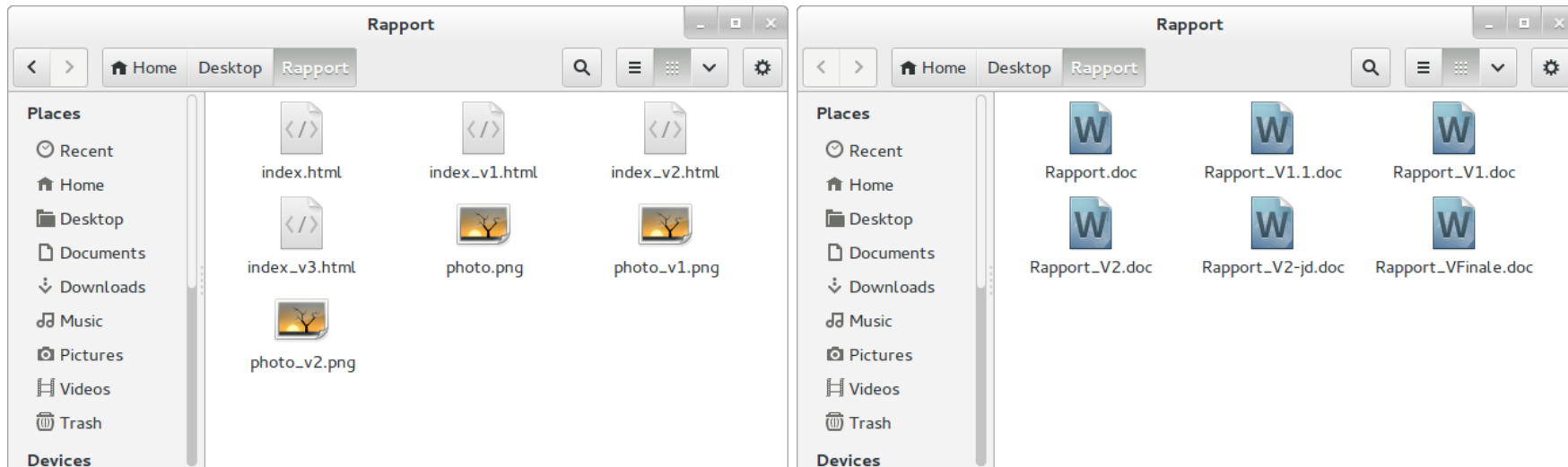


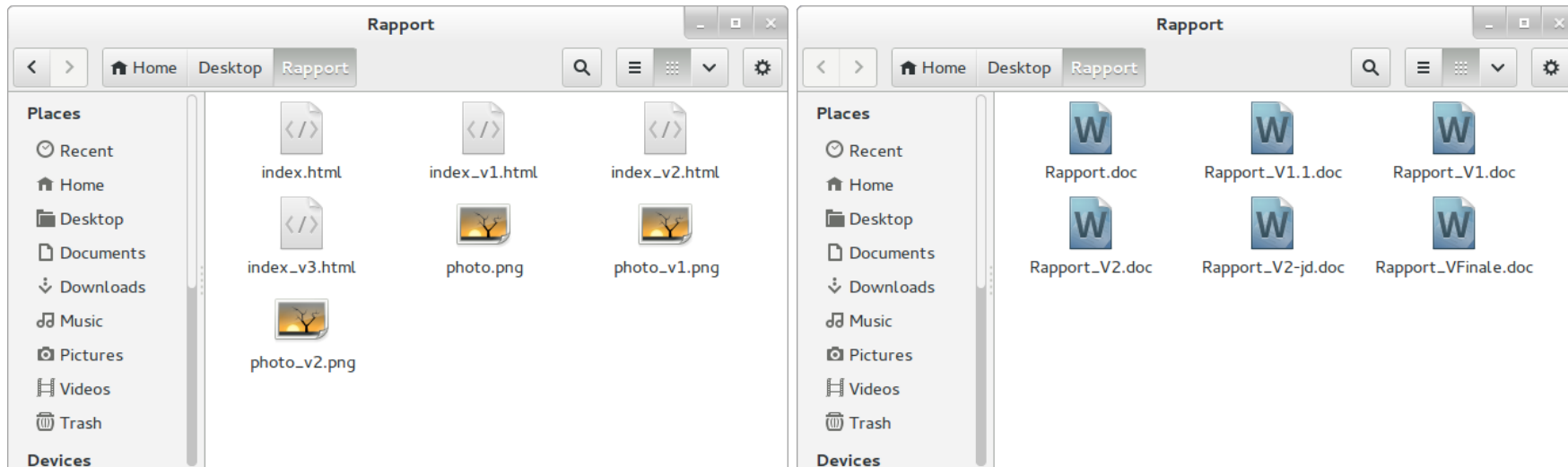


Ressources :

- Crash course : <https://www.atlassian.com/fr/git/tutorials/setting-up-a-repository>
- <https://perso.liris.cnrs.fr/pierre-antoine.champin/enseignement/intro-git/>
- Démo interactive pour devenir expert : https://learngitbranching.js.org/?locale=fr_FR



- la version la plus à jour est-elle Rapport.doc ou Rapport_VFinale.doc ?
- et si on avait aussi Rapport_VFinale1.doc et Rapport_VFinale2.doc ?
- les versions n'apparaissent pas dans l'ordre (1.1, 1, 2)
- ...



- la version la plus à jour est-elle Rapport.doc ou Rapport_VFinale.doc ?
- et si on avait aussi Rapport_VFinale1.doc et Rapport_VFinale2.doc ?
- les versions n'apparaissent pas dans l'ordre (1.1, 1, 2)
- ...

