est simplement un nom convenient que GIT utilise par défaut.

```
git fetch
```

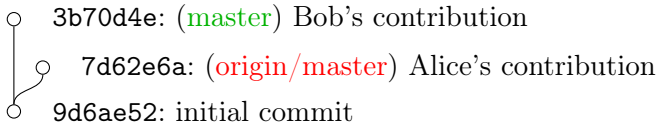

FIGURE 5 – Historique de Bob après `git fetch`.

L'historique de Bob ressemblera ainsi à la Figure 5. Bob va donc d'abord effectuer un merge/rebase pour intégrer les modifications d'Alice avec les siennes avant de les envoyer avec `git push`.

▶ Simulez plusieurs échanges entre Alice et Bob, dans les deux sens, avec et sans conflit, en utilisant `git push` et `git fetch`.

Note : La commande `git pull` permet parfois de raccourcir le travail. Il effectue un `fetch` suivi par un `merge` sur les branches mentionnés dans Figure 4 (`master merges with remote master`). La gestion avec `git fetch` est plus granulaire et permet un contrôle plus fine (p.ex. inspecter les contributions des autres avant de fusionner aveuglement).

3.3 Gestion des branches

Quand Alice et Bob créent de nouvelles branches, celles-ci ne sont pas automatiquement exportés vers le dépôt central.

▶ Créez une nouvelle branche `develop` dans le dépôt d'Alice avec au moins un commit. Vérifiez que celle-ci n'est pas exporté par `git push`.

Instruisez GIT à exporter la nouvelle branche, et récupérez-la chez Bob.

```
git push origin develop
cd ../bob
git fetch
```

▶ Regardez aussi le résultat de `git remote show origin` et dans Alice et dans Bob. Qu'observez-vous ?

3.4 Les pièges

Parmi les opérations discutés dans la Section 2, deux demandent une attention particulière dans un projet collaboratif : `git commit -amend` (Section 2.2) et `git rebase` (Section 2.5). La raison en est qu'elles modifient l'historique de vos modifications. Il est fortement déconseillé d'utiliser ces opérations sur des commits précédemment exportés vers le dépôt central. Sans entrer dans les détails, on peut facilement imaginer les dégâts ou au moins la confusion qu'une telle manipulation peut engendrer lors des échanges suivants.

3.5 Utilisation d'un dépôt dans le web

Plusieurs services web, tels que `github.com` ou similaire, offrent à leurs utilisateurs la gestion des dépôts à distance. Ceux-ci jouent alors le rôle d'un dépot central accessible à travers le web.

L'interaction précise avec ces dépôts dépend du service choisi, mais en général la seule différence syntaxique concerne la création initiale du dépôt et la commande pour en créer une copie locale. Il est recommandé d'étudier le chapître 6 du *Git Book* pour en savoir plus.