Faculté des Sciences et Ingénierie
Master Informatique
Systèmes Électroniques, Systèmes Informatiques

NMV

Noyau, mémoire et virtualisation

Adrien Bourmault
(adrien.bourmault@etu.upmc.fr)

Enseignant : Julien SOPENA

Table des matières

Organisation

Pas de projet, pas de partiel : la notation se fait avec un examen final (50%) et un exposé (50%), dont la soutenance aura lieu dix jours après l'examen.

Chapitre 1

Fonctionnement interne de git

1.1 Introduction

Pourquoi git ? C'est l'outil Nř1 pour la gestion de version et de projet décentralisé. Ce logiciel permet d'avoir un historique sur les fichiers, de maintenir des branches alternatives. Le fait qu'il soit décentralisé, contrairement à SVN et CVS, implique une gestion de la cohérence sur l'historique.

L'intérêt de ce cours est d'étudier le fonctionnement interne de git.

1.2 Principes de base

Tous les mécanismes de git reposent sur les diff/patchs et

1.2.1 Patchs et diffs

Un diff est une comparaison ligne à ligne de fichiers. Le patch est l'utilisation d'un diff pour faire évoluer un fichier et passer à une nouvelle version.

1.2.2 Notion d'historique

1.2.2.1 Définition : historique

On appelle historique un graphe orienté acyclique composé de version re-calculables à partir des versions adjacentes.

Rappel : un arbre est un graphe connexe acyclique non orienté. L'historique n'est pas un arbre car il est orienté.

1.2.2.2 Définition : branche

On appelle branche de version v_i d'un historique le sous-graphe composé de l'ensemble des noeuds adjacents.

1.2.2.3 Définition : tronc

Le tronc ou branche principale est définie selon le contexte.

1.2.2.4 #BalanceTonDépôt

Depuis le 1er octobre 2020, la branche principale des dépôts créés sur les plateforme est nommée main au lieu de master.

Le logiciel git émet un avertissement lors du git init.

1.2.2.5 Définition : sous-branche

Toute branche d'un noeud non incluse dans une autre branche est nommée sous-branche.

1.2.2.6 Définition : merge

Tout noeud du graphe ayant un degré sortant strictement supérieur à 1 est un merge (ou noeud de fusion).

1.3 Historique

— 2001 : Linux est développé sur CVS
— 2002 - 2005 : Linux est développé sur Bitkeeper
— 6 avril 2005 : Bitkeeper quitte le libre et Torvalds commence git
— 18 avril 2005 : git sait merge
— 16 juin 2005 : Linux développé officiellement sur git
— 14 février 2007 : Sortie de la version 1.5.0

1.4 Architecture interne

À l'origine, git était destiné à constituer un système de fichiers !

1.4.1 Création d'un dépôt

À la création d'un dépôt, un répertoire .git est créé.

Sur un serveur, une autre méthode est possible : il n'y a besoin que de l'historique (et pas des fichiers eux-mêmes). L'option faite pour cela est `-bare`

1.4.2 Les objets : tout est blob

Le système de git est conçu comme un système de fichiers versionnés. Linus Torvalds reprend les concepts de Linux :
— tout est fichier -> tout est blob ;
— fichiers identifiés par numéro d'inode -> blobs identifiés par SHA-1 de leur contenu.

L'identification par *hash* n'est possible que si le contenu est fixe : un blob est immuable, entraînant une réduction de la taille des dépôts par déduplication !

1.4.2.1 SHA-1

C'est une fonction cryptographique conçue par la NSA, limitée à des fichiers de 2 exaoctets, retournant un *hash* sur 160 bits noté avec 2 + 38 caractères hexadécimaux.

Note : bien que largement vulnérable aux collisions, SHA-1 ne pose pas de problèmes pour les fichiers de texte a priori (probabilité de $\frac{1}{2^{160}}$).

Note : pour vérifier l'intégrité, la commande `git fsck` est utile !

1.4.3 Types d'objets

1.4.3.1 Blob

C'est l'élement de base du stockage. Il est identifié par le SHA-1, stocké dans le dépôt sous `.git/objects/XX/YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY` sous la forme `"blob <taille de contenu>\0<contenu>"`

Les blobs sont compressés avec la zlib. Le SHA-1 est calculé sur le blob et non sur le contenu !

1.4.3.2 Trees

Ce sont des blobs. Ils stockent la liste des pointeurs vers des blobs et d'autres trees en associant un nom à chaque pointeur. Un ensemble de trees permet de décrire l'état d'une hiérarchie de blobs.

