Git

Principes et utilisation

Philippe Dosch

Philippe.Dosch@loria.fr

31 janvier 2012

Sommaire

iut Nancy-Charlemagne Informatique

Sommaire

Problématique générale

- Comment gérer l'historique des fichiers sources 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, de correction de bugs, d'ajout de fonctionnalités...
  - ...

Problématiques spécifiques au travail en groupe

- Comment partager les sources ?
- Comment travailler *ensemble* sur les sources ?
- Comment travailler *au même moment* sur les sources ?
- Comment réconcilier les changements des contributeurs ?
- Comment ne pas perdre de travail ?

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

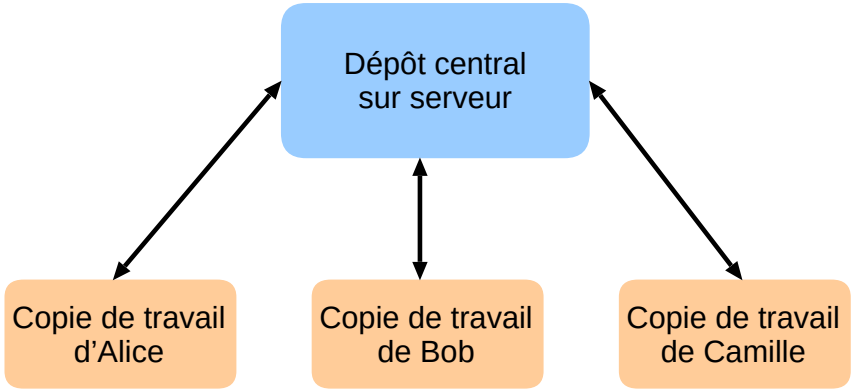
Ensemble de méthodes et d'outils qui maintiennent les différentes versions d'un projet à travers tous les fichiers qui le composent

- permet le développement *collaboratif* et *simultané*
- permet de garder tout l'historique de tous les fichiers
- permet le développement parallèle : version stable, de développement, introduction de fonctionnalités, correction de bugs...
- permet de savoir pourquoi, quand et par qui une portion spécifique de code a été introduite

Historique des systèmes de gestion de version

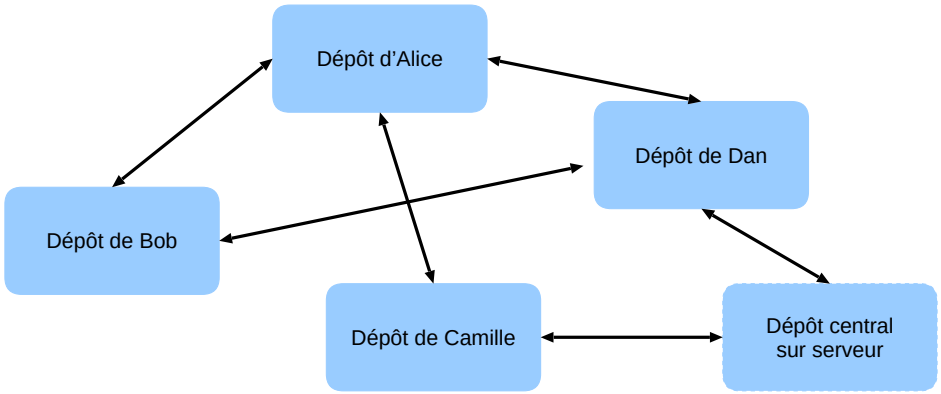
- Systèmes de gestion de version *centralisés*
  - CVS (*Concurrent versions system*), 1990
  - SVN (*Subversion*), 2004
- Systèmes de gestion de version *décentralisés*
  - BitKeeper, 1998
  - Mercurial, 2005
  - Git, 2005
  - et d'autres : GNU Arch, Bazaar, Monotone, Darcs...

