

LU2IN006 : Rendu de projet - Rapport (2022-2023)

Réalisation d'un logiciel de gestion de versions

Binôme :

- PINHO FERNANDES Enzo - 21107465 - L2-Mono-Info S4 GR6.
- DURBIN Deniz Ali - 21111116 - L2-Mono-Info S4 GR6.

Ce projet consiste à réaliser un logiciel de gestion de versions, c'est-à-dire un outil permettant de créer et stocker différentes sauvegardes/versions d'un projet ou d'un ensemble de fichiers et répertoires.

Nous pourrions donc les visionner et effectuer des opérations dessus, comme accéder à l'historique des changements effectués, revenir à une version antérieure s'il y a un problème, et même faciliter un travail collaboratif où nous pourrions fusionner différentes versions du même projet tout en gérant les conflits potentiels.

Il y a donc une utilité non négligeable pour les grands projets qui, à cause de nombreuses modifications apportées, demandent une trace de chaque version. D'autant plus pour les projets collaboratifs où chaque personne aura ses propres versions qu'il faudra réunir en une seule.

Nous pouvons prendre comme exemple de comparaison Git, qui est lui aussi un logiciel de gestion de versions, gratuit, et créé en 2005 par Linus Torvalds, alias le créateur du noyau Linux.

Arborescence du projet

Voici l'arborescence de notre projet.

Nous avons compris que chaque semaine était consacrée à une notion, une structure, un concept du logiciel.

Par conséquent, nous avons décidé de séparer chacune d'entre elles en répertoire. Vous pouvez voir le répertoire "**sha256**" comportant toutes les fonctions concernant le hachage, ou bien "**work_FileTree**" comportant toutes les fonctions et structures concernant les WorkFiles et les WorkTrees.

Chaque répertoire est composé du code en .c, accompagné de son .h et d'un MAIN que nous utilisons pour tester notre avancée. Et bien évidemment, un Makefile pour compiler le main.

Dans la racine du projet se trouve le code du logiciel de gestion de version "**myGit.c**", du Makefile nécessaire pour compiler, du README, du pdf du projet avec les demandes/exercices et de ce rapport.

Ignorez les fichiers/répertoires : .vscode, LICENSE, .gitignore

▼ LU2IN006-PROJET

> .vscode

> branch

> cellList

▼ commit

 C commit.c

 C commit.h

 C commitTest.c

 M Makefile

 ≡ test_ftc.txt

> fusion

> sha256

> work_FileTree

◆ .gitignore

🔑 LICENSE

M Makefile

C myGit.c

📄 projet-version-3avril.pdf

📄 README.md

Utilisation du logiciel

Pour commencer, il faut bien évidemment utiliser la commande make pour compiler **“myGit.c”**. C’est optionnel mais vous pouvez rajouter le programme dans votre PATH.

Placez vous de préférence dans la racine de votre projet, et voici les différentes commandes que vous pouvez utiliser grâce à **“myGit”** :

- **myGit init**
 - Initialise le répertoire de références et la branche courante.
 - Ce sera la première commande à lancer pour la première fois. Elle crée le dossier **“.refs”**, ainsi que ses fichiers **“HEAD”** et **“master”**. Elle crée également le fichier **“.current_branch”** stockant la branche courante.
- **myGit list-refs**
 - Affiche toutes les références existantes.
- **myGit create-ref <name> <hash>**
 - Crée la référence nommée **<name>** qui pointe vers le commit correspondant à **<hash>**.
- **myGit delete-ref <name>**
 - Supprime la référence **<name>**.
 - Dans ce cas, il faut faire attention. L’utilisateur, s’il supprime sa branche courante, devrait penser à faire un **“checkout-branch”** vers la branche qu’il souhaite afin de changer de branche courante.
- **myGit add <elem> [/<elem2> <elem3>...]**
 - Ajoute AU MOINS un fichier (**elem** obligatoirement, **elem2**, **elem3**...) à la zone de préparation **“.add”** pour faire partie du prochain commit.
 - Ici, la zone de préparation est représentée par un fichier caché **“.add”**. Il est composé des noms de fichiers et/ou de répertoires que l’on souhaite sauvegarder l’état.
- **myGit list-add**
 - Affiche le contenu de la zone de préparation. Donc affiche les noms des fichiers et/ou de répertoires présents dans le fichier **“.add”**.
- **myGit clear-add**
 - Vide la zone de préparation.
- **myGit commit <branch_name> [-m <message>]**
 - Effectue un commit sur une branche nommée **<branch_name>**, avec ou sans message descriptif.
 - Ici, nous créons un point de sauvegarde des fichiers et répertoires présents dans la zone de préparation, sur la branche voulue. Nous ajoutons éventuellement un message pour décrire le commit : nouvelle version, modification ou autre...

