

# Utilisation de Git

---

## Ensimag 1A

Matthieu Moy, Grégory Mounié

2019-2020

Ce document peut être téléchargé depuis l'adresse suivante :  
<http://systemes.ensimag.fr/www-unix/avance/seance1-git/seance-machine-git.pdf>.

## 1 Git et la gestion de versions

Git est un gestionnaire de versions, c'est-à-dire un logiciel qui permet de conserver l'historique des fichiers sources d'un projet, et d'utiliser cet historique pour fusionner automatiquement plusieurs révisions (ou « versions »). Chaque membre de l'équipe travaille sur sa version du projet, et peut envoyer les versions suffisamment stables à ses coéquipiers via un dépôt partagé (commande `git push`) qui pourront les récupérer et les intégrer aux leurs quand ils le souhaitent (commande `git pull`).

Il existe beaucoup d'autres gestionnaires de versions. Les pages [http://ensiwiki.ensimag.fr/index.php/Gestionnaire\\_de\\_Versions](http://ensiwiki.ensimag.fr/index.php/Gestionnaire_de_Versions) ou [https://en.wikipedia.org/wiki/Comparison\\_of\\_version-control\\_software](https://en.wikipedia.org/wiki/Comparison_of_version-control_software) vous donnent un aperçu de l'existant.

Ce TP utilise le serveur Gitlab de l'Ensimag <https://gitlab.ensimag.fr>. Gitlab, site web et logiciel libre que chacun peut installer chez soi, et Github, site web unique et dominant, sont les moyens de gestion des dépôts partagés actuellement les plus utilisés. Néanmoins, les outils de bases dans l'envers du décor, et l'automatisation de la gestion des dépôts, utilisés dans certains de vos futurs projets, ne sont pas compliqués.

### 1.1 Variante pour ceux connaissant déjà git

**Si vous ne le maîtrisez pas totalement** Ce TP est à faire en équipe de deux. Ne vous binomez pas avec quelqu'un qui connaît lui-aussi `git` ! Vous savez déjà que cet outil sera particulièrement utile pour vos projets à plusieurs. Vous voulez maximiser le nombre de vos futurs équipiers qui le maîtriseront correctement et pour cela, le plus simple, est d'aider votre binôme pendant ce TP.

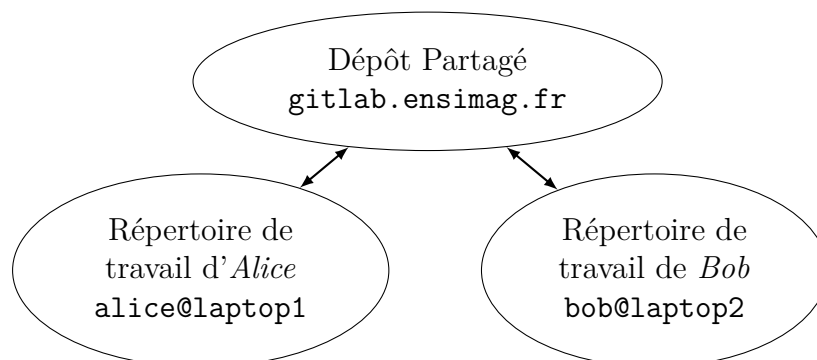
**Si vous avez peur de vous ennuyer** vous trouverez sur la page <https://git.pages.ensimag.fr/formation-git/> des slides et des TPs plus avancés. En particulier, nous conseillons de faire le "TP2 (modèle)" <https://git.pages.ensimag.fr/formation-git/tp/tp2-modele-git.pdf> sur l'implantation du stockage qui démystifie beaucoup de choses.

## 1.2 Organisation pendant la séance

Pour la séance, choisissez deux PC adjacents par équipe (on peut utiliser son ordinateur portable à la place d'un PC de l'école). Chaque étudiant travaille sur son compte.

Le dépôt partagé doit être accessible en permanence donc hébergé sur un serveur. Ce dépôt est simplement un répertoire qui contiendra l'ensemble de l'historique du projet. On ne travaillera jamais dans le dépôt directement, mais on utilisera Git pour envoyer et récupérer des révisions. Tous les membres de l'équipe auront accès à ce dépôt en lecture et en écriture. Dans la suite des explications, on suppose que le dépôt est hébergé sur `gitlab.ensimag.fr`.

L'équipe est constituée d'*Alice* qui travaille sur son portable `laptop1` et *Bob* qui travaille également sur son portable `laptop2`. Si *Alice* ou *Bob* travaille sur un PC de l'école, il faut remplacer `laptop1` ou `laptop2` par le nom de la machine (e.g. `ensipc42`). Les explications sont écrites pour 2 utilisateurs pour simplifier, mais il peut y avoir un nombre quelconque de coéquipiers.



## 2 Interlude : configurer ssh

La connexion distante au serveur Gitlab peut s'effectuer de deux façons avec `https`, ou bien avec `ssh`.


