Colloque « Sciences ouvertes »

Atelier Git

Philippe Dosch

philippe.dosch@loria.fr

Juin 2022

Sommaire

Sommaire

Sommaire

Problématiques générales de recherche

- Ouverture de la science
  - mouvement international : recherche et données produites accessibles à tous
  - appuyée par des politiques nationales, européennes
- Au CNRS, s'inscrit dans une démarche plus large (voir `https://www.cnrs.fr/sites/default/files/ressource-file/Pratiquer-une-recherche-integre-et-responsable-2017.pdf`)

- Un lien intéressant pour resituer : `https://barometredelascienceouverte.esr.gouv.fr/`

Concerne qui ? quoi ?

- Concerne déjà de nombreuses disciplines : astronomie, biologie, éducation, informatique, mathématique, médecine, sciences sociales...
- Concerne toutes les données manipulées
  - corpus, expérimentations, indicateurs, données brutes ou synthétisées diverses
  - codes informatiques (pas qu'en recherche d'ailleurs, voir `https://code.gouv.fr/`)
- Données et code (quand approprié) vont de pair

Un exemple apparenté récent

(*fondateur de Vitemadose et de CovidTracker*)

Avantages d'une science ouverte

Une science ouverte présente de nombreux atouts

- meilleures collaborations, meilleures participations
- transparence, traçabilité
- reproductibilité
- diffusion + universelle, potentiellement tous niveaux de sociétés
- meilleure réactivité
- meilleur facteur d'impact au niveau scientifique (contraintes de soumission parfois, adoption favorisée ensuite)

Quid du stockage et des manipulations ?

- Ok, je suis convaincu mais... je fais quoi ? comment ?
- Nécessité de plusieurs briques et besoins de *standards*
  - sur le stockage de données
  - sur la manipulation des données (qui ? comment ? quels droits ?)
  - sur le code informatique
  - et encore : sur la sauvegarde, sur le *versionning*...
- Les données de la science ouverte s'appuient sur des solutions, outils et protocoles éprouvés, universels, interopérables

Panorama rapide des solutions de stockage

	Stockage données	**Usage type**	**Gestion des droits**	**Sauvegarde et reprise**	**Versionning**
Serveur FTP	fichiers	*data*	RW	à gérer côté serveur	non
Bases de données	données structurées	*data*	RWA très fin	à gérer côté serveur	pas nativement
Drive / Cloud	fichiers	*data*	RW	peuvent être natives et multiples	propre à chaque solution
Gest. versions	fichiers et branches	code	RWA + workflows	natives et multiples	natif

(*R : lecture, W : écriture, A : administration,*

branche : ensemble de fichiers dont l'état — la version — est cohérent globalement)

Sommaire

Problématiques générales liées au code informatique

- Comment gérer l'historique des fichiers d'un projet ?
  - archivage
  - comparaison de la version courante par rapport à une ancienne
  - récupération d'une ancienne version
- Comment gérer les différentes versions d'un projet ?
  - version 1, version 2... : une version est un ensemble de fichiers dans un état donné
  - développements parallèles : version stable, alternatives

Problématiques spécifiques au travail en groupe

- Comment partager du code ?
- Comment travailler à plusieurs sur du code ?
- Comment travailler au même moment sur du code ?
- Comment réconcilier les changements de contributeurs ?
- Comment ne pas perdre de travail ?

Une solution : les systèmes de gestion de version (VCS)

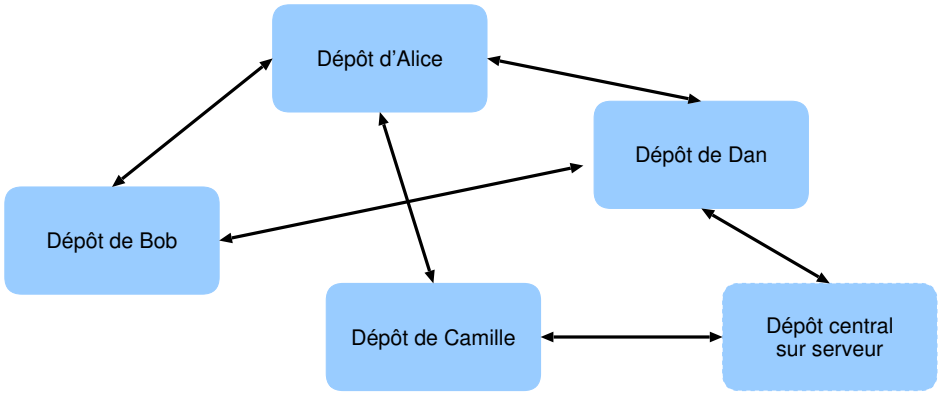
Ensemble de méthodes et d'outils qui maintiennent les différentes versions d'un projet à travers tous les fichiers qui le composent