- **myGit get-current-branch**
 - Affiche le nom de la branche courante, celle présente dans le fichier caché “.current_branch”
 - La branche courante représente la version du code que vous avez actuellement parmi, éventuellement, d’autres versions alternatives du même projet.
- **myGit branch <branch-name>**
 - Crée une branche qui se nomme <branch-name> si elle n’existe pas déjà.
- **myGit branch-print <branch-name>**
 - Affiche le hash de tous les commits de la branche <branch-name>, accompagné de leur message descriptif éventuel.
- **myGit checkout-branch <branch-name>**
 - Réalise un déplacement sur la branche <branch-name> si elle existe.
 - Ici, la commande modifie notre branche courante, elle devient <branch-name>. Mais de plus, on récupère les versions des fichiers et des répertoires de cette dernière ! Peut-être des versions antérieures du projet, ou une version alternative ou autre, suivant ce qu’a fait l’utilisateur.
- **myGit checkout-commit <pattern>**
 - Réalise un déplacement sur le commit qui commence par <pattern> s’il existe et est le seul. S’il y en a plusieurs correspondant au pattern, on les affiche. S’il n’y en a aucun, il y aura un message d’erreur.
 - Au lieu de changer de branche, nous appelons directement un commit en particulier grâce à son hash afin de récupérer une ancienne version de notre projet. Attention, la commande fonctionne uniquement si le pattern du hash est assez détaillé, et donc qu’il correspond à un seul commit.
- **myGit merge <branch> <message>**
 - S’il y a pas de collision entre la branche courante et <branch>, on réalise la fusion entre les deux branches. Sinon, l’utilisateur choisira entre : garder les fichiers/répertoires de la branche courante, ceux de la branche <branch>, ou choisir manuellement conflit par conflit les fichiers/répertoires concernés.
 - Ici, une collision arrive quand on souhaite fusionner deux branches ayant le même fichier/répertoire sauvegardé en apparence mais avec un contenu différent. Par conséquent, il faut prendre ses précautions et choisir nous-même quelle version du fichier/répertoire nous allons garder. Cette commande demande plus d’attention, on évite les risques de perte.

Présentation des différentes notions et structures

Le hachage :

- Nous avons travaillé durant la première semaine sur le “**SHA256**” signifiant “**Secure hash algorithm 256 bits**”. En appliquant la fonction de hachage “**SHA256**” sur un fichier, nous pouvons récupérer le hash de ce dernier et le transformer en un chemin où se trouvera l’enregistrement de l’instantané (sauvegarde de l’état actuel) du fichier. Il suffit de rajouter un “/” entre le deuxième et le troisième caractère du hash.
 - Nous avons également dû choisir une fonction de hachage pour les tables de hachage (Commit). Nous avons choisi le “**sdbm**” parce qu’elle permet une bonne distribution des clés et moins de fractionnement.
-
-

La cellule et la liste :