Systèmes de gestion de version centralisés (CVCS)



- Le serveur détient tout l'historique du projet
- Les utilisateurs possèdent seulement une copie des fichiers correspondant au code
- Toutes les opérations de gestion sont réalisées par l'intermédiaire du serveur (*i.e. online*)
- Les échanges de code sont obligatoirement effectués grâce au serveur

Systèmes de gestion de version décentralisés (DVCS)



- Chaque utilisateur possède un *dépôt* local complet, contenant tout l'historique du projet
- Les opérations de gestion sont réalisées localement (*i.e. offline*)
- Du code *peut* être échangé avec d'autres utilisateurs sans serveur centralisé (mail typiquement)
- Des serveurs *peuvent* aussi assurer les échanges de code

Git

- Créé en 2005 par Linus Torvalds pour la gestion des sources de Linux, en remplacement de BitKeeper
- Part de marché parfois estimée à 90% sur le segment des DVCS utilisés par la communauté logiciel libre
- Exemples de projets gérés : Linux (!), Gnome, Eclipse, KDE, X.org, Qt, Perl, Debian, Android...

Sommaire

Possibilités

- Les VCS travaillent principalement sur les fichiers texte (`.txt`, `.c`, `.java`, `.xml`...)
- Les fichiers binaires (`.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

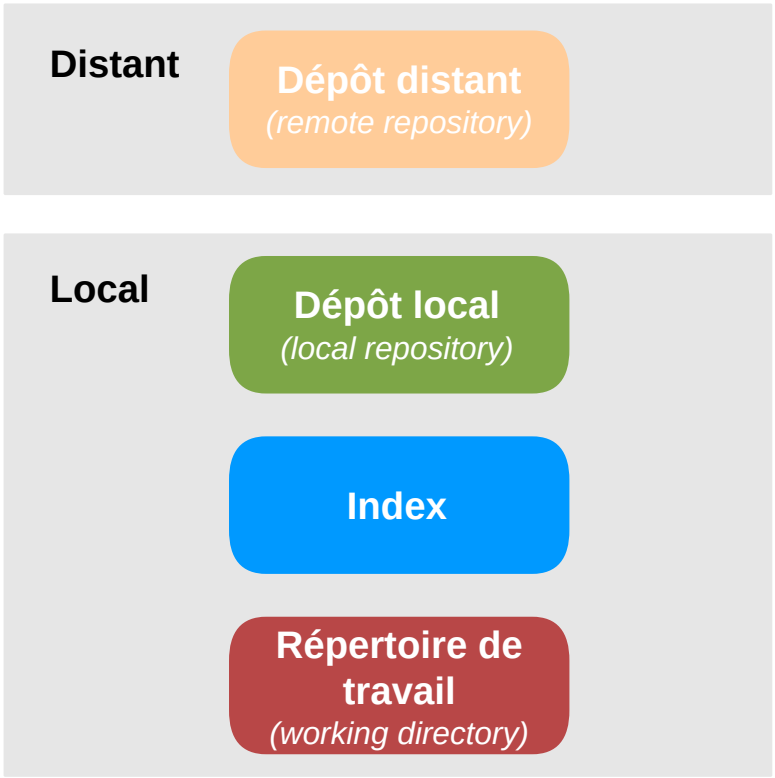
Usages

- Utiliser un VCS suppose que les développeurs travaillent en concertation !
- Les VCS supposent que les développeurs ne modifient pas la même partie d'un même fichier
- Les VCS peuvent fusionner deux modifications relatives à un même fichier si elles concernent des parties différentes
- Dans le cas contraire, un *conflit* est généré et doit être réglé manuellement (par les développeurs)

Principe de fonctionnement d'un dépôt

- Création d'un dépôt (*repository*) vide
- Alimentation du dépôt par l'intermédiaire de *commits*
  - ensemble de modifications de données, suite aux manipulations des fichiers du projet (création, édition, suppression, renommage...)
  - *log* associé : commentaire sur la nature des modifications
  - 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` de configuration
- *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*, *tags*...
  - propre à un utilisateur donné
- *Dépôt distant*
  - est intrinsèquement similaire à un dépôt local
  - configuré et déployé pour pouvoir être partagé entre utilisateurs

Principe de fonctionnement intrinsèque

- Contrairement à d'autres VCS, Git s'intéresse aux **contenus**, pas aux fichiers en tant que tels
- Les noms de fichiers, les dates de modification, n'interviennent donc pas directement pour déterminer les modifications réalisées depuis un *commit* donné
- Git calcule pour chaque fichier une *signature* SHA-1 lui permettant de détecter des changements de contenu
- Les noms de fichiers, les dates associées, ne sont considérées que comme des méta-informations