Solutions :

- Édition collaborative en temps réel (Overleaf, OnlyOffice...)
- Systèmes de gestion de versions (**git**, svn, mercurial, Dropbox...)

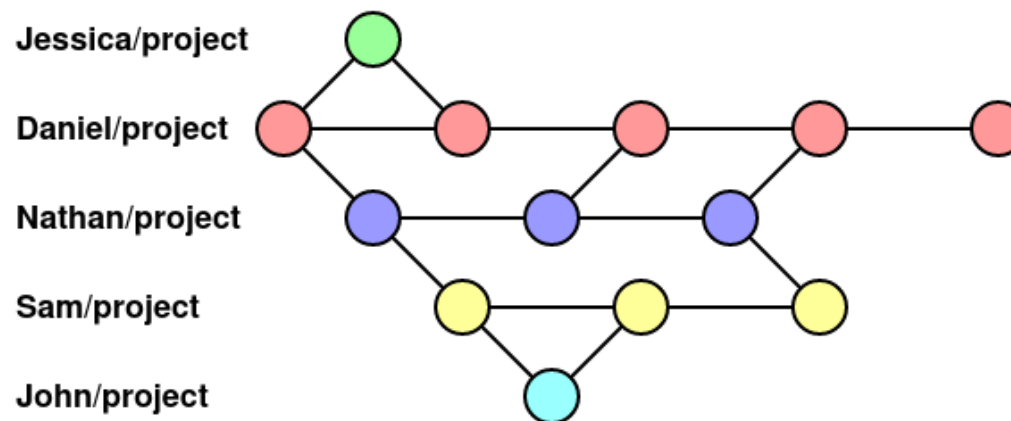
- Sauvegarde et distribution (modulo la synchronisation avec un serveur distant)
- Conservation de l'historique (nominatif) des fichiers (qui a fait quoi ?), visualiser les changements au cours du temps
- Possibilité de retour en arrière
- Fusion des modifications lors du travail collaboratif

→ Très adapté au développement informatique, mais pas seulement (dépôts de données, rédaction collaborative de documentation / documents normatifs, synchronisation de dossiers...)

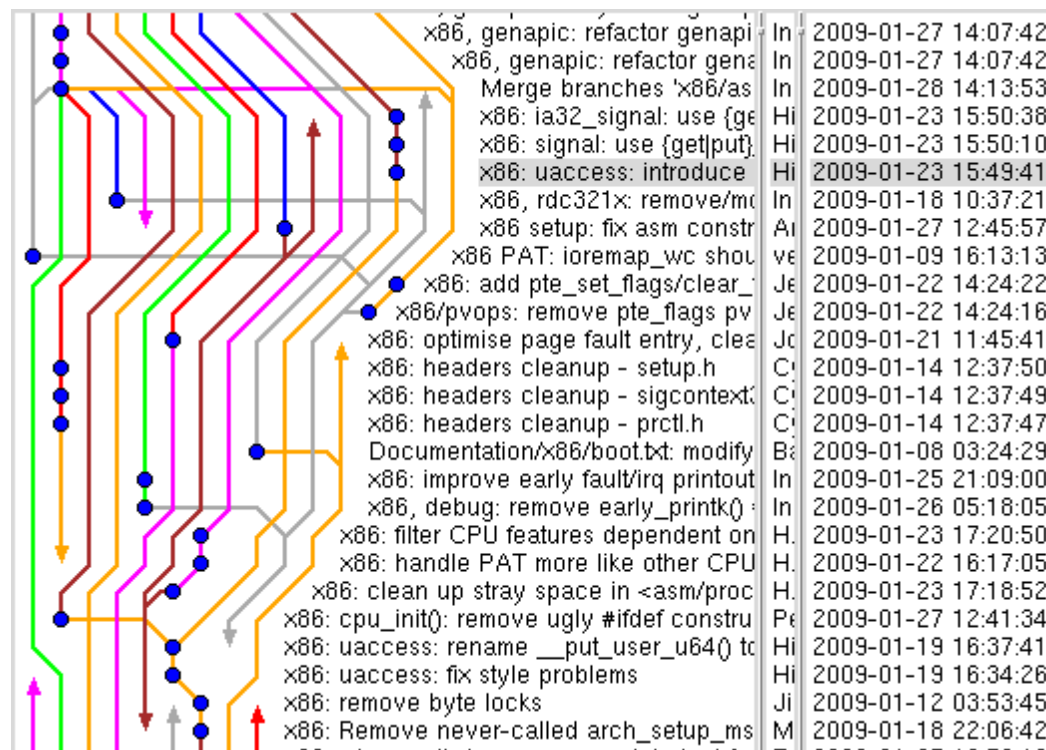
- Sauvegarde et distribution (modulo la synchronisation avec un serveur distant)
- Conservation de l'historique (nominatif) des fichiers (qui a fait quoi ?), visualiser les changements au cours du temps
- Possibilité de retour en arrière
- Fusion des modifications lors du travail collaboratif

→ Très adapté au développement informatique, mais pas seulement (dépôts de données, rédaction collaborative de documentation / documents normatifs, synchronisation de dossiers...)