Si vous ne l'avez pas déjà fait, vous devez d'abord créer votre propre pair de clef SSH. La procédure est détaillée dans [le chapitre 4.3 du livre de référence sur Git](#)

Vous avez utilisé `ssh` pour vous connecter à une machine distante, comme par exemple :

```
1 petrot@chene[:08b09e6b]% ssh petrotf@ensipc123.ensimag.fr
2 Last login: Fri Nov 16 18:25:36 2018 from chene.tima.u-ga.fr
3 [petrotf@ensipc123 ~]$ logout
4 petrot@chene[:08b09e6b]%
```

Comme le montre l'exemple, il est possible de se connecter *sans avoir à taper son mot de passe*. C'est particulièrement utile avec `git` qui réalise souvent plusieurs connexions et exige donc que l'on tape son mot de passe  $n$  fois, ce qui rend fou.

Étapes à suivre pour permettre les connexions SSH sans mot de passe :

- Créer une paire de clefs pour `ssh` grâce à la commande `ssh-keygen`. On laissera la valeur par défaut pour le nom des clés SSH générées, il suffit donc de taper sur  deux fois avant de saisir sa phrase secrète. Il faut choisir une *pass-phrase* secrète et introuvable ou presque. Cette phrase a un pouvoir supérieur à un mot de passe, il faut donc y réfléchir à deux fois, et ne pas l'oublier, car alors il devient très délicat de se connecter sur la machine distante. Cette brillante opération n'est à faire qu'une unique fois, sauf si vous désirez créer d'autres clefs, ce qui est possible. La commande génère deux fichiers dans `~/.ssh` : d'une part `id_rsa`, qui est la clef privée à garder au chaud et à ne jamais partager avec quiconque, et d'autre part la clef publique associée `id_rsa.pub`, que l'on peut partager librement.
- Si nécessaire, lancer un agent `ssh`, `ssh-agent`, sur votre machine. La plupart des systèmes graphiques en lancent un au login, mais pas toujours, et parfois la configuration foire. Il faut en lancer un par session (juste une fois après le login, l'agent restera actif sur la machine courante jusqu'à la fin de la session). Dans tous les cas, il faut savoir faire :

```
1 petrot@chene[:08b09e6b]% eval $(ssh-agent)
2 Agent pid 27487
```

La commande retourne des variables que l'on positionne grâce à `eval`.

- Fournir la phrase à l'agent :

```
1 petrot@chene[:08b09e6b]% ssh-add
```

Normalement, cette commande va vous demander de saisir votre *passphrase*. Si ce n'est pas le cas, c'est que les clefs par défaut ne sont pas accessibles, n'ont pas le bon nom, n'ont pas de passphrase, etc.

- Copier les informations de la clef publique sur la machine distante sur laquelle on souhaite se connecter. Dans notre cas, connectez-vous à votre compte sur <https://gitlab.ensimag.fr>. Dans les *User Settings* de votre compte (troisième entrée du menu à cliquer en haut à droite de la page), vous trouverez une section *SSH Keys* (menu sur la gauche de la page). Vous n'avez plus qu'à copier-coller votre clef publique (celle dont l'extension est `.pub`) dans le champ *Key*, lui choisir un nom dans le champs *Title* et cliquer sur **Add key**.

### 3 Configuration locale de Git

Si vous travaillez sur votre machine personnelle, vérifiez que Git est installé. La commande `git`, sans argument, doit vous donner un message d'aide. Si ce n'est pas le cas, installez-le. Sous Ubuntu, « `apt-get install git gitk` » ou « `apt-get install git-core gitk` » devrait faire l'affaire, ou bien rendez-vous sur <http://git-scm.com/>.

Il faut ensuite éditer, ou créer s'il n'existe pas, le fichier `~/.gitconfig` sur la machine sur laquelle on souhaite travailler (donc sur vos portables dans notre exemple) :

```
1 emacs ~/.gitconfig # ou son éditeur préféré à la place d'Emacs !
```

Le contenu du fichier `.gitconfig` (à créer s'il n'existe pas) doit ressembler à ceci :

```
1 [core]
2 # Editeur de texte au choix, n'en garder qu'un :
3     editor = nano
4     editor = atom --wait
5     editor = code --wait
6     editor = gedit -s
7     editor = emacs
8     editor = emacsclient -c
9     editor = vim
10    editor = gvim -f
11 [user]
12     name = Prénom Nom
13     email = Prenom.Nom@grenoble-inp.org
```

La ligne `editor` de la section `[core]` définit votre éditeur de texte préféré (par exemple, `emacs`, `vim`, `atom`,...). Attention, comme vous pouvez le constater ci-dessus, certains éditeurs comme Atom, Visual Studio Code ou gedit demandent des options particulières pour être compatible avec git. Sans ces options, vous seriez confronté à l'erreur suivante lors d'un `commit` git si l'éditeur était déjà lancé :

```
1 Aborting commit due to empty commit message.
```

La ligne `editor` n'est pas obligatoire; si elle n'est pas présente, la variable d'environnement `VISUAL` sera utilisée; si cette dernière n'existe pas, ce sera la variable d'environnement `EDITOR`.

