

Rapport Projet Gitar

Sven - Abel

28 mai 2023

1 Introduction

Le projet gitar consiste en la reproduction des commandes du système *git* les plus courantes. Nous avons commencé par suivre le tutoriel « Write yourself a Git! », qui nous a permis de coder les structures des différents objets puis avons ajouté les parties pour faire fonctionner git : la zone tampon, la gestion des branches, ainsi que la gestion d'un dossier à distance.

L'ensemble du projet est segmenté en différents dossiers représentant chacun la partie du code qui y correspond. On trouve comme dossiers :

1. **analyseur** : Un parseur et lexer Menhir et Ocamllex afin de lire la ligne de commande
2. **branches** : Les fonctions pour gérer les différentes branches
3. **diff** : La gestion de la différence entre des fichiers
4. **distance** : Les fonctions pour gérer les dossiers distants
5. **init** : Pour l'initialisation d'un dossier avec gitar
6. **objets** : Les structures d'objets
7. **utilitaire** : Des fonctions utiles, ainsi que l'affichage
8. **zone-tampon** : La zone-tampon et ses commandes

Le fichier `gitar.ml` pilote les différentes commandes lancées.

La liste des commandes ainsi que leur documentation et leur utilisation est disponible en lançant le projet avec la commande `gitar --aide`.

En général, le code source de chaque commande est regroupé dans un fichier du même nom.

2 Objets

Les fonctions déclarant les objets, les lisant et les écrivant est une réimplémentation en OCaml du tutoriel « Write yourself a Git! ».

Il existe des objets de différents types :

1. **blob** : La représentation utilisée est la connaissance du contenu et du sha
2. **tree** : On conserve le sha de l'arbre, ainsi qu'une liste d'objets qui sont les fichiers ou les autres dossiers imbriqués dans l'arbre
3. **commit** : On conserve une map des différents champs du commit : Les parents, l'arbre qui correspond au commit , le message ...
4. **tag** : Peu utilisé dans le projet, est censé représenter une référence.

On a cependant regroupé tous les types en un seul type **objetgit** (dans **objets/struct.ml**), qui est une classe avec tous les attributs dont peuvent avoir besoin les différents objets. Ainsi, les objets auront certains attributs non initialisés, mais c'était plus facile à manipuler que de créer des types spécifiques et différents, sachant qu'il y avait beaucoup de redondance (le chemin, le **sha**...)

Pour stocker un objet, son contenu avec un en-tête pour connaître de quel type il s'agit, est alors haché avec **sha1** puis conservé dans une base de donnée dans **.gitar/objects** où le nom de l'objet est repéré par son sha.

La fonction **object_find** permet de retrouver le sha d'un objet, à partir du nom d'une branche ou d'un commit (ou d'une référence).

On peut ainsi, avec les commandes **affiche-ref** ou **etiquette** afficher les références vers des objets qui existent, ainsi que créer une étiquette vers un objet, et **hache-objet** et **affiche-fichier** permettent respectivement de hacher un objet d'afficher un objet git.

3 Zone tampon

La zone tampon est composée d'une liste des fichiers qui sont indexés par **gitar**, en conservant toutes sortes de méta-données ainsi que le haché de la version de chaque fichier conservée. Dans le code, la zone tampon est initialisée dans **zone_tampon/index.ml** et est le type **GitIndex**. Puis chaque fichier qui est dans l'index est du type **GitIndexEntry**

Ainsi, pour ajouter un fichier à l'index il suffit de le hacher et de l'ajouter à la liste, on peut alors créer un engagement en capturant l'état actuel de l'index.

L'écriture de la commande statut a été la plus compliquée de cette section : en effet, il s'agit de vérifier les différences entre l'index, le dernier engagement et l'état actuel du dossier. Cependant, un objet de type arbre ne contient que les fichiers et dossiers qui sont directement compris dans le dossier, mais

l'index contient l'entièreté des fichiers avec des chemins par rapport à la racine de **gitar**. Il a donc fallu itérer sur ces trois structures.

3.1 Ignore

Gitar possède aussi la possibilité d'ajouter un fichier **.gitarignore** dans la racine du projet uniquement.

Quand on lance les commandes **ajout** et **statut**, on vérifie d'abord qu'il n'existe pas de nom de fichier dont le nom pourrait correspondre au début d'une des lignes du **.gitarignore**, et si c'est le cas on ne fait rien sur ce fichier.

3.2 Reinitialise

Il y a également la possibilité de réinitialiser l'index avec la commande **reinitialise**. Pour toutes ces fonctions, il a fallu créer des fonctions de traductions, d'objets **git** vers des objets **index** vers des dossiers physiques et inversement.