Chaque collaborateur a une version locale du projet : un **dépôt** / *repository*



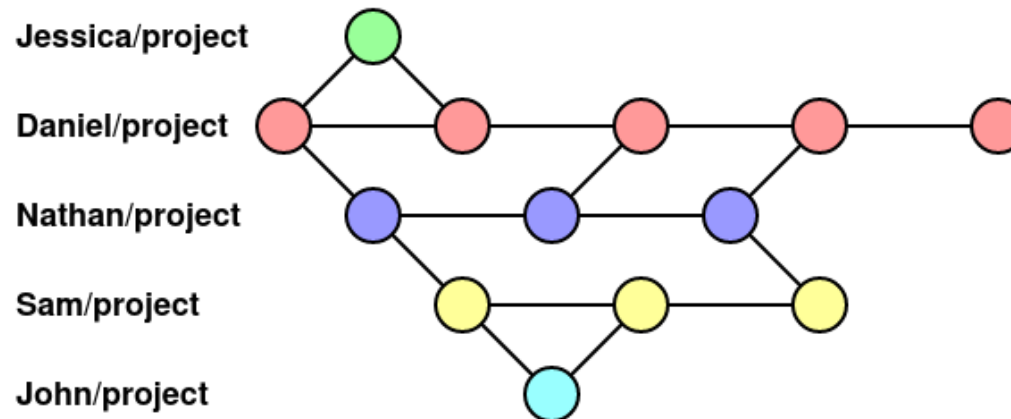
Très puissant. En pratique sur des gros projets :



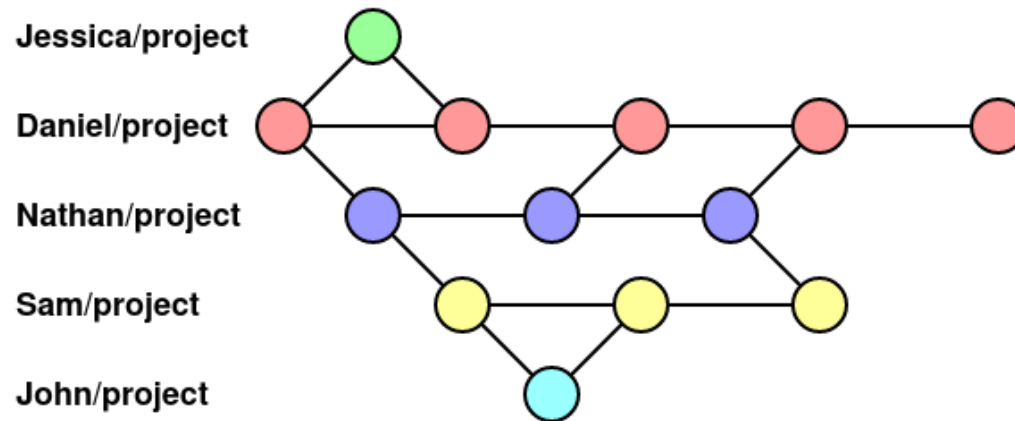
Chaque modification enregistrée et nommée = **commit**. Sorte d'*instantané* du projet.

- Date/heure • Auteur • Description textuelle
- Commit précédent parent, et différences (*deltas*) par rapport au commit précédent

⚠ Contrairement à un système d'édition en temps réel, la notion de "présent" n'existe pas. Chaque dépôt du projet a sa propre histoire, comme en relativité :



⚠ Contrairement à un système d'édition en temps réel, la notion de "présent" n'existe pas. Chaque dépôt du projet a sa propre histoire, comme en relativité :

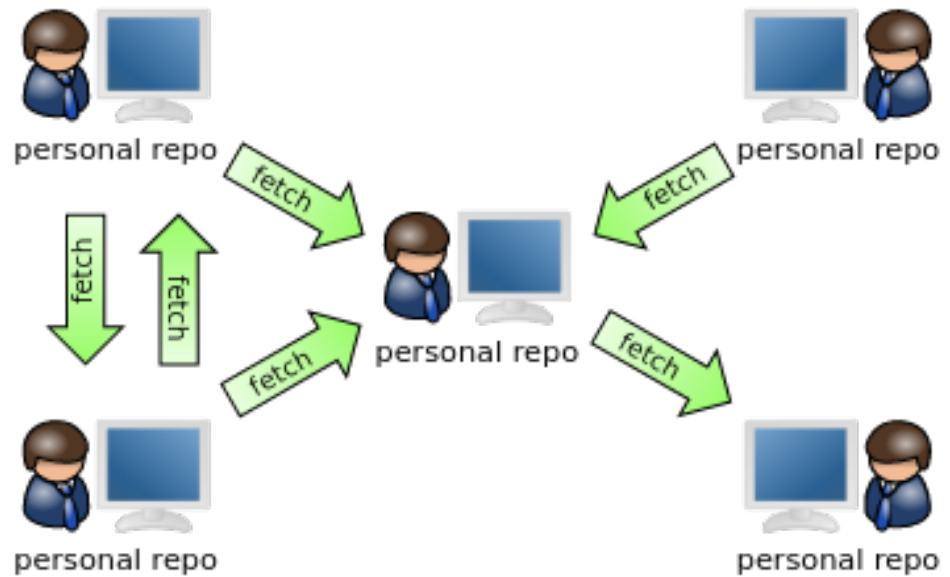


Quand on commit localement, on n'a aucune idée si des modifications ont été apportées chez quelqu'un d'autre.

Et c'est utile ! On a pas envie d'être dérangé dans son travail par les modifications des autres.

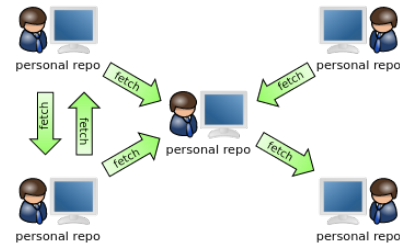
Organisation pair-à-pair

Décentralisé mais
pas pratique.