SHA-1
Définition

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

```
% echo salut | sha1sum
```

```
3775e40fbea098e6188f598cce2a442eb5adfd2c -
```

```
% echo Salut | sha1sum
```

```
06d046c7fefde2a0514cb212fd28a5a653d8137e -
```

SHA-1
Signatures, aspects mathématiques

- Un *même* contenu fourni toujours la *même* signature
- D'un point de vue mathématique, il est possible que deux contenus différents génèrent une même signature (une *collision*)
- Mais en pratique, la probabilité est infinitésimale et peut être ignorée sans risque
- D'ailleurs, les 7 ou 8 premiers caractères d'une signature sont quasi systématiquement suffisants pour désigner sans ambiguïté un contenu...

SHA-1
Collisions et probabilités

- Il faudrait que 10 milliards de programmeurs fassent 1 *commit* par seconde pendant presque 4 millions d'années pour qu'il y ait 50% de chance qu'une collision se produise
- « *Il y a plus de chances que tous les membres d'une équipe soient attaqués et tués par des loups dans des incidents sans relation la même nuit* »

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

Liste 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
- `git diff` : affiche les différences entre le répertoire de travail et l'index

Liste des commandes fréquentes

- `git reset` : supprime des modifications effectuées dans l'index ou le dépôt local
- `git mv` : déplace des fichiers
- `git rm` : supprime des fichiers
- `git blame` : affiche l'auteur et la révision de chaque ligne d'un fichier
- `git push` : répercute les changements du dépôt local vers le dépôt distant
- `git pull` : répercute les changements du dépôt distant vers le dépôt local

git init

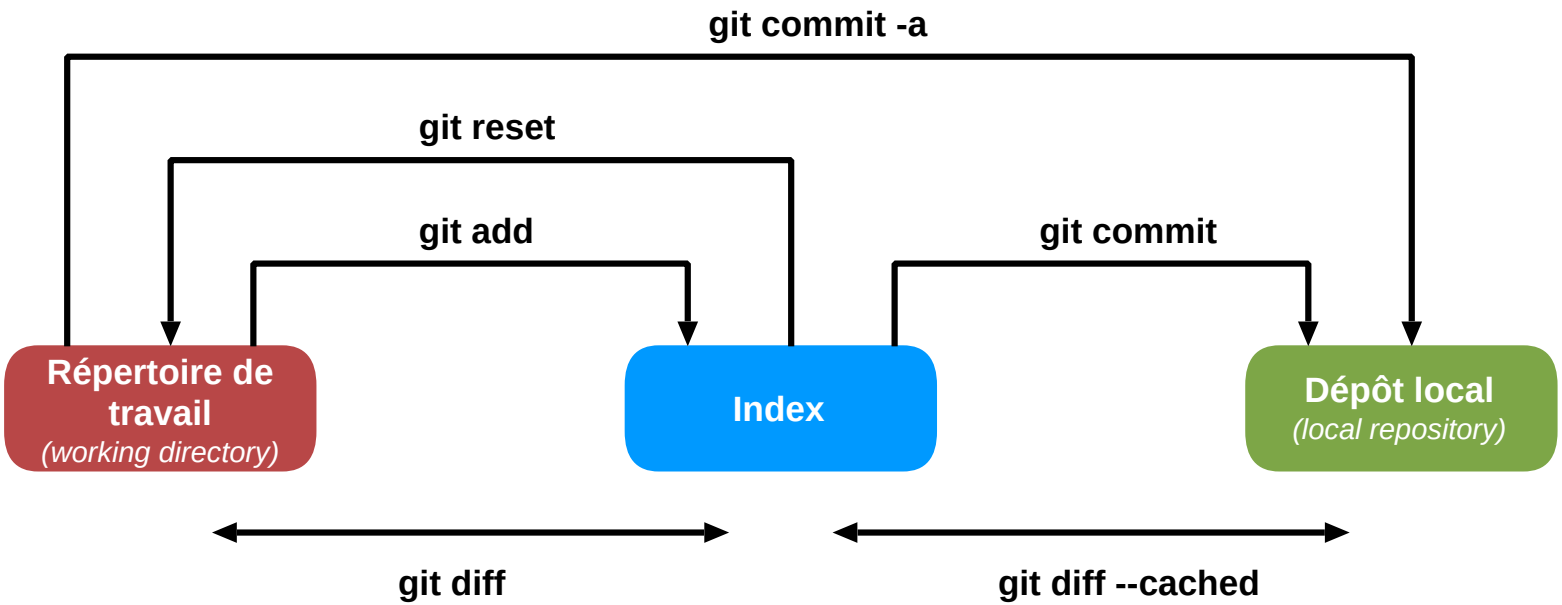
- Création d'un dépôt local vide
- Peut suffire pour gérer l'historique d'un projet pour un seul utilisateur...
- Crée une *branche* par défaut, appelée `master`
- Penser à ajouter un fichier `README` décrivant succinctement le projet
- `% git init`

```
Initialized empty Git repository in /home/phil/tmp/.git/
```