4 Branches

4.1 Branches

La gestion des branches locales se fait via une liste de branches existantes dans le répertoire **refs/heads** du répertoire **.gitar**. Il contient un fichier par branche et chaque fichier contient le sha de son commit de tête. La branche courante est conservée comme chemin relatif du fichier de la branche dans le fichier **HEAD** du répertoire **.gitar**. La fonction **branch** passée sans argument donne la liste des branches actives locales et met en avant la branche actuelle, deux paramètres **-r** et **-a** permettent d'afficher seulement les branches distantes et toutes les branches respectivement. Lorsque **branch** est passée avec un nom nouveau, la branche du nom donné est créée, et devient la branche actuelle. Le paramètre **-d** permet la suppression de la branche spécifiée. L'implémentation est plutôt simple avec essentiellement une manipulation des shas et des noms au sein des différents fichiers du répertoire **.gitar**.

4.2 Checkout

checkout est similaire à **branch nom_branche**, lorsque l'option **-b** est passée. Cette fonction a pour utilité de changer de branche en indiquant la branche destination. Lorsqu'on applique cette fonction, on vide le projet de

tous ses fichiers, puis on met l'ensemble des fichiers avec leur contenu qui correspondant au commit vers lequel on bouge.

4.3 Fusion : Merge

La fonction **merge** est la fonction clef de Git. Elle se décompose en deux parties réalisées successivement. Dans ce projet, on a implémenté seulement une stratégie de fusion. À partir de deux branches (les commits situés aux extrémités de ces branches), on détermine le plus proche commit ancêtre commun via les fonctions **ancetre_commun** et **fuseau**. On réalise ensuite une comparaison entre les différents objets présents dans chaque commit : avec deux éléments de comparaison : le nom et le sha des trois commits : les deux à fusionner et l'ancêtre. Il en découle un ensemble de correspondances entre les fichiers de chaque branche, d'où l'on obtient le "type" d'opération à réaliser lors de la fusion des branches. ces opérations sont des catégories dans lesquelles se retrouvent les fichiers :

- **crees** fichiers créés depuis l'ancêtre
- **conserves** les fichiers non modifiés et non supprimés dans les branches filles
- **modifies** les fichiers modifiés dans seulement une branche fille
- **supprimes** les fichiers supprimés dans une branche fille (ils sont conservés en sortie toutefois)
- **renommes** les fichiers renommés dans au moins une branche fille
- **conflits_nom** les susdits fichiers renommés dont les noms donnés dans les branches filles sont différents et différents du nom du fichier correspondant dans l'ancêtre
- **fusionnes** les fichiers à fusionner
- **conflits_fusion** les fichiers ayant posé des problèmes lors de la fusion

Une fois les fichiers catalogués dans ces catégories, on réalise les fusions nécessaires grâce à **Diff3**.

Comme dans notre implémentation, la branche fusionnée est supprimée, on fait attention à ne changer de branche qu'au moment opportun, pour éviter de se retrouver dans une branche "morte" qui à été supprimée suite à une fusion, en se déplaçant vers la branche d'arrivée de fusion si on est situé au préalable dans la branche vouée à mourir.

Dans le cas particulier où l'une des deux branches fusionnées serait l'ancêtre de l'autre, on se replace facilement sur la version la plus avancée, sans avoir à gérer les cas précédemment cités.

5 Diff

5.1 Diff

Nous avons réimplémenté la différence entre différents fichiers, en s'inspirant de l'algorithme de Myers.

Plus précisément : Pour calculer la différence entre deux fichiers, on calcule le nombre minimum de différence entre les lignes.

Avec $LSP(i, j)$ = nombre minimum de différences sur les i premières lignes du fichier A , j premières du fichier B.

On a alors :

$$LSP(i, j) = \min(LSP(i-1, j) + 1, LSP(i, j-1) + 1, LSP(i-1, j-1) \text{ si } A[i] = B[j])$$

En conservant d'où vient l'antécédent, on va reconstituer les deux fichiers.

Si on est dans le cas 1, on considère que c'est une suppression, si c'est le cas 2 un ajout, et dans le dernier cas c'est que la ligne existe dans les deux fichiers.

L'implémentation utilisé par gitar (qui est la même que celle de git) est de favoriser des chemins avec des **ajouts** ou des **suppressions** consécutives, afin que la différence ressemble le plus possible à ce qui a pu vraiment se passer.

A partir de cette implémentation, on peut regarder la différence entre des fichiers, des dossiers, des commits, par rapport à la zone tampon, ce qui a demandé de gérer beaucoup de cas particuliers.