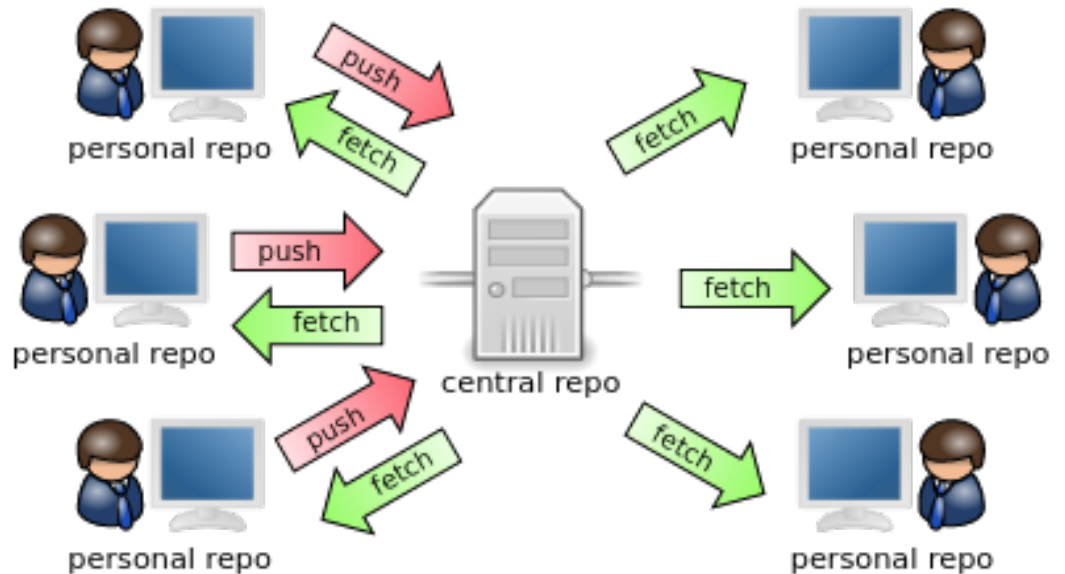
Organisation pair-à-pair

Décentralisé mais
pas pratique.



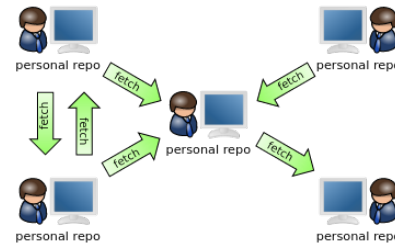
Organisation en étoile

La plus simple pour
une équipe.



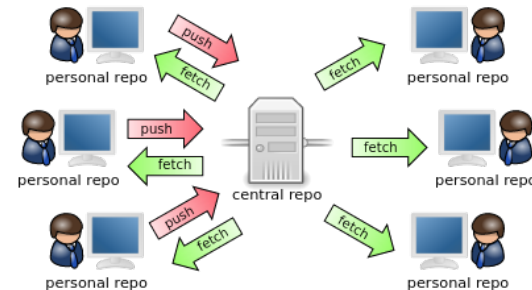
Organisation pair-à-pair

Décentralisé mais
pas pratique.



Organisation en étoile

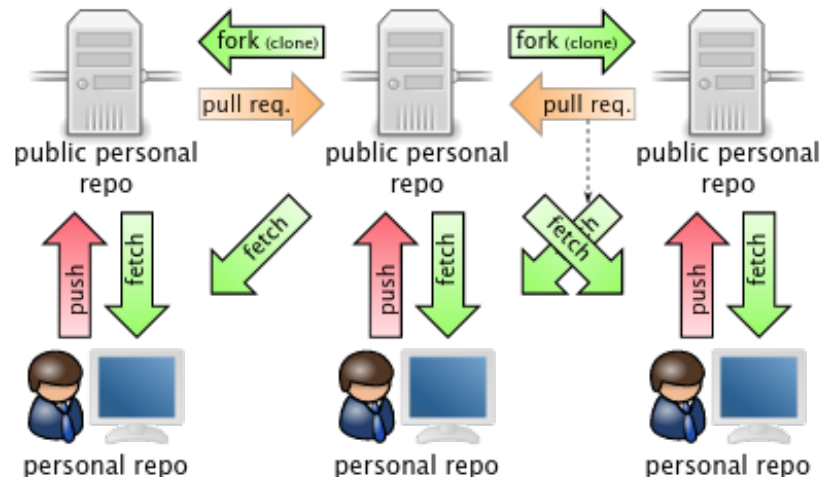
La plus simple pour
une équipe.



Organisation à la github

Chaque collaborateur dispose
d'un clone public (*fork*) du
projet, et y publie les commits
qu'il souhaite partager.

Il sollicite ensuite les autres col-
laborateur pour *pull* ces com-
mits dans leur propre dépôt
(*pull request*).



- Doit être un changement cohérent, qui a un sens pris tout seul. Ne pas faire un commit qui modifie plusieurs aspects du programme à la fois. → Découper en commits cohérents.

- Doit être un changement cohérent, qui a un sens pris tout seul. Ne pas faire un commit qui modifie plusieurs aspects du programme à la fois. → Découper en commits cohérents.

Exemples : correction d'un bug, ajout/suppression d'une fonctionnalité...

- Doit être un changement cohérent, qui a un sens pris tout seul. Ne pas faire un commit qui modifie plusieurs aspects du programme à la fois. → Découper en commits cohérents.

Exemples : correction d'un bug, ajout/suppression d'une fonctionnalité...

- Laisser le projet dans un état propre et cohérent. Si le programme compilait/fonctionnait avant le commit, il doit le rester après. → Respect des collaborateurs.

- Doit être un changement cohérent, qui a un sens pris tout seul. Ne pas faire un commit qui modifie plusieurs aspects du programme à la fois. → Découper en commits cohérents.