La section `[user]` est obligatoire, elle donne les informations qui seront enregistrées par Git lors d'un `commit`. Il est conseillé d'utiliser ici votre vrai nom (pas juste votre login) et votre adresse e-mail Ensimag officielle, et d'utiliser la même configuration sur toutes les machines sur lesquelles vous travaillez.

La section `[color]` est là pour rendre l'interface de Git plus jolie.

La section `[push]` permet d'avoir le même comportement avec Git 2.x et Git 1.x (utiliser `current` au lieu de `simple` avec les très vieilles versions de Git).

## 4 Mise en place

Le contenu de cette section est réalisé une bonne fois pour toutes, au début du projet. Si certains membres de l'équipe ne comprennent pas les détails, ce n'est pas très grave, nous verrons ce que tout le monde doit savoir dans la section 5.

### 4.1 Création du dépôt partagé

On va maintenant créer le dépôt partagé. Seule *Alice* fait cette manipulation, sur son compte sur `gitlab.ensimag.fr`. Le dépôt partagé doit être créé sur un serveur pour être

accessible en permanence ; `gitlab.ensimag.fr` est celui sur lesquels sont hébergés les dépôts Git à l'Ensimag.

Il faut dans un premier temps vous connecter à votre compte :

```
1 alice@laptop1$ firefox gitlab.ensimag.fr
```

Si votre mot de passe n'est pas reconnu, rendez-vous sur la page <https://copass-client.grenet.fr/> et modifiez ou re-validez votre mot de passe, puis réessayez.

#### 4.1.1 Création d'un groupe (de développeur)

Nous allons commencer par créer un groupe pour les deux membres de l'équipe. Cette étape n'est pas nécessaire, mais elle permet de gérer efficacement les droits d'accès pour un ensemble de projets.

En utilisant le menu/Bouton « + » au sommet de la page, Alice crée un groupe (*New group*) nommé avec les logins séparés par un **souligné/underscore**, par exemple `alice_bob`. Ce groupe est créé avec une visibilité *Private* (le défaut).

En utilisant le bouton « Groups » au sommet de la page, Alice sélectionne son groupe nouvellement créé et ajoute Bob aux membres du groupe (*Members* dans le menu gauche, puis « Bob » dans *Add new member*). Elle lui donnera le même rôle que le sien, celui de *Owner*.

#### 4.1.2 Création du dépôt (de code)

En utilisant le menu/Bouton « + » au sommet de la page, Alice crée un projet (*New project*) nommé `projet1`. Ce projet est créé avec le groupe `alice_bob`. Si elle était sur la page du groupe, elle peut directement indiquer que ce projet appartient au groupe. Sinon, elle peut remplacer son login (le groupe par défaut lors de la création d'un projet) par `alice_bob`.

Bob faisant partie du groupe, il a directement l'accès au projet.

Nous avons terminé la création du dépôt sur `gitlab.ensimag.fr`. Vous pouvez maintenant **revenir sur votre machine de travail habituelle** (PC de l'Ensimag ou votre machine personnelle).

La figure 1 résume les échanges entre dépôt local et distant ainsi que les différents états<sup>1</sup> dans lesquels vos fichiers vont se retrouver au fur et mesure des manipulations du TP.

---

1. <https://git-scm.com/book/en/v2/Getting-Started-Git-Basics>

Git has three main states that your files can reside in : committed, modified, and staged :

- Committed means that the data is safely stored in your local database.
- Modified means that you have changed the file but have not committed it to your database yet.
- Staged means that you have marked a modified file in its current version to go into your next commit snapshot.