- C’est une simple structure, une *liste chaînée* comportant des *cellules*. Chaque cellule contient une chaîne de caractères.
- On l’a beaucoup utilisé quand il fallait, par exemple, stocker tous les noms de fichiers et répertoires d’un chemin. Cela nous a permis de faire une fonction vérifiant l’existence d’un fichier ou répertoire. Ou bien, ça nous a permis de créer une liste de conflits pour la fusion des branches.



```
1  typedef struct cell {
2      char *data;
3      struct cell *next;
4  } Cell;
5
6  typedef Cell* List;
```

Le WorkFile et le WorkTree :


- Un “**WorkFile**” est la représentation d’un fichier/répertoire. Il est donc naturellement composé du nom de ce dernier, mais aussi du hash associé à son contenu et du mode correspondant à ses permissions.
- Grâce à cette structure, nous pouvions manipuler un fichier/répertoire, et créer son instantané. Mais aussi et surtout, de pouvoir créer l’instantané de PLUSIEURS fichiers/répertoires. Là est le rôle du WorkTree.
- Un “**WorkTree**” est une structure représentant un tableau de pointeurs de WorkFiles. Nous avons décidé de faire cela plutôt qu’un tableau vers des WorkFiles, nous trouvons cela plus adapté au reste du projet. Il y a également différentes informations comme la taille maximale <size> étant définie par **MAX_TAB_WF**, ou le nombre d’éléments présents <n>.



```
1  #define MAX_TAB_WF 100
2
3  typedef struct{
4      char *name;
5      char *hash;
6      int mode;
7  } WorkFile;
8
9  typedef struct{
10     WorkFile **tab;
11     int size;
12     int n;
13 } WorkTree;
```

Le Commit :

- Déjà, une “**key_value_pair**” (kvp) est une structure permettant d’associer une clé et une valeur, un genre de dictionnaire.
- Il nous sera utile afin de stocker différentes informations d’un Commit.
- Un “**Commit**” est une structure représentant une table de hachage de kvp, on retrouve également encore sa taille maximale définie par **MAX_SIZE_COMMIT**, et le nombre d’éléments présents <n>.
- Il représente l’enregistrement instantané d’un WorkTree ainsi que de diverses informations utiles, représentées par les kvp. Par exemple, avec la clé “tree”, nous pouvons retrouver le hash de l’instantané du WorkTree. On peut stocker aussi un message, ou bien avec la clé “predecessor” si elle existe, retrouver le commit précédent !



```
1  #define MAX_SIZE_COMMIT 100
2
3  typedef struct key_value_pair {
4      char *key;
5      char *value;
6  } kvp;
7
8  typedef struct hash_table {
9      kvp** T;
10     int n;
11     int size;
12 } HashTable;
13
14 typedef HashTable Commit;
```

Les branches :

- Avant tout ça, nous travaillons uniquement dans le but de gérer une seule **branche**, donc de suivre un historique du projet linéaire. Mais en utilisant plusieurs branches à la fois, nous pouvons avoir plusieurs versions alternatives d’un même projet !
- Cela est très utile pour, par exemple, travailler séparément sur différentes fonctionnalités du projet tout en gardant en sécurité la branche principale de tout malheur, avoir plusieurs versions du projet, créer une version brouillon, ou même permettre un travail collaboratif où chaque personne aura sa propre version avant la fusion !
- Dans notre code, on a donc des fonctions pour gérer ces différentes branches, savoir quelle est la branche courante grâce au fichier caché “**.current_branch**” ou de naviguer entre elles !

La fusion de branches :

- Il existe certains cas où l’on souhaiterait fusionner plusieurs branches. Par exemple, si moi et mon binôme travaillons sur ce projet en simultané, nous voudrions forcément en créer une version finale en réunissant nos deux versions. Là est l’utilité de cette notion !
- Une fusion est donc un simple commit des WorkTrees des deux branches fusionnées. Mais imaginons que l’on travaille nous deux sur le même fichier. Nous avons donc le même fichier, mais avec un contenu différent, comment choisir ? La fusion doit également gérer les conflits de ce genre, et ce logiciel propose également ça. Nous avons le choix entre garder uniquement les fichiers en conflits de la branche courante, de l’autre branche, ou bien même de choisir au cas par cas ! Tout cela permet une utilisation pertinente des branches, pour une collaboration optimale.