Exemples : correction d'un bug, ajout/suppression d'une fonctionnalité...

- Laisser le projet dans un état propre et cohérent. Si le programme compilait/fonctionnait avant le commit, il doit le rester après. → Respect des collaborateurs.
- Messages de commits descriptifs !



	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAAAA	3 HOURS AGO
○	ADKFJSLKDFJSOKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT
MESSAGES GET LESS AND LESS INFORMATIVE.

<https://xkcd.com/1296/>

 Dashboard Issues Pull Requests Explore   

New Repository

A repository contains all project files, including revision history. [Migrate repository.](#)

Owner  xif

Some organizations may not show up in the dropdown due to a maximum repository count limit.

Repository Name

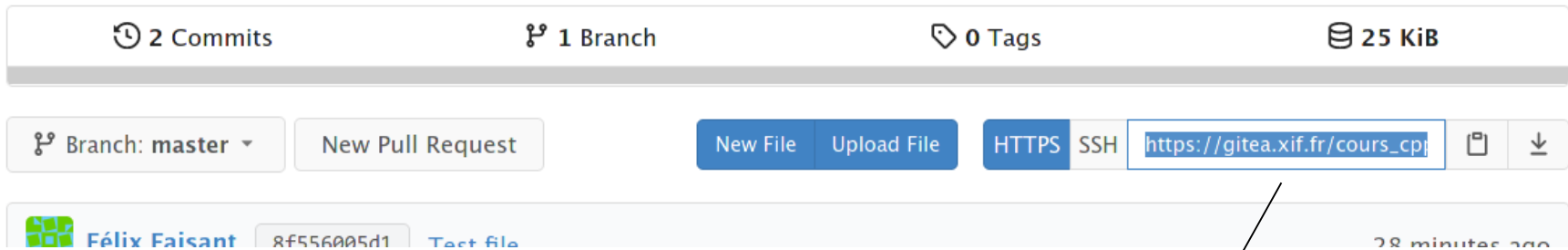
Good repository names use short, memorable and unique keywords.

Visibility ☐ Make Repository Private

Only the owner or the organization members if they have rights, will be able to see it.

Description

 New Repository
 New Migration
 New Organization



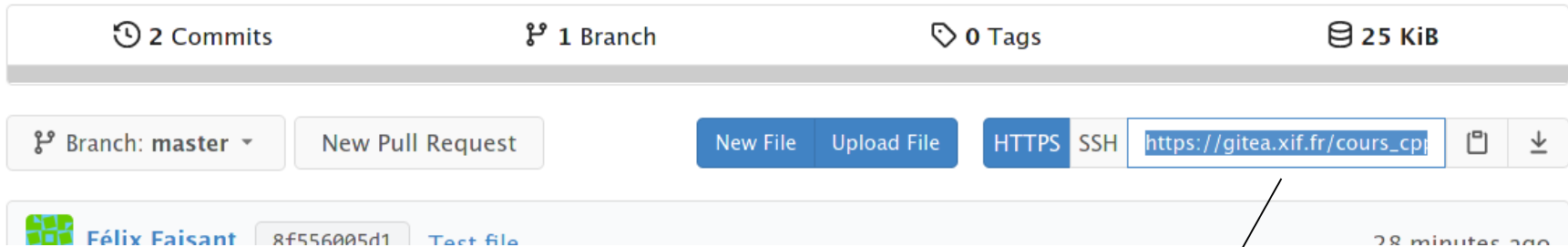
Clonage du dépôt sur la machine locale, *remote* HTTPS :

```
$ git clone https://gitea.xif.fr/cours_cpp/repo_test.git
Cloning into 'repo_test'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
```

Changement d'éditeur si on aime pas vim :

```
$ git config --global core.editor "nano"
```

Le clonage crée un *nouveau* dépôt sur la machine, avec le même historique que le dépôt distant. À ne faire qu'une seule fois par machine/collaborateur !



Clonage du dépôt sur la machine locale, *remote* HTTPS :

```
$ git clone https://gitea.xif.fr/cours_cpp/repo_test.git
Cloning into 'repo_test'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
```

Changement d'éditeur si on aime pas vim :

```
$ git config --global core.editor "nano"
```

On se place dans le dépôt (toutes les commandes git relatives à un dépôt doivent être exécutées dans ce dossier) :

```
$ cd repo_test/
```

Création / modification d'un fichier :

```
$ echo "Coucou !" >> text.txt
```

Création / modification d'un fichier :

```
$ echo "Coucou !" >> text.txt
```

```
$ git status
```

```
On branch master
```

```
Your branch is up to date with 'origin/master'.
```

```
Changes not staged for commit:
```

```
  (use "git add <file>..." to update what will be committed)
```

```
  (use "git restore <file>..." to discard changes in working directory)
```

```
    modified:   text.txt
```

Création / modification d'un fichier :

```
$ echo "Coucou !" >> text.txt
```

```
$ git status
```

```
On branch master
```

```
Your branch is up to date with 'origin/master'.
```

```
Changes not staged for commit:
```

```
  (use "git add <file>..." to update what will be committed)
```

```
  (use "git restore <file>..." to discard changes in working directory)
```

```
    modified:   text.txt
```

Prise en compte du fichier dans le prochain commit / modification d'un fichier :

```
$ git add text.txt
```

(git add, ce n'est pas seulement pour ajouter un fichier, c'est à faire à *chaque commit* pour sélectionner les modifications que l'on veut prendre en compte; on pourra faire `git add .` pour prendre en compte toutes les modifications effectuées)

Création / modification d'un fichier :

```
$ echo "Coucou !" >> text.txt
```

Prise en compte du fichier dans le prochain commit / modification d'un fichier :

```
$ git add text.txt
```

```
$ git status
```

On branch master

Your branch is up to date with 'origin/master'.