- permet le développement collaboratif et simultané
- permet de garder tout l'historique de tous les fichiers
- permet le développement parallèle : version principale, versions alternatives
- permet de savoir pourquoi, quand et par qui une modification a été introduite

Git

- Créé en 2005 par Linus Torvalds pour la gestion des sources de Linux, dont il est également le créateur
- Usage intensif depuis pour le développement logiciel
- Popularisé par des sites et outils comme Github, Bitbucket, Gitlab...
- Très performant sur la gestion des codes informatiques, mais également utilisé pour des données *binaires* (images, PDF, documents « office », *etc*.)
- Juste sur Github (juin 2022) : $\approx$ 83 millions d'utilisateurs, $\approx$ 200 millions de dépôts

Principe général de fonctionnement



- Chaque utilisateur possède un *dépôt* local complet, contenant tout l'historique du projet
- Les opérations sont réalisées localement (*i.e. offline*)
- Côté synchronisation
  - des serveurs *peuvent* assurer les échanges de données
  - des données *peuvent* aussi être échangées avec d'autres utilisateurs sans serveur centralisé

Sommaire

Sommaire

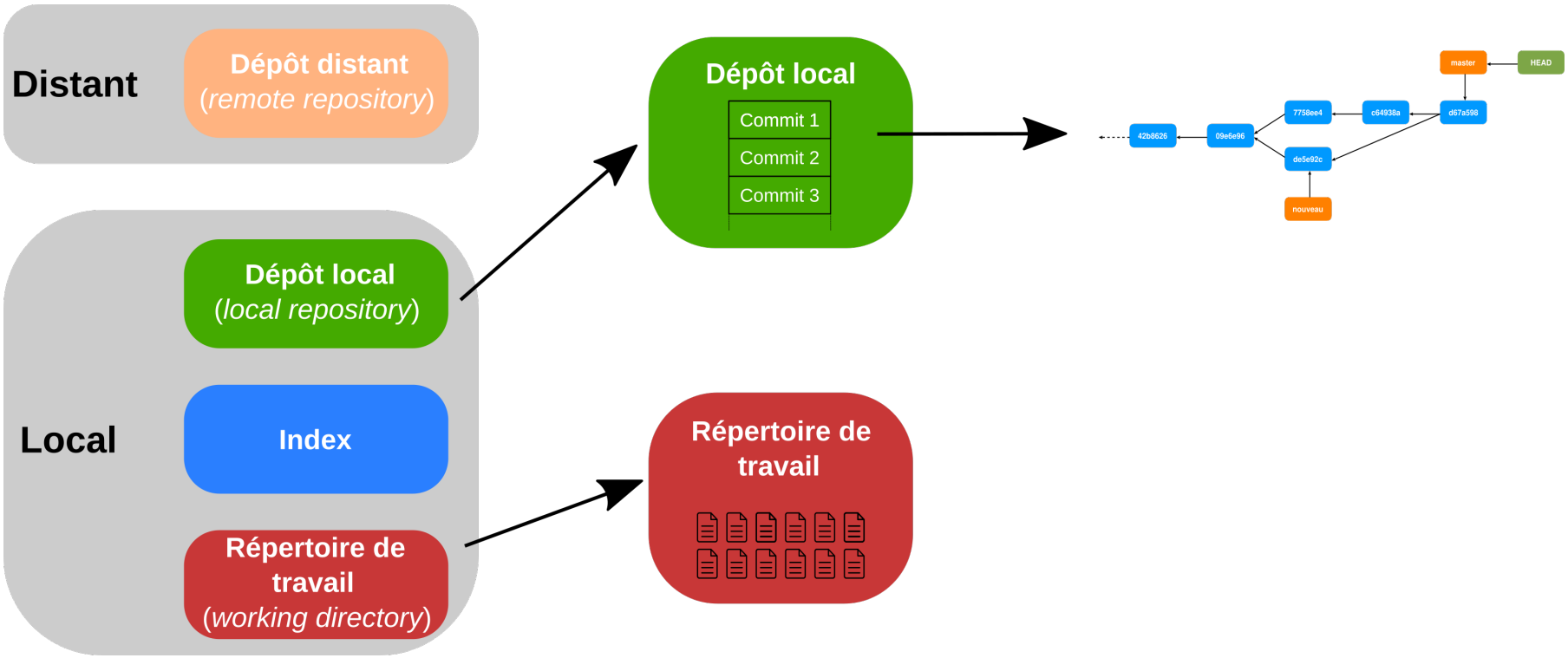
Possibilités et principes

- Travaille principalement sur les fichiers texte (`.txt`, `.c`, `.py`, `.html`, `.xml`...)
- Les fichiers binaires (`.jpg`, `.doc`, `.pdf`...) peuvent également être intégrés mais ne peuvent prétendre qu'au versionage, pas à l'édition collaborative
- Que faut-il stocker dans un dépôt ?
  - **toutes** les ressources nécessaires à la construction d'un projet...
  - ...**à l'exception** de celles qui sont générées automatiquement (`.o` en C, `.class` en Java...) et des données **sensibles**

Principe d'utilisation d'un dépôt (*repository*)

- Création d'un dépôt vide ou clonage d'un existant
- Alimentation du dépôt par l'intermédiaire de *commits*
- Un *commit* contient
  - un ensemble de modifications de données, suite aux manipulations des fichiers du projet (création, édition, suppression, renommage...)
  - un *log* associé : commentaire sur la nature des modifications