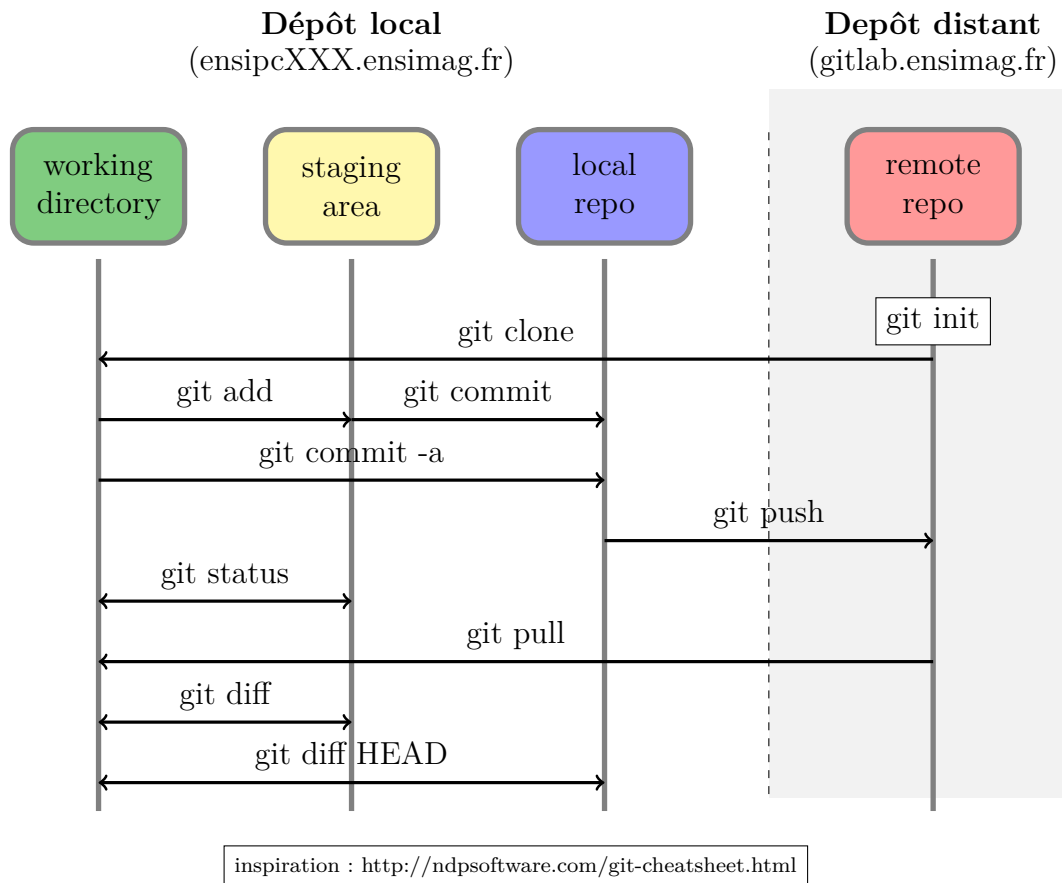


FIGURE 1 – Répertoire de travail (*working dir.*), index (*stag. area*) et dépôts (*repo.*) local/distant git

## 4.2 Création des répertoires de travail

On va maintenant créer le premier répertoire de travail. Pour l'instant, il n'y a aucun fichier dans notre dépôt, donc la première chose à faire sera d'y ajouter les fichiers sur lesquels on veut travailler. Dans notre exemple, c'est *Alice* qui va s'en occuper.

Pour créer un répertoire de travail dans le répertoire `~/projet1` (qui n'existe pas encore), *Alice* entre la commande :

```
1 alice@laptop1$ git clone ssh://git@gitlab.ensimag.fr/alice_bob/projet1.git
  ↪ projet1
```

Pour l'instant, ce répertoire est vide, ou presque : il contient un répertoire caché `.git/` qui contient les méta-données utiles à Git (c'est là que sera stocké l'historique du projet).

Pour cette séance machine, un répertoire `sandbox/` a été prévu pour vous, pour pouvoir vous entraîner sans casser un vrai projet. *Alice* télécharge le dépôt depuis <https://ensiwiki.ensimag.fr/index.php/Fichier:Sandbox.tar.gz> puis le décompresse et l'ajoute au dépôt local git :

```
1 alice@laptop1$ cd ~/projet1/
2 alice@laptop1$ tar xzvf ~/chemin/vers/le/repertoire/sandbox.tar.gz
3 alice@laptop1$ git add sandbox/
4 alice@laptop1$ git commit -m "ajout du repertoire sandbox/"
```

La commande « `git add sandbox/` » dit à Git de « traquer » tous les fichiers du répertoire `sandbox/`, c'est-à-dire qu'il va enregistrer le contenu de ces fichiers, et suivre leur historique ensuite. La commande `git commit` enregistre effectivement le contenu de ces fichiers dans le dépôt local.

Alice peut maintenant envoyer le squelette qui vient d'être importé vers le dépôt partagé :

```
1 alice@laptop1$ git push
```

Si ça ne marche pas, essayez `git push --all`.

Tout est prêt pour commencer à travailler. Bob peut à son tour récupérer sa copie de travail :

```
1 bob@laptop2$ cd
2 bob@laptop2$ git clone ssh://git@gitlab.ensimag.fr/alice_bob/projet1.git
  ↪ projet1
3 bob@laptop2$ cd projet1
4 bob@laptop2$ ls
```

Si tout s'est bien passé, la commande `ls` ci-dessus devrait faire apparaître le répertoire `sandbox/`.

## 5 Utilisation de Git pour le développement

Pour commencer, on va travailler dans le répertoire `sandbox`, qui contient deux fichiers pour s'entraîner :

```
1 both@laptops$ cd sandbox
2 both@laptops$ emacs hello.c
```

Il y a deux problèmes identifiés par des commentaires dans le fichier `hello.c`. Alice résout l'un des problèmes, et Bob choisit l'autre. Par ailleurs, chacun ajoute son nom en haut du fichier, et enregistre le résultat.