Changes to be committed:

(use "git restore --stage <file>..." to unstage)

modified: text.txt

Création / modification d'un fichier :

```
$ echo "Coucou !" >> text.txt
```

Prise en compte du fichier dans le prochain commit / modification d'un fichier :

```
$ git add text.txt
```

```
$ git status
```

On branch master

Your branch is up to date with 'origin/master'.

Changes to be committed:

(use "git restore --stage <file>..." to unstage)

modified: text.txt

Commit :

```
$ git commit -m "Modification d'un fichier test"
```

```
[master 8f55600] Modification d'un fichier test
```

```
1 file changed, 1 insertion(+), 1 deletion(-)
```

Push (envoi des nouveaux commits vers le dépôt central) :

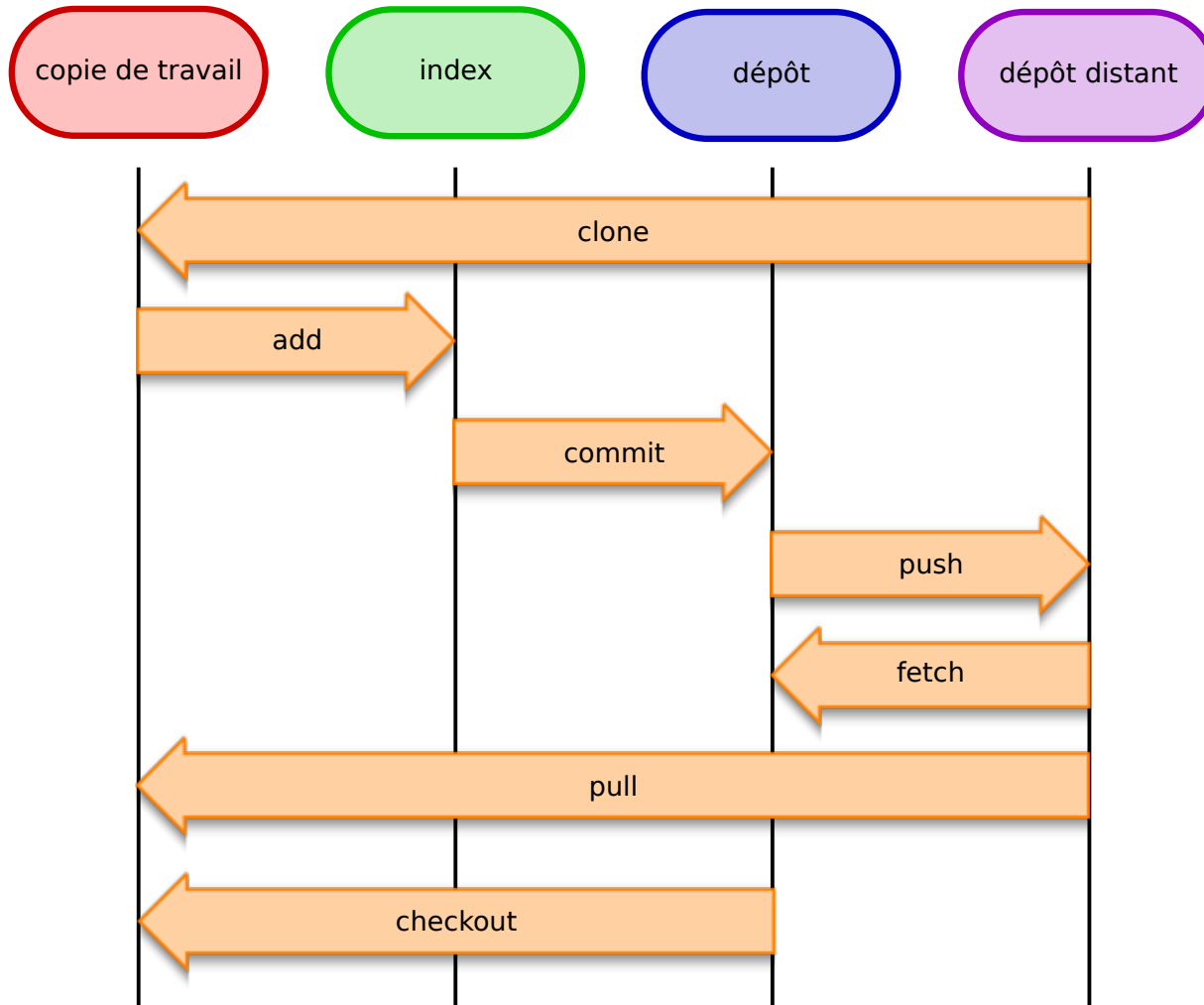
```
$ git push
Username for 'https://gitea.xif.fr': xif
Password for 'https://xif@gitea.xif.fr': *****
Writing objects: 100% (3/3), 288 bytes | 288.00 KiB/s, done.
To https://gitea.xif.fr/cours_cpp/repo_test.git
    4a0d8ea..8f55600  master -> master
```

Push (envoi des nouveaux commits vers le dépôt central) :

