rapport stage

Oumaima Hajji

August 2, 2021

Contents

1 Abstract (TODO)

2 Introduction (TODO)

3 Etat de l'art (TOEDIT)

3.1 Historique, gros fichiers

Dans la science moderne, particulièrement la science computationnelle et la data science, on gère un énorme volume des fichiers et de données qui évoluent fréquemment au cours du temps. Il faut donc garder une trace de chaque version de données utilisées et du code exécuté à un instant pour pouvoir garder une empreinte continue du développement de la recherche et donc pour comprendre et suivre tous les pas de l'expérience. C'est là où se manifeste l'importance de 'Data Version Control' dans un domaine de recherche où on manipule un tas de données; on perd la reproductibilité du résultat quand on ne manage pas du contrôle de données. C'est avec cette philosophie en tête que l'on cherche à utiliser un outil pour s'occuper de cette tâche. Le logiciel que l'on a décidé d'utiliser est Git puisque c'est l'outil le plus utilisé pour la gestion des versions. Mais il y a toujours un problème avec cet outil malgré son excellente performance: il ne permet pas de gérer les fichiers binaires de grandes tailles. Cela est dû au fait qu'un dépôt Git stock toutes les versions de chacun des fichiers ajoutés dedans. Ainsi, au fur et à mesure qu'un projet avance, l'utilisateur se rend compte de l'inflation de la taille de ce projet et donc les opérations importante que l'on peut effectuer comme les fetch/pull ne seront plus performant. Il faut aussi ne jamais dépasser 10 GB pour chaque dépôt car c'est la taille maximale possible. Heureusement, Git a plusieurs extensions dont on peut se servir quand on a des fichiers de grandes tailles. Les deux outils les plus recommandés sont: git-annex et Git LFS. Git LFS est .... git-annex ...

3.2 Archivage

Maintenant que l'on sait comment gérer les fichiers il faut passer à l'autre étape importante dans ce procès qui est l'étape de l'archivage. Dans le cadre de la recherche, c'est impératif de bien archiver pas seulement ses trouvailles, mais aussi tous les outils utilisés pour y arriver (Il faut mettre en disposition le code source, les données utilisés, les notes détaillant les pistes prises, ...). On ne peut pas faire un puzzle sans avoir toutes les pièces nécessaires. Ainsi, quand un rechercheur archive bien ses travaux, il garantie leur pérennité et assure leur disponibilité pour une communauté. Non seulement cela, mais il peut aussi récupérer un identificateur pérenne pour référencer ses travaux. Il existe plusieurs outils à utiliser pour bien archiver les fruits de son labeur, en particulier: Zenodo, Nakala, et figshare. Les trois sont utilisés dans le domaine de la recherche et permettent de stocker, partager, et préserver les travaux scientifiques. Ils fournissent aussi un identificateur unique pour les bien citer et référencier. Et puisque les trois outils offrent des services similaires, on a décidé de se servir de Zenodo qui est un outil multidisciplinaire (on peut déposer des papiers de recherche, des datasets, des logiciels, des rapports, ...) développé par OpenAIRE ( ) et exploité par CERN ( ). C'est l'un des entrepôts les plus utilisés dans tous les domaines de recherche qui ne coûte rien et qui permet d'avoir des dépôts de 50 GB (`https://public.tableau.com/app/profile/bibdesponts/viz/tableauDATAv2_0/Tableaudebord1`). Maintenant que l'on connait les deux parties imporantes pour bien gérer les gros fichiers et les archiver, il faut forger une liaison directe entre elles. Un souci que l'on rencontre c'est que l'archivage est une opération manuelle qui se fait sur les plateformes d'archivage sans passer par un mécanisme d'automatisation. C'est donc parfois compliqué de bien gérer les versions de ses fichiers en local avant de les déplacer vers un entrepôt d'archivage. Une solution possible est d'utiliser des raccourcis entre Zenodo et un serveur de Git. En effet, il y a un shortcut entre Zenodo et github où les deux comptes de l'utilisateur sont connectés pour lui permettre de mettre ses projets github directement sur Zenodo et de les archiver facilement. Même si ce méchanisme est facile à utiliser et garantie une automatisation du processus, il y a toujours le problème de la taille des dépôts qui sont hébergés sur Github. Un autre problème c'est le fait que ce raccourci est personnalisé pour Github, et donc on ne peut pas faire cela avec des autres plateformes comme gitlab sans passer par des bibliothèques (link). Et même quand utilise une bibliothèque pour faire ce lien Zenodo-Gitlab, il y a toujours un problème puisque cette méthode permet juste de publier des fichiers sur Zenodo en utilisant l'API et ne permet pas de faire plus que ça (on ne peut pas par exemple