  - des méta-informations : identifiant de *commit*, auteur, date

Différents niveaux de stockage



Différents niveaux de stockage

- *Répertoire de travail*
  - contient la copie locale des sources du projet
  - contient, à sa racine, le répertoire `.git` (gestion interne technique)
- *Index*
  - espace temporaire utilisé pour préparer la transition de données entre le répertoire de travail et le dépôt local
  - permet de **choisir** quel sous-ensemble de modifications, présentes dans le répertoire de travail, répercuter dans le dépôt local lors d'un *commit*

Différents niveaux de stockage

- *Dépôt local*
  - contient la totalité de toutes les versions de tous les fichiers du projet, par l'intermédiaire des *commits*
  - contient toutes les méta-informations : historique, *logs*...
  - chaque contributeur possède son propre dépôt local
- *Dépôt(s) distant(s)*
  - est intrinsèquement similaire à un dépôt local
  - peut être
    - le dépôt local d'un autre utilisateur
    - un dépôt centralisé dédié : il est alors configuré et déployé pour pouvoir être partagé entre utilisateurs

SHA-1

Définition

- Fonction de hachage cryptographique conçue par la NSA
  - prend en entrée un texte de longueur maximale 2^{64} bits, soit environ 2.3×10^{18} caractères (~ 2.3 Eo)
  - produit une signature sur 160 bits, soit 20 octets, soit 40 caractères hexadécimaux ($\sim 1.5 \times 10^{48}$ possibilités)
  - très bonne répartition des hashs (signatures) produits
- Exemples

```
% echo salut | sha1sum
3775e40fbea098e6188f598cce2a442eb5adfd2c -
% echo Salut | sha1sum
06d046c7fefde2a0514cb212fd28a5a653d8137e -
```

SHA-1

Signatures, aspects mathématiques

- Un *même* contenu fournit toujours la *même* signature
- Mathématiquement, il est possible que deux contenus différents génèrent une même signature (une *collision*)
- En pratique, la probabilité est infinitésimale et est ignorée
- D'ailleurs, les 7 ou 8 premiers caractères d'une signature sont quasi systématiquement suffisants pour désigner sans ambiguïté un contenu...

SHA-1

À retenir

Important !