5.2 Diff3

Pour l'implémentation de **diff3**, on a besoin d'un fichier ancien **origine** , et de deux fichiers A et B.

On réalise d'abord **diff** entre **origine** et A, et entre **origine** et B.

Le fichier global est ensuite décomposé en blocs, où la délimitation d'un bloc est une ligne conservée commune par les trois documents. Puis on gère chaque bloc, en créant un conflit si A et B ont été modifiés, ou en prenant celui qui a été modifié par rapport à **origine**.

Il n'y a cependant pas de gestion des conflits outre le fait de mettre les deux blocs dans le fichier séparé de chevrons, mais les conflits sont annoncés. Il faut donc ensuite aller dans le fichier pour régler les erreurs soit-même.

6 Distance

Nous avons décidé de créer un protocole de toutes pièces afin que ce soit mis en application de manière assez simple.

En plus des protocoles, chaque commande fait des vérifications de base pour s'assurer que tout va bien se passer ou gagner du temps. Par exemple, quand on pousse, s'il n'existe pas de branche correspondante distante, il faut d'abord la créer grâce à l'option **--set-upstream**, et si la branche distante est déjà à jour on ne fait rien.

Alors que ces commandes marchent parfaitement bien sur l'ordinateur de Sven (sur Linux) comme montré par la démonstration lors de la présentation, les commandes de base d'Unix par rapport aux sockets ne fonctionnent pas sur l'ordinateur d'Abel (Mac), et nous n'avons pas trouvé comment régler le problème. En particulier, les commandes `Unix.establish_server` ou `Unix.open_connection` sont la cause du problème.

Ces commandes ne fonctionnent également que avec l'utilisation d'une adresse IP internet, pas avec l'adresse du localhost.

6.1 branches à distance

Afin de pouvoir faire **tire**, **pousse**... il faut que savoir vers où.

C'est pour cela que de manière similaire à la vraie version de `git`, dans un fichier `.gitar/config`, on met les informations suivantes :

- Nous avons décidé de nommer par "dossier distant" le nom communément **origin**. Ainsi quand on veut ajouter un dossier distant on met l'url d'une racine ainsi que le nom du dossier, et on va créer à cette racine un projet **gitar** qui va avoir pour nom ce dossier.
- Pour chaque dossier distant, on conserve l'url de la racine qui correspond.
- Pour chaque branche qui possède une version distante, il existe une entrée dans `.gitar/config` qui indique quel est le nom du dossier distant. Ainsi, on va toujours par défaut faire **pousse** ou **tire** sur le dossier distant vers lequel pointe la branche même si c'est indiqué autrement dans la ligne de commande.

Les branches distantes sont stockées dans

`.gitar/refs/remotes/<nom_distant>/<nom_branche>`, et chacune contient le sha du commit de tête présumé dans le dossier distant. Ainsi, quand on **tire** ou **pousse**, on va mettre à jour le sha vers lequel pointe la branche distante, et si besoin créer de nouvelles branches distantes.

De plus, par soucis d'optimisation et de bonne utilisation, si quand on **tire** la branche sur laquelle on est n'est pas distante, mais qu'un dossier distant a été donné, on essaie de lier la branche à la branche distante correspondante si possible, plutôt que de devoir le faire manuellement avec `gitar branche -t <nom_distant>`

Pour des raisons d'utilisations, ainsi que le module qui ne fonctionnait pas, pour l'écriture dans le fichier `.gitar/config` de type `ini`, nous avons introduit le code source du module `inifiles`, ainsi que ajouté quelques méthodes pour des cas particuliers qui semblaient manquer.

6.2 Etablissement du serveur

De manière générale, quand on lance **cherche** (**fetch**), **tire** ou **pousse**, le côté client va lancer selon le chemin déterminé pour le dossier distant la commande

correspondante sur l'autre machine (par exemple pour `fetch`, le client va lancer la commande `gitar fetch-serveur` sur la machine distante.)

La communication entre le client et le serveur se fait alors par l'écoute du serveur d'un de ses ports donné.

Pour une raison d'implémentation pratique, quand on lance le serveur avec la commande `Unix.establish_server`, comme cette commande ne termine pas, on `fork` sur la commande et le père tue le serveur au bout de 5 secondes, pour que le serveur s'arrête après la liaison et que le port soit libéré.

