RÉALISATION D'UN LOGICIEL DE GESTION DE VERSIONS (GIT)

SAMAHA ELIO
SOUAIBY CHRISTINA

LU2IN006 | SOUTENANCE AVRIL 2023

TABLES DES MATIÈRES

1. INTRODUCTION

Un logiciel de gestion de versions est un outil indispensable pour les professionnels de l'informatique qui travaillent sur des projets complexes ou collaboratifs. En particulier, le "Global Information Tracker" (ou GIT) est un système de contrôle de versions distribué très largement utilisé dans le développement de logiciels. C'est un logiciel libre et gratuit qui permet de gérer efficacement les différentes versions d'un projet, en offrant des fonctionnalités avancées telles que la fusion de branches, la gestion des conflits et la gestion des sauvegardes locales et distantes. Il est particulièrement apprécié pour sa rapidité, sa stabilité et sa flexibilité.

Le projet consiste à examiner en profondeur les structures de données impliquées dans le fonctionnement d'un logiciel de gestion de versions comme le GIT, et les interactions entre elles. Nous allons nous intéresser à l'exploration de plusieurs fonctionnalités essentielles telles que la création d'enregistrements instantanés, la navigation aisée entre les différentes versions du projet, ainsi que la construction et la maintenance d'une arborescence claire qui permet de suivre son évolution. Cette analyse approfondie des structures de données permettra également de mettre en évidence les avantages et les limites du logiciel de gestion de versions, ainsi que les bonnes pratiques à adopter pour l'utiliser de manière optimale.