- Le SHA-1 est un *hash* tenant sur 40 caractères hexa
- Propriété : un même contenu est toujours associé au même hash
- Ce hash est réputé suffisamment discriminant pour être utilisé comme *identifiant*

Usage des signatures SHA-1

- Sous Git, les signatures SHA-1 permettent d'identifier les contenus
  - de fichiers
  - de versions d'un projet (à travers ses fichiers)
  - de *commits* (en y associant des infos relatives à leur auteur)
- À chaque fois, la signature obtenue est supposée unique et constitue un identifiant fiable
- Cette gestion de signatures est à l'origine des performances de Git
- Elle lui permet aussi de garantir l'intégrité d'un projet dans un contexte distribué

Sommaire

Les commandes de `git`

- `git` est un *toolkit*
- `git`, c'est plus de 150 commandes différentes
  - ≈ une cinquantaine de haut-niveau (*porcelaine*)
  - ≈ une centaine de bas-niveau (*plomberie*)
- En pratique, une trentaine de commandes suffisent à un usage « classique » (une quinzaine au quotidien)
- Elles sont utilisables en ligne de commande, mais également interfacées à la quasi-totalité des IDE
- Cela autorise de nombreux *workflows* d'utilisation, même si seulement 2 ou 3 sont très populaires (et suffisants pour la plupart des usages), comme les *pull requests*

Principales commandes, par thème

Création

```
git init
git clone
```

Ajout

```
git add
git commit
```

Interrogation

```
git log
git show
git status
git diff
```

Opérations

```
git restore
git mv
git rm
git tag
```

Synchronisation

```
git push
git pull
```

Récapitulatif des commandes fréquentes

- `git init` : création d'un dépôt local vide
- `git clone` : création d'un dépôt local à partir d'un dépôt existant (local ou distant)
- `git add` : « indexe » des fichiers en prévision d'un *commit*
- `git commit` : répercute les changements de l'index dans le dépôt local, sous forme d'un *commit*
- `git log` : examine l'historique du projet
- `git show` : affiche un objet (un *commit* par exemple)
- `git status` : affiche le status du répertoire de travail

Récapitulatif des commandes fréquentes

- `git diff` : affiche les différences entre le répertoire de travail et l'index
- `git tag` : associe une balise à un *commit*
- `git restore` : supprime des modifications effectuées dans l'index ou le répertoire de travail
- `git mv` : déplace des fichiers
- `git rm` : supprime des fichiers
- `git pull` : répercute les changements du dépôt distant vers le dépôt local
- `git push` : répercute les changements du dépôt local vers le dépôt distant

Sommaire

Configuration utilisateur

À faire une fois sur tout compte informatique, valable ensuite pour tous les projets édités à partir de ce compte

- Positionnement du nom utilisateur
  `git config --global user.name "Philippe Dosch"`
- Positionnement de l'adresse mail
  `git config --global user.email "dosch@loria.fr"`
- Sorties en couleurs
  `git config --global color.ui "auto"`
- Choix de l'éditeur par défaut
  `git config --global core.editor "chemin vers l'éditeur"`
  (exemple de chemin sous Windows pour Atom : `C:/Program Files (x86)/atom/atom.exe --wait`)

Fichiers à ignorer

- Lors de commandes du type `git status` affiche des avertissements sur les fichiers qui n'ont jamais été indexés
- Et certains fichiers ne sont jamais intégrés dans un projet (les fichiers temporaires, les résultats de compilation, les sauvegardes...)
- D'autres fichiers, nécessaires au bon fonctionnement d'un projet, **ne doivent pas être intégrés**
  - les fichiers contenant des informations confidentielles, tels que les **logins et mots de passe**, fichiers de configuration...
  - ne pas les intégrer et expliquer au besoin dans le fichier `README` comment créer ces ressources

Fichiers à ignorer

- Il est possible d'indiquer à Git d'ignorer ces fichiers
  - par un fichier `.gitignore`, à placer typiquement à la racine du projet : ce fichier sera partagé avec les autres membres du projet (*généralement plus intéressant*)
  - grâce du fichier `.git/info/exclude` : fichier propre au projet, mais qui ne sera pas partagé avec les autres membres du projet
- La commande `git ignore` permet également d'alimenter ces fichiers
- Un dépôt Github regroupe des *templates* de `.gitignore` (par langage / framework)
  `https://github.com/github/gitignore`

Sommaire

Introduction

- De nombreuses solutions d'hébergement de dépôts existent
- Elles procurent des accès à des dépôts distants centralisés, toujours disponibles, même si on peut travailler en P2P avec `git`...
- Les plus populaires sont Github, Bitbucket, Framagit...
- Avec Gitlab, il est aussi possible d'auto-héberger une *forge*
- L'université de Lorraine a déployé une instance Gitlab
- Suivant les solutions, connexion en SSH, en HTTPS ou par *token*