### 5.1 Création de nouvelles révision

Avec la commande `git status`, un utilisateur peut comparer le répertoire de travail et le dépôt. On voit apparaître :

```
1 both@laptops$ git status
2 On branch master
3 Changes not staged for commit:
4   (use "git add <file>..." to update what will be committed)
5   (use "git checkout -- <file>..." to discard changes in working
  ↪   directory)
6
7       modified:   hello.c
```

Ce qui nous intéresse ici est la ligne « `modified : hello.c` » qui signifie que vous avez modifié `hello.c`, et que ces modifications n'ont pas été enregistrées dans le dépôt local.

La distinction entre « Changes not staged for commit » et « Changes to be committed » n'est pas importante pour l'instant. On peut vérifier plus précisément ce qu'on vient de faire :

```
1 both@laptops$ git diff HEAD
```

Comme *Alice* et *Bob* ont fait des modifications différentes, le diff affiché sera différent, mais ressemblera dans les deux cas à :

```
1 diff --git a/sandbox/hello.c b/sandbox/hello.c
2 index a47665a..7f67d33 100644
3 --- a/sandbox/hello.c
4 +++ b/sandbox/hello.c
5 @@ -1,5 +1,5 @@
6  /* Chacun ajoute son nom ici */
7  -/* Auteurs : ... et ... */
8  +/* Auteurs : Alice et ... */
9
10 #include <stdio.h>
```

Les lignes commençant par '-' correspondent à ce qui a été enlevé, et les lignes commençant par '+' à ce qui a été ajouté par rapport au précédent commit. Si vous avez suivi les consignes ci-dessus à propos du fichier `.gitconfig`, vous devriez avoir les lignes supprimées en rouge et les ajoutées en vert.

Maintenant, *Alice* et *Bob* font :

```
1 both@laptops$ git commit -a # Enregistrement de l'état courant de
  ↪ l'arbre de travail dans le dépôt local.
```

L'éditeur est lancé et demande d'entrer un message de 'log'. Ajouter des lignes et d'autres renseignements sur les modifications apportées à `hello.c` (on voit en bas la liste des fichiers modifiés). Un bon message de log commence par une ligne décrivant rapidement le changement, suivi d'une ligne vide, suivi d'un court texte expliquant pourquoi la modification est bonne.

On voit ensuite apparaître :

```
1 [master 2483c22] Ajout de mon nom
2 1 files changed, 2 insertions(+), 12 deletions(-)
```

Ceci signifie qu'un nouveau « commit » (qu'on appelle aussi parfois « revision » ou « version ») du projet a été enregistrée dans le dépôt. Ce commit est identifié par une chaîne hexadécimale (« 2483c22 » dans notre cas).

On peut visualiser ce qui s'est passé avec les commandes

```
1 both@laptops$ gitk --all # Visualiser tout l'historique graphiquement
```

et



```
1 both@laptops$ git gui blame hello.c    # voir l'historique de chaque
  ↪ ligne du fichier hello.c
```

On va maintenant mettre ce « commit » à disposition des autres utilisateurs.

## 5.2 Fusion de révisions (merge)

SEULEMENT *Bob* fait :

```
1 bob@laptop2$ git push    # Envoyer les commits locaux dans le dépôt
  ↪ partagé
```

Pour voir où on en est, les deux équipes peuvent lancer la commande :

```
1 both@laptops$ gitk --all    # afficher l'historique sous forme graphique
```

ou bien :

```
1 both@laptops$ git log    # afficher l'historique sous forme textuelle.
```

À PRESENT, *Alice* peut tenter d'envoyer ses modifications et on voit apparaître :

```
1 alice@laptop1$ git push
2 To ssh://git@gitlab.ensimag.fr/alice_bob/projet1.git
3 ! [rejected]        master -> master (non-fast forward)
4 error: failed to push some refs to
5 'ssh://git@gitlab.ensimag.fr/alice_bob/projet1.git'
6 To prevent you from losing history, non-fast-forward updates were
  ↪ rejected
7 Merge the remote changes (e.g. 'git pull') before pushing again. See
  ↪ the
8 'Note about fast-forwards' section of 'git push --help' for details.
```

L'expression « non-fast-forward » (qu'on pourrait traduire par « absence d'avance rapide ») veut dire qu'il y a des modifications dans le dépôt vers lequel on veut envoyer nos modifications et que nous n'avons pas encore récupérées. Il faut donc fusionner les modifications avant de continuer.

L'utilisateur *Alice* fait donc :

```
1 alice@laptop1$ git pull
```

Après quelques messages sur l'avancement de l'opération, on voit apparaître :

```
1 Auto-merging sandbox/hello.c
2 CONFLICT (content): Merge conflict in sandbox/hello.c
3 Automatic merge failed; fix conflicts and then commit the result.
```

Ce qui vient de se passer est que *Bob* et *Alice* ont fait des modifications au même endroit du même fichier dans les commits qu'ils ont faits chacun de leur côté (en ajoutant leurs

noms sur la même ligne), et Git ne sait pas quelle version choisir pendant la fusion : c'est un conflit, et nous allons devoir le résoudre manuellement. Allez voir `hello.c`.

