École normale supérieure — Info-L3-Systèmes
Année 2023-2024

Projet « Mongit »

Le but de ce projet est d'implémenter un petit gestionnaire de versions d'après le modèle du logiciel *git*. En le faisant, vous gagnerez une compréhension profonde du fonctionnement de git, vous maitriserez des parties importantes de l'interface POSIX, et vous aurez la possibilité d'aller plus loin en développant de nouvelles fonctionnalités.

Le point de départ du projet est le tutoriel de Thibault Polge « Write yourself a Git! » (`https://wyag.thb.lt`). Ce tutoriel décrit une implémentation en Python d'une petit interface git. Nous vous suggérons de lire ce tutoriel et de faire des expériences sur la ligne de commande avec un dépôt git que vous n'aurez pas peur de corrompre... Dans un premier temps, il n'est pas nécessaire de débuter votre propre implémentation. Le site `https://git-scm.com` fournit quelques bonnes références sur git si vous en avez besoin. Il y a même une traduction en français du livre « Pro Git » de Scott Chacon et Ben Straub (`https://git-scm.com/book/fr/`).

Notez que vous n'êtes pas obligé, ni de reprendre complètement l'interface de git, ni de rester compatible avec ses structures de données (mais vous pouvez le faire si vous le souhaitez). Après avoir compris comment marche git, n'hésitez pas à faire vos propres choix de conception et d'implémentation.

Certaines parties du tutoriel sont spécifiques à Python. Notamment, l'utilisation de certaines bibliothèques et le traitement d'arguments de la ligne de commande (§2), mais aussi l'utilisation d'objets pour structurer le programme (`GitRepository`, `GitObject`, `GitCommit`, ...). Il y a quelques suggestions ci-dessous pour retrouver certaines fonctionnalités en OCaml. Sinon il vaudrait mieux structurer le programme autrement en OCaml.

Il y a quelques bibliothèques OCaml pour analyser la ligne de commande, comme *Arg* dans la bibliothèque standard et *Cmdliner*, mais vous pouvez aussi interpréter directement le tableau `Sys.argv`. Pour les tableaux associatifs, la bibliothèque standard fournit `Map` (immuable) et `Hashtbl` (impératif), mais il faudra travailler un peu plus pour fixer l'ordre des clés. Il y a une bibliothèque OCaml pour lire les fichiers `.ini` (*ocaml-inifiles*, mais son interface et ses résultats sur les fichiers `.git/config` laissent à désirer (si vous implémentez une meilleure version, n'hésitez pas à la partager avec la communauté OCaml). Les modules `Unix` et `Filename` de la bibliothèque standard seront presque incontournables pour cette tâche.

La bibliothèque *CamlZip* peut décomprimer les fichiers git. Vous pouvez l'installer et tester avec les commandes suivantes.

```
opam install camlzip
wget https://github.com/xavierleroy/camlzip/blob/master/test/testzlib.ml
ocamlfind ocamlopt -o testzlib -package zip -linkpkg testzlib.ml
./testzlib -d .git/objects/78/bc26100f2cd5888cb49f0c8d9f0b3bb6aabfd7 file
```

La bibliothèque *ocaml-sha* peut calculer les clés de hachage SHA1 utilisées par git. Vous pouvez l'installer et tester avec les commandes suivantes.

```
opam install sha
echo "let _ = print_endline Sha1.(to_hex (channel stdin (-1)))" \
    > testsha1.ml
ocamlfind ocamlopt -o testsha1 -package sha -linkpkg testsha1.ml
./testsha1 < file
```

Si vous réussissez à reproduire en OCaml les résultats du tutoriel, voici quelques extensions possibles.

- Implémenter le *protocole stupide* (GitPro, §10.6) pour télécharger des objets git en lecture seule avec le protocole http.
- Concevoir et implémenter votre propre protocole « moins stupide » en utilisant les prises (`Unix.socket`).
- Traiter les *fichiers pack* de git.
- Analyser et implémenter d'autres fonctionnalités (`git merge`, `git stash`, `git rebase`).
- Concevoir et implémenter un système de collaboration en temps-réel basé sur git. Par exemple, imaginez que vous travailliez ensemble avec d'autres sur la même branche d'un même projet. Au lieu d'attendre de faire un `git pull` pour apprendre qu'il y a un conflit à gérer, vous voudriez avoir une information en direct sur les lignes et fichiers qui ont été modifiées par vos collaborateurs, mais pas encore sauvegardés dans le dépôt local (commit). On pourrait imaginer que chacun lance un client pour surveiller les fichiers locaux et communiquer les modifications aux autres clients (avec les prises). Pour une première version, il suffirait d'afficher les modifications sur le stdout du client. Une version plus avancée pour fournir une intégration avec un éditeur de texte pour montrer les parties des fichiers qui sont en train d'être éditées ailleurs.

Modalité de rendu

Le projet est à faire en binôme. Il doit être remis par email à

`Timothy.Bourke@ens.fr`
avant le **dimanche 26 mai 2024**.

Votre projet doit se présenter sous forme d'une archive `tar` compressée (option `z` de `tar`), appelée `vos_noms.tgz` qui doit contenir un répertoire appelé `vos_noms` (par exemple : `dupont-durand.tgz`). Dans ce répertoire doivent se trouver les sources de votre programme (inutile d'inclure les fichiers compilés). Quand on se place dans ce répertoire, la commande `make` doit compiler votre programme. La commande `make clean` doit effacer tous les fichiers que `make` a engendrés et ne laisser dans le répertoire que les fichiers sources. L'archive doit également contenir un court rapport expliquant les différents choix techniques qui ont été faits et, le cas échéant, les difficultés rencontrées ou les éléments non réalisés. Ce rapport pourra être fourni dans un format ASCII, PostScript ou PDF.