2. STRUCTURES MANIPULÉES ET DOCUMENTATION

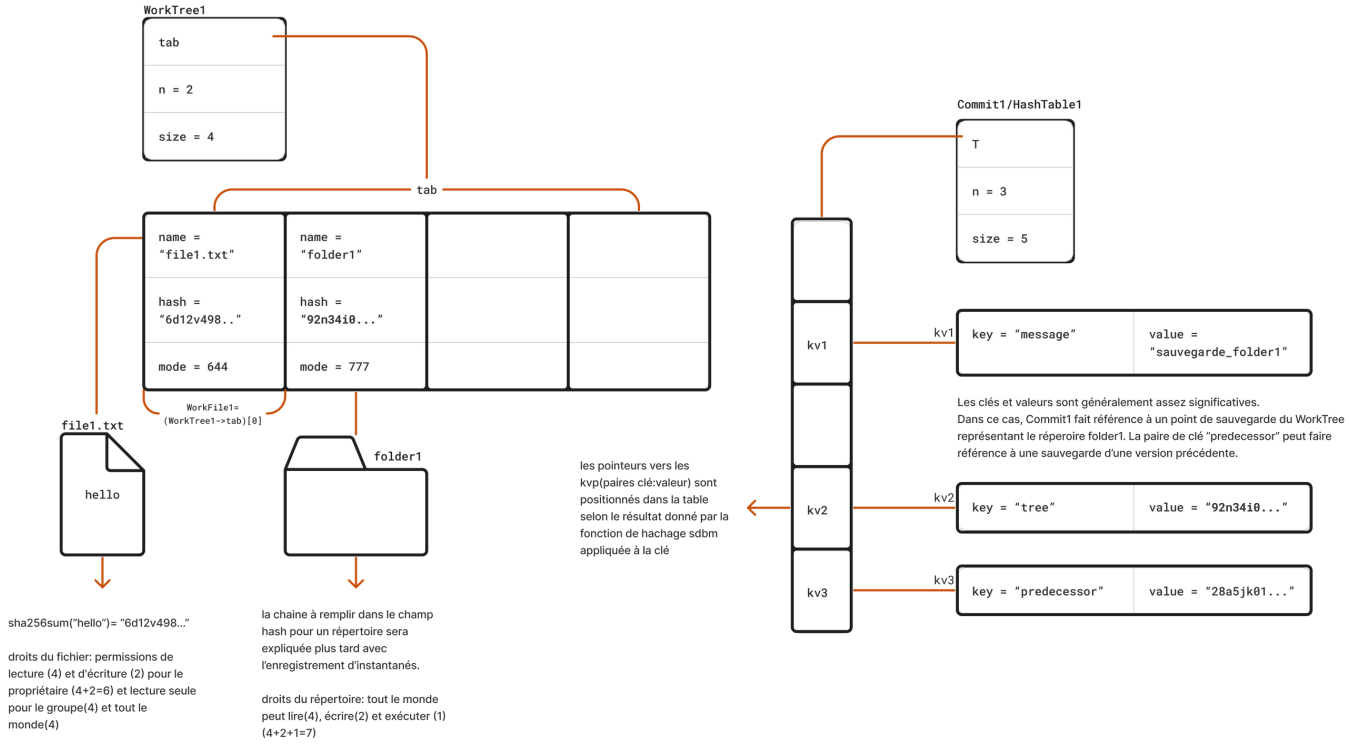
2.1. STRUCTURES MANIPULÉES

Trois structures principales (mises à part de simples listes chaînées) sont manipulées dans ce projet:

Un **WorkFile** contient 3 champs: 2 chaînes de caractères (name et hash), et un octal (mode).

Un **WorkTree** contient un tableau de WorkFile (tab), un champ pour la taille du tableau (size) et un compteur du nombre de cases remplies (n).

Un **Commit** est une table de hachage contenant un tableau (T) de pointeurs vers des structures (kvp) qui contiennent une paire de chaînes de caractères (key et value), ainsi que les champs n et size.



2. STRUCTURES MANIPULÉES ET DOCUMENTATION

2.2. DOCUMENTATION

Notre travail s'organise en 7 fichiers source (.c), un fichier header (.h), un Makefile et quelques fichiers/répertoires spécialement créés à des fins de test. Le header contient les signatures de fonctions et définitions de structures, le Makefile gère la compilation, l'édition de liens et les exécutables, et les fichiers sources contiennent respectivement :

Les fonctions qui servent essentiellement d'**outils** pour la suite: hachage du contenu d'un fichier et création du chemin correspondant, vérification de l'existence de fichiers/répertoires, copie de contenu d'un fichier vers un autre, désallocation de mémoire, etc.

Les fonctions qui permettent la **manipulation des WorkFile et WorkTree**: création et ajout de WorkFile dans un WorkTree, écriture du contenu des structures dans des fichiers et inversement, distinction entre fichier et répertoire, gestion des droits/modes, enregistrement des instantanés, restauration de versions des fichiers de l'arborescence de travail, etc.

Les fonctions de **gestion des Commit** : création, insertion de paires (clé:valeur) en fonction du hachage, recherche de clé dans la table, transformations en chaînes de caractères/fichiers et inversement, enregistrement, création et suppression de références, etc.

Les fonctions de **contrôle de l'arborescence**: création/suppression, affichage d'une branche, parcours et recherche de Commit dans les branches, filtrages à partir d'un motif, déplacement entre les branches, restauration de Commit, etc.

Les fonctions de **fusion** de branches et de WorkTree, de **gestion** et suppression **des conflits.**

Un main provisoire qui permet de **tester** le fonctionnement de toutes nos fonctions et de gérer l'allocation et la libération de la mémoire.

Un fichier myGit qui concrétise tout le projet et permet l'exécution de différentes **commandes** en faisant appel aux fonctions implémentées dans les autres fichiers.

3. FONCTIONS PRINCIPALES ET CHOIX D'IMPLÉMENTATION

3.1. OUTILS DE BASE

Dans notre progression, nous avons eu recours à plusieurs outils:

Le hachage:
La fonction "**sha256file**" prend en entrée le nom d'un fichier, crée un fichier temporaire, calcule le hachage SHA-256 du contenu fichier d'entrée et stocke le résultat dans le fichier temporaire. Elle retourne ensuite le hachage sous forme de chaîne de caractères.
La fonction "**hashToPath**" prend en entrée un hachage SHA-256 et génère un chemin à partir de celui-ci en séparant par "/" les deux premiers caractères et le reste du hachage. Cette fonction peut être utilisée pour stocker des fichiers de manière hiérarchique dans une structure de répertoires en fonction de leur hachage, ce qui peut faciliter leur gestion et leur recherche.