La bonne nouvelle, c'est que les modifications faites par *Alice* et Bob sur des endroits différents du fichier ont été fusionnées. Quand une équipe est bien organisée et évite de modifier les mêmes endroits en même temps, ce cas est le plus courant : les développeurs font les modifications, et le gestionnaire de versions fait les fusions automatiquement.

En haut du fichier, on trouve :

```
1 <<<<<<< HEAD
2 /* Auteurs : Alice et ... */
3 =====
4 /* Auteurs : ... et Bob */
5 >>>>>>> 2483c228b1108e74c8ca4f7ca52575902526d42a
```

Les lignes entre `<<<<<<<` et `=====` contiennent la version de votre commit (qui s'appelle HEAD). Les lignes entre `=====` et `>>>>>>>` contiennent la version que nous venons de récupérer par « pull » (nous avions dit qu'il était identifié par la chaîne 2483c22, en fait, l'identifiant complet est plus long, nous le voyons ici).

Il faut alors « choisir » dans `hello.c` la version qui convient (ou même la modifier). Ici, on va fusionner à la main (*i.e.* avec un éditeur de texte) et remplacer l'ensemble par ceci :

```
1 /* Auteurs : Alice et Bob */
```

Si *Alice* fait à nouveau `git status`, on voit apparaître :

```
1 alice@laptop1$ git status
2 On branch master
3 Your branch and 'origin/master' have diverged,
4 and have 1 and 1 different commit(s) each, respectively.
5
6 Unmerged paths:
7   (use "git add/rm <file>..." as appropriate to mark resolution)
8
9       both modified:      hello.c
10
11 no changes added to commit (use "git add" and/or "git commit -a")
```

Si on n'est pas sûr de soi après la résolution des conflits, on peut lancer la commande :

```
1 alice@laptop1$ git diff      # git diff sans argument, alors qu'on avait
   ↪ l'habitude d'appeler 'git diff HEAD'
```

Après un conflit, Git affichera quelque chose comme :

```
1 diff --cc hello.c
2 index 5513e89,614e4b9..0000000
3 --- a/hello.c
4 +++ b/hello.c
5 @@@ -1,5 -1,5 +1,5 @@@
```

```

6  /* Chacun ajoute son nom ici */
7  - /* Auteurs : Alice et ... */
8  -/* Auteurs : ... et Bob */
9  ++/* Auteurs : Alice et Bob */
10
11  #include <stdio.h>

```

(les '+' et les '-' sont répartis sur deux colonnes, ce qui correspond aux changements par rapport aux deux « commits » qu'on est en train de fusionner. Si vous ne comprenez pas ceci, ce n'est pas très grave!)

Après avoir résolu manuellement les conflits à l'intérieur du fichier, on marque ces conflits comme résolus, explicitement, avec `git add` :

```

1  alice@laptop1$ git add hello.c
2  alice@laptop1$ git status
3  On branch master
4  Your branch and 'origin/master' have diverged,
5  and have 1 and 1 different commit(s) each, respectively.
6
7  Changes to be committed:
8
9      modified:   hello.c

```

On note que `hello.c` n'est plus considéré « both modified » (c'est-à-dire contient des conflits non-résolus) par Git, mais simplement comme « modified ».

Quand il n'y a plus de fichier en conflit, il faut faire un commit (comme « `git pull` » nous l'avait demandé) :

```

1  alice@laptop1$ git commit

```

(Dans ce cas, il est conseillé, même pour un débutant, de ne pas utiliser l'option `-a`, mais c'est un détail)

Un éditeur s'ouvre, et propose un message de commit du type « Merge branch 'master' of ... », on peut le laisser tel quel, sauvegarder et quitter l'éditeur.

NB : s'il n'y avait pas eu de conflit, ce qui est le cas le plus courant, « `git pull` » aurait fait tout cela : télécharger le nouveau commit, faire la fusion automatique, et créer si besoin un nouveau commit correspondant à la fusion.

On peut maintenant regarder plus en détails ce qu'il s'est passé :

```

1  alice@laptop1$ gitk

```

Pour *Alice*, on voit apparaître les deux « commit » fait par *Bob* et *Alice* en parallèle, puis le « merge commit » que nous venons de créer avec « `git pull` ». Pour *Bob*, rien n'a changé.