`git clone`

- Création d'un dépôt local à partir d'un dépôt existant (local ou distant)
- Met à jour la configuration du dépôt local pour garder une référence vers le dépôt distant
- Permet ensuite la communication entre les deux dépôts, typiquement par le biais des commandes `git push` et `git pull`

Index et commandes Git



`git add`

- Indexe le *contenu* des fichiers du répertoire courant passés en paramètre
- Rappel : Git travaille sur les contenus, pas sur les fichiers
- Conséquence : si des fichiers sont modifiés après leur indexation, c'est la version indexée qui sera répercutée dans le dépôt (et donc pas celle du répertoire courant)
- Un fichier qui a été indexé au moins une fois est ensuite suivi par Git (typiquement par `git status`)
- Mais l'indexation de chaque nouvelle version de ce fichier doit être réalisée par un nouveau `git add`

`git add`

- Il n'est pas nécessaire d'indexer en seule fois tous les changements d'un projet
- On peut donc typiquement utiliser `git add` sur un sous-ensemble des fichiers concernés
- Cela permet de créer par la suite des répercutions (*commits*) séparées

`git commit`

- Répercute le contenu de l'index dans le dépôt local
- L'index est ainsi complètement vidé suite au *commit*
- Un message de *commit* doit obligatoirement être défini à cette occasion
  - `git commit` : un éditeur externe sera lancé pour la saisie du message
  - `git commit -m "xxx"` : le message est fourni en ligne de commande

Que mettre dans un log ?
Techniquement...

- Une première ligne (obligatoire)
  - synthétise les changements
  - apparaît comme description courte du *commit*
- Une ligne vide (facultative si pas de description longue)
- Une description longue (facultative), de taille arbitraire

Que mettre dans un log ?

Et dans l'intention...

- Fondamentalement, doit expliquer le « pourquoi » d'un *commit*
- Trouver un « bon » message de log s'apparente à un exercice de style, presque un art...
- Intuitivement, doit être proche d'un résumé ($\sim$ une phrase) que l'on pourrait faire à un *collègue* (initié donc !)
- Exemples
  - Remplacement de conditionnelles imbriquées en `switch` pour améliorer la lisibilité (forme)
  - Suppression de la fonctionnalité DDFD_08 entravant la stabilité du code (fond)

`git commit`

- La commande `git commit -a` permet
  1. d'indexer automatiquement tous les fichiers qui ont déjà été indexés au moins une fois
  2. de répercuter l'index dans le dépôt local
- Les fichiers qui n'ont jamais été indexés (typiquement, les nouveaux fichiers du projet) ne peuvent donc pas être concernés

`git log`

- Affiche l'historique des *commits* du projet dans l'ordre chronologique inverse
- Affiche, pour chaque *commit*, son identifiant, l'auteur, la date et la première ligne du log
- `git log commit1...commit2` : affiche les logs entre 2 *commits* spécifiques (le premier commit fourni doit être le plus récent)