La recherche et la vérification d'existence de fichiers/répertoires:
La fonction "**searchList**" prend en entrée une liste chaînée et une chaîne de caractères, et retourne un pointeur sur l'élément de la liste qui contient cette chaîne de caractères, ou NULL si elle n'est pas présente. Elle utilise une boucle while pour parcourir la liste chaînée, et la fonction strcmp pour comparer les chaînes de caractères.
La fonction "**listdir**" permet de créer une liste chaînée de tous les fichiers et répertoires contenus dans un répertoire spécifié en entrée. Elle utilise la fonction opendir pour ouvrir le répertoire, puis la fonction readdir pour lire les fichiers et les répertoires un par un.
La fonction "**fileExists**" prend en entrée le nom d'un fichier et retourne 1 si le fichier existe et 0 sinon. Notre première implémentation faisait appel aux deux fonctions précédentes pour rechercher un fichier/répertoire dans le répertoire courant, mais nous avons choisi de la généraliser en utilisant la fonction stat. La structure "stat" contient les informations du fichier, et si la fonction stat renvoie 0, cela signifie que le fichier existe et on retourne 1.

3. FONCTIONS PRINCIPALES ET CHOIX D'IMPLÉMENTATION

La copie de contenu d'un fichier vers un autre:
La fonction "**cp**" permet de copier le contenu d'un fichier source situé à l'emplacement "from" vers un autre fichier destination situé à l'emplacement "to" passé en paramètre. Elle ouvre les deux fichiers respectivement en mode lecture et écriture. En cas d'erreur lors de l'ouverture, un message d'erreur est affiché. La fonction copie chaque ligne du fichier source dans le fichier de destination. Elle utilise la fonction "getChmod" pour récupérer les permissions du fichier source et "setMode" pour donner les mêmes permissions au fichier destination.
Les fonctions "getChmod" et "setMode" fournies retournaient un mode décimal plutôt qu'un octal. Dans notre implémentation nous avons décidé de le transformer en octal pour éviter toute incohérence sur les droits d'accès aux fichiers.
Cette fonction est utile pour la suite dans le cadre de la gestion de fichiers et de la création de copies de sauvegarde.

3.2. ENREGISTREMENT D'INSTANTANÉS

L'enregistrement et la restauration de versions se font à travers 3 fonctionnalités principales: blob, save et restore, appliquées à des fichiers, des WorkTree ou des Commit.
La fonction "**blobFile**" prend en entrée le chemin d'un fichier et le transforme en un objet "blob" dans le système de gestion de versions Git. Pour cela, elle calcule le hachage SHA-256 du contenu du fichier à l'aide de "sha256file", le transforme en une chaîne de caractères représentant un chemin d'accès grâce à "hashToPath", crée un répertoire portant les deux premiers caractères du hash s'il n'existe pas avec "mkdir", puis y copie le fichier sous le nom correspondant au hash complet, et cela en utilisant "cp". Enfin, elle libère la mémoire allouée pour les chaînes de caractères temporaires utilisées dans le processus.
Les fonctions "**blobCommit**" et "**blobWorkTree**" commencent par écrire le contenu de la structure dans un fichier temporaire, ensuite par ajouter l'extension adéquate au chemin créé (.t pour WorkTree, .c pour Commit) puis procèdent selon le même principe que "blobFile" pour la suite.