La fusion étant faite, *Alice* peut mettre à disposition son travail (le premier commit, manuel, et le commit de fusion) avec :

```

1  alice@laptop1$ git push

```

et *Bob* peut récupérer le tout avec :

```
1 bob@laptop2$ git pull
```

(cette fois-ci, aucun conflit, tout se passe très rapidement et en une commande)

Les deux utilisateurs peuvent comparer ce qu'ils ont avec :

```
1 both@laptops$ gitk
```

Ils ont complètement synchronisé leurs répertoires. On peut également faire :

```
1 both@laptops$ git pull
2 both@laptops$ git push
```

Mais ces commandes se contenteront de répondre `Already up-to-date.` et `Everything up-to-date.`

## 5.3 Ajout de fichiers

À présent, *Alice* crée un nouveau fichier, `toto.c`, avec un contenu quelconque.

*Alice* fait :

```
1 alice@laptop1$ git status
```

On voit apparaître :

```
1 On branch master
2 Untracked files:
3   (use "git add <file>..." to include in what will be committed)
4
5     toto.c
6 nothing added to commit but untracked files present (use "git add" to
   ↪ track)
```

Notre fichier `toto.c` est considéré comme « Untracked » (non suivi par Git). Si on veut que `toto.c` soit ajouté au dépôt, il faut l'enregistrer (`git commit` ne suffit pas) : `git add toto.c`

*Alice* fait à présent :

```
1 alice@laptop1$ git status
```

On voit apparaître :

```
1 On branch master
2 Changes to be committed:
3   (use "git reset HEAD <file>..." to unstage)
4
5     new file:   toto.c
```

*Alice* fait à présent (-m permet de donner directement le message de log) :

```
1 alice@laptop1$ git commit -m "ajout de toto.c"
```

On voit apparaître :

```
1 [master b1d56e6] Ajout de toto.c
2 1 files changed, 4 insertions(+), 0 deletions(-)
3 create mode 100644 toto.c
```

toto.c a été enregistré dans le dépôt. Alice peut publier ce changement :

```
1 alice@laptop1$ git push
```

Bob fait à présent :

```
1 bob@laptop2$ git pull
```

Après quelques messages informatifs, on voit apparaître :

```
1 Fast forward
2 toto.c | 4 +++
3 1 files changed, 4 insertions(+), 0 deletions(-)
4 create mode 100644 toto.c
```

Le fichier toto.c est maintenant présent chez Bob.

## 5.4 Fichiers ignorés par Git

Bob crée à présent un nouveau fichier temp-file.txt, puis fait un `git status`. On voit maintenant apparaître :

```
1 bob@laptop2$ git status
2 On branch master
3 Untracked files:
4   (use "git add <file>..." to include in what will be committed)
5
6   temp-file.txt
7
8 nothing added to commit but untracked files present (use "git add" to
   ↪ track)
```

Si Bob souhaite que le fichier temp-file.txt ne soit pas enregistré dans le dépôt (soit « ignoré » par Git), il doit placer son nom dans un fichier .gitignore dans le répertoire contenant temp-file.txt. Concrètement, Bob tape la commande

```
1 bob@laptop2$ emacs .gitignore
```

et ajoute une ligne

```
temp-file.txt
```

puis sauve et quitte.

Dans le répertoire `sandbox/` qui vous est fourni, il existe déjà un fichier `.gitignore` qui peut vous servir de base pour vos projets.

Si *Bob* souhaite créer un nouveau `.gitignore` (par exemple, à la racine du projet pour que les règles s'appliquent sur tout le projet), pour que tous les utilisateurs du dépôt bénéficient du même fichier `.gitignore`, *Bob* fait :

```
1 bob@laptop2$ git add .gitignore
```

*Bob* fait à nouveau un `git status`, et on voit apparaître :

```
1 bob@laptop2$ git status
2 On branch master
3 Changes to be committed:
4   (use "git reset HEAD <file>..." to unstage)
5
6     new file:   .gitignore
```

Quelques remarques :

- Le fichier `temp-file.txt` n'apparaît plus. C'était le but de la manuvre. Une bonne pratique est de faire en sorte que « `git status` » ne montre jamais de « `Untracked files` » : soit un fichier doit être ajouté dans le dépôt, soit il doit être explicitement ignoré. Cela évite d'oublier de faire un « `git add` ».
- En général, on met dans les `.gitignore` les fichiers générés (\*.o, fichiers exécutables, ...), ce qui est en partie fait pour vous dans le `.gitignore` du répertoire `sandbox/` (qu'il faudra adapter pour faire le `.gitignore` de votre projet). Les « wildcards » usuels (\*.o, \*.ad?, ...) sont acceptés pour ignorer plusieurs fichiers.
- Le fichier `.gitignore` vient d'être ajouté (ou bien il est modifié s'il était déjà présent). Il faut à nouveau faire un commit et un push pour que cette modification soit disponible pour tout le monde.

## 5.5 Fin de l'exercice

A ce stade, vous devriez avoir les bases pour l'utilisation quotidienne de Git.

Pour utiliser Git sur un vrai TP ou projet, on peut reprendre les explications du début de ce document (section 4.1) pour créer un nouveau dépôt Git pour chaque TP.