`git show`

- Affiche le détail d'un *commit* (ou d'autres entités Git)
- L'identifiant (court / long) correspondant doit être fourni en paramètre
- Sur un *commit*, `git show` affiche en particulier la différence de contenu avec le *commit* précédent

git status

- Affiche des informations sur l'état du répertoire de travail et de l'index
- Permet de savoir ce que contient l'index (et donc ce qui sera concerné par le prochain *commit*)
- Permet de savoir quels fichiers sont suivis par Git et quels sont ceux qui ne le sont pas

`git diff`

- Affiche les différences de contenu entre le répertoire de travail et l'index
- `git diff commit1...commit2` : affiche les changements de contenus entre 2 *commits* spécifiques (le premier commit fourni doit être le plus récent)

git reset

- Supprime des modifications effectuées dans l'index ou le dépôt local
- À utiliser avec précaution, certaines suppressions deviennent irrévocables...
- Peut parfois être remplacé avantageusement par un nouveau *commit*...

git mv

- Permet de déplacer ou de renommer un fichier ou répertoire
- L'historique de la ressource concernée est alors conservé
- À utiliser plutôt qu'un simple `mv` système qui ne permet pas la conservation de l'historique
- Le changement est répercuté dans l'index (et nécessite ensuite d'être répercuté par un *commit*)

git rm

- Efface un fichier ou un répertoire physiquement, ainsi qu'au niveau du suivi Git
- À utiliser plutôt qu'un simple `rm` système qui n'informerait pas Git de la suppression
- Le changement est répercuté dans l'index (et nécessite ensuite d'être répercuté par un *commit*)

git push

- Propage les changements du dépôt local vers le dépôt distant

`git pull`

- Récupère les changements du dépôt distant et les fusionne dans le dépôt local et le répertoire de travail
- Peut d'ailleurs être utilisé pour récupérer des changements de n'importe quel dépôt distant...
- À utiliser avant de propager les changements du dépôt local vers le dépôt distant (`git push`) s'il y eu des changements sur le dépôt distant

Sommaire

Fichiers de configuration

- *Au niveau projet*
  Fichier `.git/config` à la racine du projet
- *Au niveau utilisateur*
  Fichier `~/.gitconfig`
- *Au niveau système*
  Fichier `/etc/gitconfig` (rarement utilisé)

Configuration utilisateur

- 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"
  ```

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...)
- Il est possible d'indiquer à Git d'ignorer ces fichiers
  1. par un fichier `.gitignore`, à placer à la racine du projet : ce fichier pourra être suivi et partager avec les autres membres du projet
  2. grâce du fichier `.git/info/exclude` : fichier propre au projet, mais qui ne sera pas partagé avec les autres membres du projet

Fichiers à ignorer

- Quel que soit le fichier utilisé, la syntaxe est la même
- On peut y placer des noms de fichiers (un par ligne)
- Le caractère $*$ est autorisé, permettant de désigner facilement des familles de fichiers (typiquement sur l'extension)

Sommaire

Inscription et configuration

- *Github* : `https://github.com/`
- Inscription : choisir un compte gratuit
- Ajouter sa clé RSA publique (au besoin, la créer, les indications sont fournies) en passant par l'icône *Account settings*

Configuration

- À l'initiative d'une seule personne, le créateur du projet
- Création d'un dépôt public avec un simple nom
- Suivre les indications fournies par le site !
- Effectuer la configuration globale sur votre machine (si non déjà effectuée)
- Réaliser les étapes d'initialisation du dépôt décrites
- Le dépôt est créé (vérifier sous Github)
- Ajouter les collaborateurs (*Admin / collaborators*)

Rallier un projet existant

- Recommandé, y compris pour le créateur du projet
- Aller sur la page consacrée au projet
- Choisir le protocole SSH parmi ceux proposés
- Copier l'URL associée
- Taper

```
% git clone test git@github.com:dosch/test.git
```

```
Cloning into test...
remote: Counting objects: 54, done.
remote: Compressing objects: 100% (54/54), done.
remote: Total 54 (delta 20), reused 0 (delta 0)
Receiving objects: 100% (54/54), 11.25 KiB, done.
Resolving deltas: 100% (20/20), done.
```

Liens

- *Homepage* : `http://git-scm.com/`
- *Livre en français* : `http://progit.org/book/fr/`
- *Github* : `https://github.com/`