3. FONCTIONS PRINCIPALES ET CHOIX D'IMPLÉMENTATION

La fonction "**saveWorkTree**" a pour but de sauvegarder la version actuelle de tous les fichiers du WorkTree dans l'arborescence du système de fichiers local. Elle prend en entrée un pointeur vers un WorkTree et un chemin vers le répertoire racine de ce WorkTree.
Elle commence par parcourir le tableau de WorkFile. Pour chaque WorkFile, la fonction concatène le chemin passé en entrée avec un "/" suivi du nom du fichier/dossier correspondant (fonction "concat_paths") pour obtenir le chemin absolu. Elle vérifie ensuite si le chemin absolu pointe vers un fichier ou un répertoire à l'aide de la fonction "isDirectory".
Si le chemin absolu pointe vers un fichier("isDirectory" retourne 0), la fonction fait appel à "blobFile" pour créer une sauvegarde du fichier, puis met à jour la structure avec les permissions ("getChmod") et le hachage ("sha256file") actuels du fichier.
En revanche, si le chemin absolu pointe vers un répertoire, la fonction initialise un nouveau WorkTree, et parcourt à l'aide de "listDir" les fichiers/dossiers contenus dans ce répertoire en ignorant les fichiers cachés (dont le premier caractère du nom est "."). Elle ajoute ensuite chaque fichier/dossier visible dans le WorkTree à l'aide de la fonction "appendWorkTree" en passant en paramètre le WorkTree ainsi que le nom, le hachage et le mode de l'élément à ajouter. (Nous avons une autre implémentation de la fonction apppendWorkTree ("appendWorkTreeWf") qui prend uniquement le WorkTree et un WorkFile et appelle "appendWorkTree" avec les informations nécessaires, mais elle n'est pas utilisée dans cette fonction précise). Une fois que tous les fichiers/dossiers ont été ajoutés, la fonction sauvegarde le WorkTree grâce à un appel récursif, puis récupère le hachage du WorkTree sauvegardé pour le mettre à jour dans le WorkFile associé au répertoire.
Enfin, la fonction retourne le hachage du WorkTree initial en utilisant la fonction "blobWorkTree". Ainsi, elle permet de sauvegarder le WorkTree complet (fichiers et dossiers) et de mettre à jour les hachages et les modes des WorkFile dans le WorkTree.
La fonction "**restoreCommit**" recherche dans le Commit passé en entrée la valeur de la clé "tree" si elle existe et restaure le WorkTree associé en appelant "restoreWorkTree" qui permet de restaurer une copie locale de l'arborescence de fichiers à partir d'un WorkTree.

3. FONCTIONS PRINCIPALES ET CHOIX D'IMPLÉMENTATION

"restoreWorkTree" prend en entrée les mêmes paramètres que saveWorkTree et crée le répertoire associé au chemin path ("mkdir -p") pour s'assurer que le répertoire cible où l'on veut restaurer la sauvegarde existe. La fonction itère sur les éléments du WorkTree et pour chacun, elle récupère le chemin de hachage en appliquant "hashToPath" au champ hash de la structure, elle crée le chemin complet où il doit être restauré ("concat_paths" entre path et name) et vérifie si cet élément correspond à un fichier ou un répertoire. Dans le premier cas, la fonction "cp" est utilisée pour copier le contenu du fichier du chemin de hachage vers le chemin de restauration. Dans le second cas, on ajoute l'extension ".t" au chemin de hachage pour retrouver le fichier de sauvegarde, et on récupère le WorkTree associé, pour ensuite effectuer un appel récursif avec ce dernier et le chemin de restauration.

3.3. GESTION DE L'ARBORESCENCE

Dans ce projet, la gestion de l'arborescence est basée sur l'utilisation de Commit, qui sont des snapshots du code à un moment donné. Pour effectuer un Commit, on utilise les commandes ./myGit add (fonction "myGitAdd") et ./myGit commit (fonction "myGitCommit").
"myGitAdd" permet d'ajouter un fichier/dossier au répertoire de travail enregistré dans le fichier .add. Elle crée ce dernier s'il n'existe pas, puis récupère le workTree correspondant. Si le fichier/dossier à ajouter n'existe pas, la fonction affiche une erreur. Sinon, elle l'ajoute au WorkTree et le sauvegarde dans .add.
"myGitCommit" crée un nouveau commit pour une branche donnée en utilisant les fichiers ajoutés dans .add. Elle vérifie l'initialisation correcte des références et l'existence de la branche spécifiée, ainsi que la correspondance entre HEAD et la branche. Elle récupère le WorkTree depuis le fichier .add et le sauvegarde ("saveWorkTree"). Elle crée un Commit avec 3 paires principales si les informations correspondantes existent: la clé "tree" de valeur le hachage de la sauvegarde effectuée, la clé "predecessor" de valeur l'ancienne référence de la branche, et la clé "message" de valeur le message passé en paramètre. Finalement, elle efface le fichier .add et sauvegarde le Commit ("blobCommit") puis met à jour les références de la branche et de HEAD pour pointer sur celui-ci.