Présentation des fonctions principales

Les fonctions blob

- Toutes les fonctions “**blob**” permettent l’enregistrement instantané d’un ou plusieurs fichiers/répertoires dans le répertoire caché “.autosave”. Nous prenons le hash de ce que l’on souhaite sauvegarder. Les deux premiers caractères seront le nom du répertoire, dans “.autosave” où l’on stockera l’instantané. Le reste du hash sera son nom, avec éventuellement une extension.
- “**void blobFile(char *file)**” crée l’enregistrement de l’instantané d’un fichier donné en paramètre. Il suit la même logique expliquée plus haut, sans extension. On aura donc un nouvel instantané avec un contenu identique à celui du fichier <file>.
- “**char* blobWorkTree(WorkTree *wt)**” crée l’enregistrement du WorkTree <wt> cette fois mais avec l’extension “**.t**” afin de le différencier des autres. Nous aurons donc un nouvel instantané avec plusieurs noms de fichiers/répertoires ainsi que leur hash et leur permission. La fonction renvoie le hash de l’instantané.
- “**char* blobCommit(Commit *c)**” crée l’enregistrement du Commit <c> avec l’extension “**.c**”. Nous aurons donc un nouvel instantané avec plusieurs informations comme le hash du WorkTree correspondant au point de sauvegarde (Clé : “tree”), éventuellement un message description (Clé : “message”) et le hash du Commit correspondant au point de sauvegarde précédent (Clé : “predecessor”). La fonction renvoie le hash de l’instantané.

char* saveWorkTree(WorkTree *wt, char *path)

- Cette fonction permet de créer les enregistrements instantanés de tous les WorkFiles dans le WorkTree <wt>, puis de lui-même.
- Si le WorkFile représente un fichier, alors on fait appelle à “**blobFile**”, puis on récupère son hash et son mode pour le sauvegarder dans ce même WorkFile.
- Si le WorkFile représente un répertoire, alors on recrée un deuxième WorkTree pour y stocker tout le contenu de ce répertoire et faire un appel récursif. On récupère le hash et le mode de ce répertoire ensuite et on le stocke dans ce même WorkFile.
- Après tout ça, il manque plus que sauvegarder le WorkTree <wt> en appelant “**blobWorkTree**” et pour retourner son hash.

void restoreWorkTree(WorkTree *wt, char *path)

- Cette fonction permet de récupérer une version antérieure de certains fichiers et répertoires, grâce à un WorkTree.
- On parcourt le tableau de WorkFiles. S’il ne contient pas l’extension “**.t**”, alors c’est un fichier. On crée une copie de l’instantané à l’endroit indiqué par <path>, et on lui donne le bon nom et la bonne permission indiqué dans le WorkFile.

- S'il possède l'extension **“.t”**, c'est un répertoire. On crée le WorkTree qui le représente, on ajoute à la variable <path> ce répertoire et on fait un appel récursif, permettant de restaurer ces fichiers dedans.

=====

void myGitAdd(char *file_or_folder)

- Cette fonction ajoute un fichier ou un répertoire dans le WorkTree correspondant à la zone de préparation. Elle permettra de savoir quels fichiers et répertoires commit.