1.4.3.3 Commits

Un commit est l'enregistrement d'une nouvelle version de fichiers. Auparavant, dans CVS par exemple, on utilise la numération des patchs fichier par fichier (deltas, on a donc pas d'état global et ça oblige à ajouter des tags pour retrouver un état cohérent, l'accès à une version spécifique nécessite de ré-appliquer les patchs). Dans SVN, chaque numéro correspond à un état global cohérent mais là aussi il faut ré-appliquer les patchs.

La résolution de ce problème par git consiste à numéroter globalement tous les fichiers, puisque la déduplication rend l'opération atomique.

Ce sont des blobs. Ils un pointeur vers un tree dont on veut sauver l'état, un pointeur vers un ou plusieurs autre commits pour constituer un historique, et également le nom d'un auteur et d'un committeur, puis une description sous forme d'une chaîne de caractère.

ATTENTION : les blobs de commit contiennent des références de date en notation Epoch, rendant le hash liée à la date de création du commit. Ainsi même avec les mêmes fichiers et les mêmes modifications, personne n'aura le même hash de commit sauf à l'effectuer exactement au même moment.

1.4.3.4 Tags

Ce sont des blobs. Ils contiennent un pointeur vers un blob, un tree ou un commit, ainsi qu'une signature.

1.5 Architecture générale

1.5.1 Commandes

Ce logiciel est conçu avec la philosophie Unix des commandes. Ainsi, chaque commande git a son propre binaire ! Il y a 145 commandes, dont 19 sont dédiées au dépôts locaux, 5 pour les dépôts distants, et quelques autres pour gérer les conflits et les recherches d'erreurs.

1.5.2 Exemple simple

Lors de la création d'un projet, on utilise `git init` pour créer l'enveloppe du dépôt. Ensuite on ajoute un ou des fichiers avec `git add` crée un ou des blobs liés à un tree. En utilisant `git commit` on obtient alors un commit.

L'état courant est indiqué par `head` et l'index pointe vers le tree pointé par le commit pointé par `head`.

On remarque facilement que la structure de git a inspiré la *blockchain*.

1.5.3 Branches et fusion

Pour manipuler les branches on a :
— branch : liste les branches avec une * pour la branche active ;
— branch <nom> : créé une nouvelle branche <nom> ;
— branch -m : permet de renommer une branche ;
— branch -d : permet de supprimer une branche ;
— switch : permet de changer de branche active ;
— switch -c : créé une branche et l'active ;
— show-branch : affiche les branches et leur commits.
Créer une branche, c'est créer un pointeur vers un commit (comme un tag !). Si on réalise un commit sur une branche, on va créer un historique et la référence à la branche se déplacera.

Note : une bonne pratique est de créer une branche à chaque séance de travail et la fusionner (*fast-forward* !) en fin de session de travail.

La fusion en *fast-forward* est atomique (on déplace le pointeur de la branche). Par contre, si cela n'est pas possible parcequ'on a deux branches parallèles impliquées, il faut créer un nouveau commit dit *commit de fusion*. Pourtant, les modifications dans ces deux branches sont probablement indépendantes ! La solution : `git rebase`.

Le *rebase* est une transposition d'un commit de sa base initiale à une nouvelle base. Lors de cette transposition, de nouveaux commits sont créés. Les précédents ne sont pas effacés, sauf après utilisation de `git gc` (après 7 jours parceque Torvalds sait que les programmeurs sont des boulets).

1.5.4 Les remords

On a trois types de correction :
— revert : pour annuler un commit ;
— amend : modifier le dernier commit ;
— reset : rétablir la situation d'un ancien commit.
Le revert et le amend sont assez simples à comprendre.

Pour reset, on a trois modes `-hard` (restaure les fichiers tels qu'ils étaient avant), normal (laisse les fichiers modifiés dans leur état, non suivis) et `-soft` (laisse les fichiers modifiés dans leur état dans l'index, prêts à commit).

1.5.5 Utilisation de l'historique

La commande `git diff` permet de travailler sur l'index lui-même. On peut notamment obtenir les différences entre le répertoire de travail et l'index, `head` et l'index etc.

La commande `git show` donne le diff sous forme d'un patch (pour un commit).

1.6 Décentralisation

1.6.1 Dépôts centralisés vs décentralisés

Contrairement à SVN ou CVS, git est décentralisé. Cela signifie qu'il y a deux actions pour pousser une modification et également deux actions pour récupérer les modifications. De façon intéressante, `git pull` est la fusion entre `git fetch` et `git merge`. En tout cas, git permet de versionner localement et sans connexion internet avec un accès complet au passé.

1.6.2 Gestion des conflits

Le logiciel git permet de gérer les conflits localement lors des fusions, contrairement à SVN ou CVS qui rencontraient les conflits dès la récupération des modifications.

1.6.3 Vue de la branche distance

Sur un dépôt local, lors d'un clonage, le dépôt local contient une branche locale ainsi que la vue de la branche distante (généralement préfixée par *origin*).