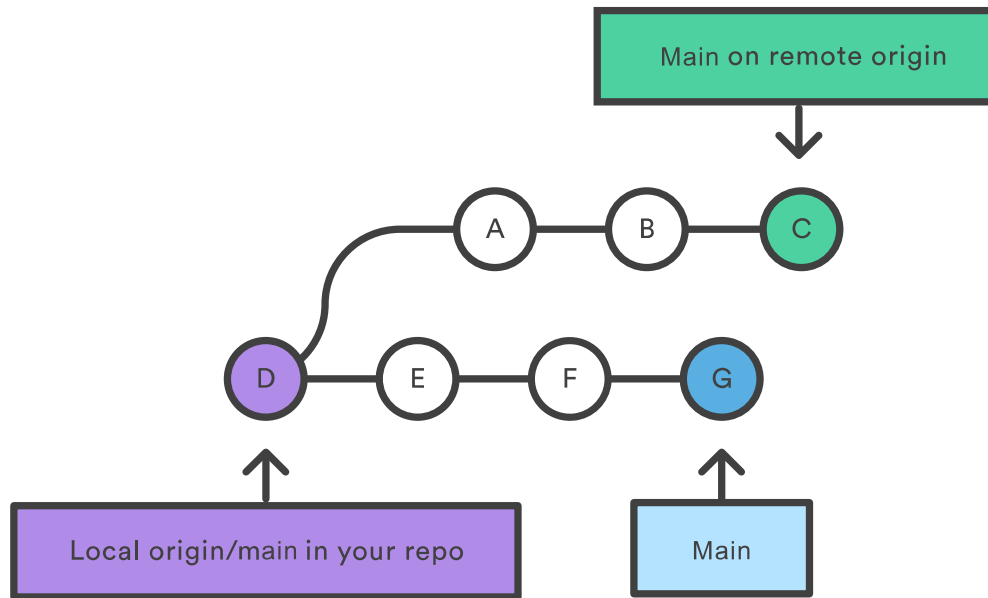
```
$ git push
Username for 'https://gitea.xif.fr': xif
Password for 'https://xif@gitea.xif.fr': *****
Writing objects: 100% (3/3), 288 bytes | 288.00 KiB/s, done.
To https://gitea.xif.fr/cours_cpp/repo_test.git
    4a0d8ea..8f55600  master -> master
```

Pull (récupération des nouveaux commits créés par les autres collaborateurs depuis le dépôt central = fetch, puis application des modifications aux fichiers = checkout) :

```
$ git pull
remote: Enumerating objects: 4, done.
Unpacking objects: 100% (3/3), 297 bytes | 297.00 KiB/s, done.
From https://gitea.xif.fr/cours_cpp/repo_test
    8f55600..ae7f975  master      -> origin/master
Updating 8f55600..ae7f975
Fast-forward
   truc.txt | 1 +
   1 file changed, 1 insertion(+)
```

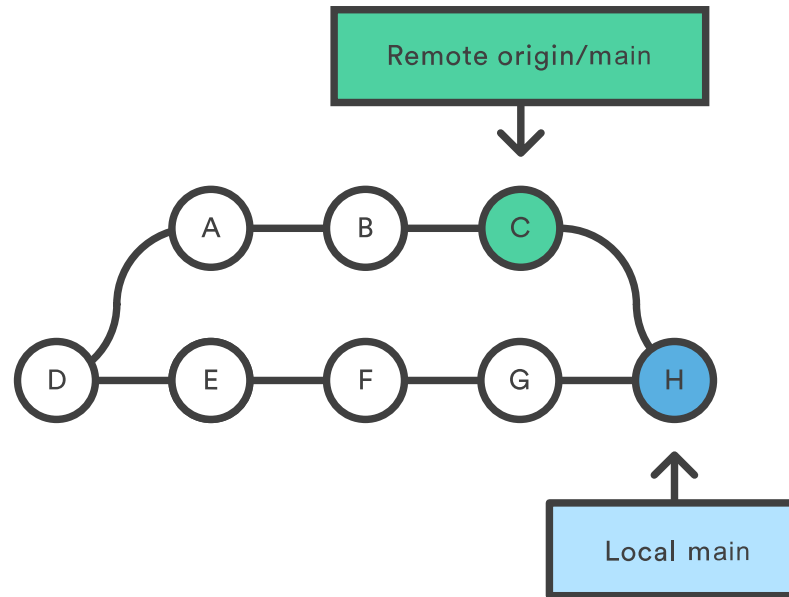


Que se passe-t-il si un (ou plusieurs) collaborateur a effectué des modifications et a *commit* entre temps, et que les histoires ont divergées ?¹



1. <https://www.atlassian.com/fr/git/tutorials/syncing/git-pull>

Que se passe-t-il si un (ou plusieurs) collaborateur a effectué des modifications et a *commit* entre temps, et que les histoires ont divergées ?²



`git pull` va, si possible, **fusionner** les modifications (*merge*) et créer un commit H réconciliant la branche locale et la branche distante. Ensuite, un `git push` met à jour le dépôt distant.

Sinon, c'est un *conflict*. Il faut le résoudre à la main en modifiant les fichiers, puis `commit`³.

². <https://www.atlassian.com/fr/git/tutorials/syncing/git-pull>

³. Voir <https://perso.liris.cnrs.fr/pierre-antoine.champin/enseignement/intro-git/#gerer-les-conflits>

1. Préféablement, achever ses modifications et avoir un dossier propre. Regarder l'état du dépôt avec `git status`

2. Enregistrer toutes les modifications dans un ou plusieurs commits :

```
git add .
```

```
git commit -m "Description"
```

3. Le serveur refusera que l'on push si quelqu'un d'autre a push entre temps, car les histoires ont divergées et il ne pourra pas simplement "fast-forward". Il faut alors fusionner les deux histoires :

```
git pull
```

(qui fait `git fetch` pour télécharger, puis `git merge` pour fusionner, puis `git checkout` pour appliquer les modifications). Si il y a un conflit, il faut **1.** régler le conflit dans le(s) fichier(s) en choisissant la bonne version, **2.** `git add` sur ce(s) fichier(s), **3.** `git commit` pour achever la fusion.