Sommaire

Présentation (rapide) de Gitlab

- Gitlab : logiciel libre de *forge* basé sur `git`
- Propose des outils : wiki, gestion des bugs, documentation, CI/CD...
- Utilisé par de nombreux laboratoires et universités, comme par exemple
  - Université de Lorraine
  - INRIA
- Nombre d'utilisateurs estimé à 30 millions, déployé par environ 100.000 organisations
  (source : `https://about.gitlab.com/company/`)

Présentation du Gitlab UL

- Doc : `https://numerique.univ-lorraine.fr/catalogue-des-services/gitlab-forge-git`
- Lien : `https://gitlab.univ-lorraine.fr/`
- Lancé il y a quelques années, en phase de « rodage », pleinement fonctionnel (montée en puissance planifiée)
- Accessible à tout membre (personnel, étudiant) de l'université (interfacé LDAP)
- Possibilité d'inviter des extérieurs (création d'*invités numériques*, accessible à tout personnel UL)

Caractéristiques du Gitlab UL

- Pour les membres (personnels, étudiants)
  - limite à 5 dépôts (pour le moment)
  - quota de 10 Go par dépôt
- Possibilité de créer un *groupe de projets* (composante / projet / laboratoire)
  $\rightarrow$ plus de limite sur le nombre de projets
- Authentification
  - connexion SSH non supportée pour le moment (en attente de la nouvelle infrastructure prévue)...
  - ...mais support des *tokens*, solution à la fois plus fine et plus souple (propres à un site, facilement révocables)
  - voir *Préférences / Access tokens*

Sommaire

Inscription et configuration

- *Github* : `https://github.com/`
- Inscription : choisir un compte gratuit (autorise les dépôts publics et privés illimités, voir `https://github.com/pricing`)
- Ajouter sa clé SSH publique en passant par *Settings / SSH and GPG keys* (activer un `ssh-agent` !)
- Au besoin, la créer : la commande `ssh-keygen` (*accepter les propositions par défaut si c'est la première créée*)
  - sous Linux / Mac : commande installée en standard
  - sous Windows : passer par un *shell* `gitbash`, à installer au préalable (et à utiliser pour les commandes `git` à la place de l'interpréteur de commandes standard)

Sommaire

Sommaire

Outils liés à Git

- `gitk` : un navigateur graphique de dépôt Git (*i.e.* lecture)
- `gitstats` : un outil de génération de statistiques pour dépôt Git
- `gource` : visualisation, sous forme d'une animation, de l'évolution d'un dépôt (inutile, donc rigoureusement indispensable)
- et plein d'autres, `git` étant devenu un standard de fait pour le développement

Sommaire

Vidéos Youtube d'initiation à Git

Public : DUT informatique (Nancy-Charlemagne)

- *Git : principes et utilisation de base*
  `https://youtube.com/watch?v=CrMVRQFMeyU`
- *Live coding Git : usages avec un seul utilisateur*
  `https://youtube.com/watch?v=3thnpVwLyMc`
- *Live coding Git : usage avec deux (et plus) utilisateurs*
  `https://youtube.com/watch?v=sbdqRBsXt5k`
- *Git : fin du cours et création de paire de clés SSH*
  `https://youtube.com/watch?v=CkVPVcaePVs`

Liens généraux

- *Homepage* : `http://git-scm.com/`
- *Livre en français* : `http://git-scm.com/book/fr`
- *Gitlab UL* : `https://gitlab.univ-lorraine.fr/`
- *Github* : `https://github.com/`
- *Bitbucket* : `https://bitbucket.org/`
- *Cette présentation* : `https://gitlab.univ-lorraine.fr/dosch5/git-jso`

Liens divers

- *Git interactif* : `http://ndpsoftware.com/git-cheatsheet.html`
- *Git explorer* : `https://gitexplorer.com/`
- *Apprendre Git* (les bases, dans un navigateur) : `https://try.github.io`
- *LearnGitBranching* (pour apprendre à gérer des branches) : `http://pcottle.github.io/learnGitBranching/`
- *Gource* (animation d'un projet sous forme d'arbre) : `https://github.com/acaudwell/Gource`
- *Git LFS* (stockage de fichiers volumineux) : `https://git-lfs.github.com/`