6.3 Cherche (fetch)

Le protocole de communication pour cherche est le suivant :

1. Le client envoie l'ensemble des têtes de branches qu'il possède.
2. Le serveur calcule les objets qu'il possède mais que le client ne possède peut-être pas à partir de l'information précédent et crée une archive tar de ces objets.
3. Le client récupère l'archive avec `scp`, décompresse l'archive et ajoute les objets à sa base de donnée d'objets, puis met à jour les têtes distantes, dans le dossier `.gitar/refs/remotes/..`.

6.4 Pousse (push)

Le but est d'envoyer les informations d'une unique branche. Le protocole est le suivant :

1. Le client envoie au serveur l'état de la tête de la branche actuelle et son nom
2. Le serveur répond en envoyant l'état de la branche correspondante
3. Le client calcule quels objets manquent au serveur et crée une archive tar qu'il envoie au serveur
4. Le serveur décompresse l'archive et stocke les nouveaux objets dans sa base de donnée puis met à jour sa tête.

6.5 Tire (pull)

Le protocole est le suivant : On applique la fonction `cherche` puis la fonction `fusionne`.

De plus, s'il existe une branche distante qui n'existe pas en local, elle est créée en local et pointe vers la branche distante, afin qu'on puisse bouger facilement dans cette branche avec l'état distant. (Il aurait sinon fallu créer la branche locale, puis `pull`, ce qui aurait sûrement amené à des conflits de fusion, car la branche créée localement aurait eu l'état d'une autre branche locale, peut-être incompatible.)

Dans cette section, des optimisations on également été faites. Par exemple, s'il n'y a rien à envoyer ou à recevoir, le protocole va s'arrêter avant l'échange de l'archive, et le signaler à l'utilisateur.

De plus, quand on fait les commandes **statut**, **bouge**, et les commandes distantes, on affiche si la branche sur laquelle on est est locale ou si est à jour, en retard ou en avance avec ce qu'on connaît de l'état distant de la branche.

7 Utilitaires

Ce répertoire regroupe les fonctions très générales utilisées un peu partout au sein du projet. On y trouve notamment :

- Les fonctions d'affichage **affiche**, **affiche_fond** et **affiche_gras** qui ont la même puissance que la fonction `Printf.printf` (sur laquelle leur fonctionnement repose) avec une possibilité d'ajout de couleur -au texte, ou au fond- grâce à la pose de balises d'échappement du terminal. **affiche_gras** affiche le texte en caractères gras.
- La fonction `--aide` de **gitar** offre une aide relative à chaque fonction gitar passée en paramètre de même que la liste de toutes les fonctions disponibles, couplée avec les descriptions des paramètres éventuels de chaque fonction. La commande **gitar documentation -a** liste toute la documentation de manière détaillée pour toutes les fonctions.
- La gestion des erreurs diverses pouvant être relevées dans les autres fonctions, et leur affichage correspondant.
- La gestion d'avertissements éventuels qui seront alors affichés à la fin de l'exécution de la commande.
- Un dictionnaire de correspondance français-anglais sert pour avoir une traduction dans les deux langues des commandes d'appel des fonctions **gitar**.
- Un ensemble de fonction usuelles utilisées notamment pour convertir les contenus de fichiers et les chaînes de caractères.

8 Makefile et commande générale

On a créé un script bash, et un makefile permettant d'automatiser la compilation et de bénéficier d'une commande générale **gitar** permettant d'appeler facilement le programme quel que soit le répertoire dans lequel on se trouve. On crée donc à cet effet un lien symbolique vers **gitar** dans le dossier `/usr/local/bin`.

Le Makefile contient également quelques démonstrations :

- `demo_distant` pour le dossier distant
- `demo` fait une démonstration des commandes de base ainsi que d'une fusion
- `demo_diff` fait une démonstration des commandes **diff** et **diff3**
- `demo_ignore` fait une démonstration de l'utilisation de `.gitarignore`

9 Conclusion

Nous sommes très satisfait de l'état final du projet qui est quasiment utilisable en l'état. Il manquerait cependant quelques fonctionnalités utiles, comme par exemple **rebase** pour faire des appels distants plus ergonomiques.

Il est également difficile de prévoir tous les cas d'utilisations des commandes, et nous avons fait en sorte de produire des messages d'erreur ou des avertissements dans les différents cas que nous avons découvert, ce qui a demandé beaucoup d'écriture de cas dans toutes les fonctions, et ainsi certaines fonctions simples sont devenues relativement complexes à écrire, comme la manipulation, la création, la suppression, la gestion distantes des branches.

Le projet est également relativement compatible avec un projet créé avec git. On peut par exemple reprendre un projet git existant, renommer le `.git` en `.gitar` et décompresser les **packfiles** que nous ne supportons pas, et la lecture de l'arbre de commits, les branches sont alors lisibles et utilisable.