Pour créer un projet sans utiliser gitlab, mais directement un compte accessible par ssh, les explications sont également disponibles sur la page [http://ensiwiki.ensimag.fr/index.php/Creer\\_un\\_depot\\_partage\\_avec\\_Git](http://ensiwiki.ensimag.fr/index.php/Creer_un_depot_partage_avec_Git) d'EnsiWiki.

Une configuration classique pour des étudiants Ensimag : créer un dépôt partagé par TP, et pour chaque TP, chaque étudiant de l'équipe aura sa copie de travail pour le TP. Par exemple, si *Alice* travaille avec *Bob* en TP d'algo, et avec *Charlie* en TP de méthodes numériques, il y aura un dépôt partagé pour le TP d'algo (partagé entre *Alice* et *Bob*), et une pour le TP de méthodes numériques (partagé entre *Alice* et *Charlie*). Quand *Alice* travaillera sur son TP d'algo, elle sera dans la copie de travail correspondante, et la commande `git push` enverra les changements au dépôt qu'elle partage avec *Bob*, et quand elle travaillera sur son TP de méthodes numériques, elle travaillera dans son autre copie de travail, et `git push` enverra ses changements au dépôt qu'elle partage avec *Charlie*.

On peut avoir autant de dépôts partagés qu'on veut, et autant de copie de travail qu'on veut par dépôt partagé.

Dans l'immédiat, si vous souhaitez continuer à travailler dans le dépôt que vous venez de créer, vous pouvez aussi effacer le répertoire `sandbox/` :

```
1 git rm -r sandbox/
2 git status           # Pour vérifier qu'on n'a pas fait de bêtise
3 git commit -m "Suppression de sandbox/ (exercice termine)"
```

puis continuer à travailler. Si vous avez besoin d'importer un squelette de code, vous pouvez le faire par exemple avec ces commandes :

```
1 tar xzvf ~/chemin/vers/le/squelette.tar.gz
2 git add .
3 git commit -m "import du squelette"
```

## 6 Pour conclure...

Bien sûr, Git est bien plus que ce que nous venons de voir, et nous encourageons les plus curieux à se plonger dans le manuel utilisateur et les pages de man de Git pour en apprendre plus. Au niveau débutant, voici ce qu'on peut retenir :

### Les commandes

- git commit** enregistre l'état courant de tous les fichiers qui ont été au préalable ajoutés par `git add`,
- git commit -a** enregistre l'état courant du répertoire de travail,
- git push** publie les commits,
- git pull** récupère les commits publiés,
- git add, git rm et git mv** permettent de dire à Git quels fichiers il doit surveiller ("traquer" ou "versionner" dans le jargon),
- git status, git diff HEAD** pour voir où on en est.

### Conseils pratiques

- Ne *jamais* s'échanger des fichiers sans passer par Git (email, scp, clé USB), sauf si vous savez *vraiment* ce que vous faites.
- Faire souvent `git status`, pour observer l'état du dépôt local.
- Faire un `git push` souvent, sauf si on veut garder ses modifications privées. Il peut être nécessaire de faire un `git pull` avant un `git push` si des nouvelles révisions sont disponibles dans le dépôt partagé.
- Faire des `git pull` régulièrement pour rester synchronisés avec vos collègues. Il faut faire un `git commit -a` avant de pouvoir faire un `git pull` (ce qui permet de ne pas mélanger modifications manuelles et fusions automatiques).

- Ne faites jamais un « `git add` » sur un fichier binaire généré : si vous les faites, attendez-vous à des conflits à chaque modification des sources ! Git est fait pour gérer des fichiers sources, pas des binaires.

(quand vous ne serez plus débutants<sup>2</sup>, vous verrez que la vie n'est pas si simple, et que la puissance de Git vient de `git commit` sans `-a`, des `git commit` sans `git push`, ... mais chaque chose en son temps !)

## Quand rien ne va plus. . .

En cas de problème avec l'utilisation de Git :

- Consulter la page [http://ensiwiki.ensimag.fr/index.php/FAQ\\_Git](http://ensiwiki.ensimag.fr/index.php/FAQ_Git) sur Ensi-Wiki. Cette page a été écrite pour le projet GL, mais la plupart des explications s'appliquent directement pour vous,
- Demander de l'aide aux enseignants,
- Demander de l'aide sur la mailing-list de Git,
- En cas de problème non-résolu et bloquant, poser la question par email à un enseignant (par exemple Grégory Mounié).

Dans tous les cas, lire la documentation est également une bonne idée : <http://git-scm.com/documentation> ! Par exemple, le livre numérique de Scott Chacon « Pro Git », simple d'accès et traduit en français : <http://git-scm.com/book/fr/v2>

---

2. cf. par exemple [http://ensiwiki.ensimag.fr/index.php/Maintenir\\_un\\_historique\\_propre\\_avec\\_Git](http://ensiwiki.ensimag.fr/index.php/Maintenir_un_historique_propre_avec_Git)