3. FONCTIONS PRINCIPALES ET CHOIX D'IMPLÉMENTATION

Le déplacement entre les différentes branches ou les diverses versions du travail dans notre dépôt Git se fait grâce aux deux commandes ./myGit checkout-branch (qui fait appel à la fonction "myGitCheckoutBranch" et ./myGit checkout-commit (fonction "myGitCheckoutCommit").
La fonction "**myGitCheckoutBranch**" permet de changer de branche en mettant à jour la référence HEAD pour pointer sur le commit le plus récent de la branche spécifiée. Elle écrit également le nom de la branche dans le fichier .current_branch qui sert à référencer la branche actuelle.
La fonction "**myGitCheckoutCommit**" permet de changer de commit en filtrant tous les Commit à l'aide d'un motif spécifié par l'utilisateur. Ce motif représente généralement les premiers caractères du hachage du commit recherché. Si aucun Commit ne correspond au motif, la fonction affiche un message d'erreur. Si un seul Commit correspond à ce motif, elle met à jour la référence HEAD pour pointer sur ce commit et restaure l'état du WorkTree associé à ce Commit. Si plusieurs Commit correspondent, elle affiche une liste de tous les Commit correspondants et invite l'utilisateur à sélectionner le Commit souhaité.

La commande ./myGit merge (qui exécute le code de la fonction "merge") permet de fusionner deux branches de l'arborescence.
Si les modifications apportées aux deux branches ne se chevauchent pas, la fonction "**merge**" fusionne la branche locale avec la branche distante spécifiée automatiquement et sans obstacles. Elle récupère les WorkTree des deux branches et les fusionne en utilisant la fonction "mergeWorkTrees". Elle crée ensuite un nouveau commit à partir du résultat de la fusion et y ajoute les informations nécessaires.
Cependant, si les modifications apportées aux deux branches se chevauchent, on signale une fusion en conflit. En cas de conflit, la fonction renvoie la liste des conflits à résoudre. Pour ce faire, nous avons choisi pour chaque conflit, d'éliminer l'un des deux fichiers en conflit. Le choix des fichiers à supprimer est fait par l'utilisateur, et la suppression est possible grâce à la fonction "createDeletionCommit".

4. CONCLUSION

La réalisation d'un système de gestion de version a été une expérience très enrichissante qui nous a permis d'acquérir des compétences diverses dans le cadre de cette UE.
Parmi les défis rencontrés, nous pouvons citer les confusions que nous avons eues par rapport à certaines fonctions fournies qui ont freiné notre progression comme par exemple setMode et getChmod, la difficulté et le temps consacré à l'élimination de toutes les fuites mémoire, ainsi que la compréhension approfondie, après de longues recherches, de certains concepts fondamentaux qui semblaient vagues dans l'énoncé, notamment le rôle de certaines références comme HEAD ou master, comment et à quel moment les modifier, la différence entre une référence et une branche, etc.
Malgré les contraintes, nous sommes satisfaits du résultat final. Nous avons réussi à surmonter ces obstacles grâce à notre persévérance et à notre collaboration et à concevoir un système de gestion de version qui permet de gérer les différentes versions d'un projet de manière efficace, avec des fonctionnalités telles que l'enregistrement des modifications apportées, la gestion des conflits et la fusion de branches.
En fin de compte, ce projet nous a appris l'importance de la planification et de la collaboration dans le développement de logiciels, ainsi que des concepts liés à la gestion de versions que nous pourrons appliquer dans d'autres projets à l'avenir.