4. Enfin,

```
git push
```

SSH = secure shell. Seul mode d'accès possible pour *push* sur github.
Pas de mot de passe, à la place : clé SSH ($\approx$ certificat cryptographique).

SSH = secure shell. Seul mode d'accès possible pour *push* sur github.

Pas de mot de passe, à la place : clé SSH ($\approx$ certificat cryptographique).

Il faut **créer une paire de clés** asymétrique, et donner la clé publique au serveur (github/gitea...) ⁴ :

```
$ ssh-keygen -t ed25519 -C "nom@machine"
```

```
Generating public/private ed25519 key pair.
```

```
Enter file in which to save the key (/home/xif/.ssh/id_ed25519):
```

```
Enter passphrase (empty for no passphrase): pas_forcément_le_mdp_github
```

```
Enter same passphrase again: *****
```

```
Your identification has been saved in .ssh/id_ed25519
```

```
Your public key has been saved in .ssh/id_ed25519.pub [clé publique]
```

```
$ cat .ssh/id_ed25519.pub
```

```
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIFjYijS0xPGpBkpPmMXPNJYp6lNUcQd6ImvRbrJxP xif@t440-xif
```

4. <https://docs.github.com/en/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent>

SSH = secure shell. Seul mode d'accès possible pour *push* sur github (avec le client github).
Pas de mot de passe, à la place : clé SSH (= certificat cryptographique).

Il faut **créer une paire de clés** asymétrique, et donner la clé publique au serveur (github/gitea...) :

```
$ ssh-keygen -t ed25519 -C "nom@machine"
```

Your identification has been saved in `.ssh/id_ed25519`

Your public key has been saved in `.ssh/id_ed25519.pub` **[clé publique]**

```
$ cat .ssh/id_ed25519.pub
```

```
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIFjYijSOxPGpBkpPmMXPNJYp6lNUcQd6ImvRbrJxP xif@t440-xif
```

Sur github (<https://github.com/settings/ssh/new>) ajouter le contenu de la clé publique

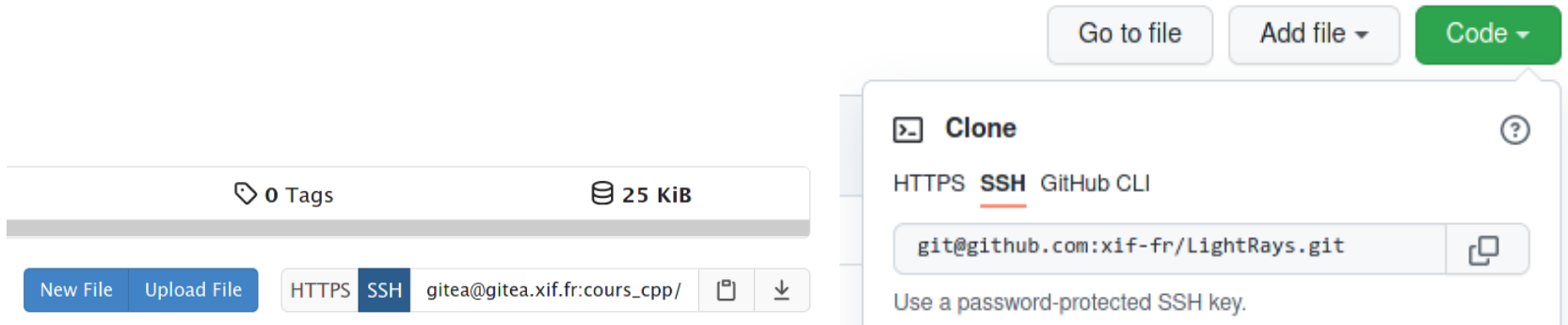
SSH keys / Add new

Title

Ma super clé

Key

```
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIFjYijSOxPGpBkCOMpPmMXPNJYp6lNUcQd6ImvRbrJxP  
xif@t440-xif
```



```
$ git clone git@github.com:xif-fr/LightRays.git
```

C'est la seule différence. Pour un dépôt local déjà existant avec un remote HTTP, un remote SSH peut être ajouté :

```
$ git remote add ssh_remote git@github.com:truc/bidule.git  
$ git push ssh_remote
```

Au moment de push, ne pas rentrer le mot de passe github, rentrer le mot de passe *de la clé* !

- `git status` : statut du dépôt local
- `git log --graph --oneline` : historique sous forme d'un arbre
- `git add <fichier>` : ajout d'un fichier dans la *staging area* / l'index
`git add *` pour ajouter toutes les modifications
- `git commit -m "message"` (ou `git commit` en interactif) : enregistrer ($\pm$ définitivement) les modifications dans l'arbre
- `git commit --amend` : modifier un commit après coup (message, fichier manquant...); possible uniquement si le commit n'a pas été push !
- `git push <remote>` : envoi des commits au dépôt central (possible uniquement si personne n'a push sur le dépôt central entre temps; si c'est le cas, faire `git pull` avant)
- `git pull <remote>` (= `git fetch`, `git merge` si nécessaire puis `git checkout`) : télécharge les nouveaux commits et applique les modifications sur la copie de travail; possible uniquement si la copie de travail est propre
- `git restore <fichier>` : annule/écrase les modifications non commitées d'un fichier de la copie de travail (attention, irréversible !)
- `git reset --hard HEAD~n` : revient à *n* commits en arrière en écrasant (!) les fichiers

<https://docs.github.com/en/authentication/troubleshooting-ssh/using-ssh-over-the-https-port>