=====

void myGitCommit(char *branch_name, char *message)

- Cette fonction permet de créer un point de sauvegarde des fichiers et répertoires présents dans la zone de préparation, dans la branche <branch_name> avec éventuellement un message descriptif.
- Nous avons quelques contraintes avec cette fonction. Par exemple, la branche doit au préalable exister et en plus, **“HEAD”** doit pointer sur le dernier commit de cette branche.
- Cette fonction se déroule en plusieurs étapes.
 - Création du WorkTree représentant la zone de préparation, et suppression de cette dernière.
 - On enregistre le WorkTree avec **“saveWorkTree”** qui renvoie son hash.
 - On crée le commit correspondant à ce WorkTree en précisant le hash de ce dernier avec la clé **“tree”**. De plus, on rajoute le hash du **“predecessor”** s'il existe. On peut le trouver car au fichier **“.refs/<branch_name>”**. On a éventuellement le message avec.
 - On crée l'instantané du commit grâce à la fonction **“blobCommit”** qui renvoie son hash.
 - On modifie <branch_name> et **“HEAD”** pour qu'ils pointent sur le hash du commit.
- Nous voyons donc que cette fonction sera une des plus importantes. Elle utilise toutes les autres fonctions vu précédemment, pour faire l'instantané le plus important et le plus rempli en information.
- Nous aurons également la fonction **“void restoreCommit(char *hash_commit)”** qui permet de restaurer ce commit, comme pour le WorkTree.

=====

void myGitCheckoutBranch(char *branch)

- Réalise un déplacement sur la branche <branch-name> si elle existe.
- Ici, la commande modifie notre branche courante, elle devient <branch-name>. Mais de plus, on récupère les versions des fichiers et des répertoires de cette dernière ! Peut-être des versions antérieures du projet, ou une version alternative ou autre, suivant ce qu'a fait l'utilisateur.

=====

void myGitCheckoutCommit(char *pattern)

- La fonction réalise un déplacement sur le commit qui commence par <pattern> s'il existe et est le seul. S'il y en a plusieurs correspondant au pattern, on les affiche. S'il n'y en a aucun, il y aura un message d'erreur.
- Au lieu de changer de branche, nous appelons directement un commit en particulier grâce à son hash afin de récupérer une ancienne version de notre projet. Attention, la commande fonctionne uniquement si le pattern du hash est assez détaillé, et donc qu'il correspond à un seul commit.

void createDeletionCommit(char *branch, List *conflicts, char *message)

- La fonction permet de créer un commit de suppression sur la branche **<branch>**. Cela est très utile dans le cas où nous avons des conflits lors d'une fusion. Ce commit contiendra que les fichiers et répertoires qui ne sont pas en conflits. La fonction suit ces étapes :
 - On se déplace sur **<branch>**.
 - On récupère le commit pointé par cette branche ainsi que son WorkTree.
 - Puis, avec une zone de préparation vide, nous y ajoutons tous les noms de fichiers et répertoires du WorkTree qui ne sont pas présent dans la liste **<conflicts>**
 - On fait un enregistrement instantané de ce nouveau commit avec éventuellement un message descriptif, et on revient sur la branche précédemment courante.

List *merge(char *remote_branch, char *message)

- Cette fonction permet la fusion entre la branche courante et **<remote_branch>** s'il y a aucun conflit. La fonction suit ces étapes :
 - On crée le WorkTree de fusion contenant les noms de fichiers et répertoires des deux autres WorkTrees, représentés par les branches. Puis avec lui, on crée son commit, en pensant à rajouter le **"predecessor"** mais également le **"merged-predecessor"** ! On garde donc en mémoire les deux différentes versions. On ajoute éventuellement un message descriptif.
 - Puis encore une fois, on réalise un enregistrement instantané du WorkTree et du commit, et on modifie la branche courante et **"HEAD"** pour qu'ils pointent sur l'instantané du commit.
 - On supprime **<remote_branch>** et on restaure le WorkTree de la fusion, pour avoir la nouvelle version sur votre ordinateur.
 - La fonction renvoie à NULL.
- Tout de fois, s'il y a des conflits, la fonction se contente de renvoyer la liste des conflits.

Conclusion du projet

Ce projet est très intéressant. En effet, nous avons pu apprendre plus en détail le fonctionnement d'un logiciel de gestion de version, ce qui nous sera forcément utile dans le futur. Que ce soit pour nos projets personnels, pour travailler en équipe avec une petite ou une grande équipe, ou juste garder une trace de plusieurs versions de nos documents.