récupérer des fichiers dans l'autre sens). La proposition que j'ai est donc de commencer par git (sans passer par ses serveurs) et de construire un chemin vers Zenodo. C'est ce que l'on va faire git-annex en s'appuyant sur le mécanisme des remotes. Un special remote de git-annex c'est un backend que l'on peut utiliser pour transferer les données. Les commandes git-annex permettent de contrôler le déplacement de ces données et de savoir où elles sont à chaque moment. Il y a déjà une dizaine de remotes qui sont développés par git-annex et sont prêts pour être configurés et utilisés (ex: adb, Amazon S3, git lfs, ...) , mais Zenodo ne figure pas dans cette liste. On va donc implémenter un special remote git-annex pour Zenodo qui va répondre à toutes nos attentes.

4 Contributions (TOEDIT)

4.1 Modèle de données

Avant de commencer l'implémentation du remote, il y avait quelques choix à faire pour savoir comment bien répondre à des problèmes qui couvrent le côté git-annex mais aussi l'architecture et le modèle d'un dépôt Zenodo. La première question que l'on s'est posé c'était par rapport aux contraintes sur les tailles et le nombre de fichiers. Puisque l'on a déjà une information sur la taille maximale de tout le dépôt (50 GB mais un utilisateur peut demander d'en avoir plus dans des cas particuliers), il fallait aussi savoir si Zenodo impose des limites sur le nombre des fichiers dans un dépôt. On a contacté Zenodo pour poser cette question, et en attendant la réponse, on a aussi fait des tests où on a déposé des milliers de fichiers de différentes tailles. La réponse était positive et c'est donc possible de stocker un nombre indéfini de fichiers mais la taille du dépôt ne doit pas atteindre 50GB. C'est la seule limite imposée par Zenodo. Le deuxième problème s'est avéré lors de la conception du remote; Il fallait faire un choix de mappage remote git-annex / dépôt Zenodo. Les deux entités sont différentes et alors le fonctionnement final de notre mécanisme de gestion et d'archivage de données dépend de comment on décide de relier les deux concepts. Un dépôt sur Zenodo c'est un récipient où on peut mettre des fichiers de différents types et que l'on peut publier à la fin pour archiver les fichiers. De l'autre côté, un remote git-annex est un dépôt distant qu'il faut initialiser et configurer afin de l'utiliser pour gérer les données. On peut donc initialiser plusieurs remotes dans un répertoire de fichiers et on peut choisir les fichiers à stocker dans un remote, et ceux à laisser en local. Donc pour faire le mapping git-annex / Zenodo, on avait deux possibilités: avoir une implémentation bijective 1-to-1

où on associe chaque dépôt Zenodo à un remote git-annex, ou une relation surjective où l'utilisateur choisit le nombre de dépôt Zenodo à lier à un seul remote. La première option paraît la plus logique puisqu'elle permet d'éviter les problèmes de confusion entre les dépôts Zenodo qui peuvent d'avérer. L'utilisateur peut également créer un autre dépôt Zenodo avec un remote nouveau remote git-annex s'il le souhaite; c'est toujours possible d'initialiser plusieurs remotes git-annex dans la même directory. Le troisième problème est purement architecturel; Zenodo a une architecture plate et donc on n'a pas de notion de répertoire dans un dépôt. Donc il fallait bien penser à comment structurer le dépôt pour pouvoir retrouver facilement les fichiers que l'on met dessus. Heureusement, et grâce à git-annex qui relie chaque fichier annexé à une clé unique, on a pu trouvé comment bien structurer le dépôt Zenodo. Au lieu de laisser les noms des fichiers que l'on a en local quand on fait un upload sur Zenodo, on a décidé de remplacer les noms des fichiers par les clés. Et puisque l'on a un lien unique entre le remote et le dépôt Zenodo, on peut facilement retrouver les fichiers que l'on veut et les récupérer en local.

4.2 Implémantation de remote Zenodo: le backend

4.2.1 Introduction à l'API REST Zenodo

Afin de se communiquer avec Zenodo pour effectuer des opérations sur les fichiers, il faut utiliser son API. La première partie du process est donc de comprendre comment elle fonctionne et de la tester. On a fait les tests de manière chronologique en suivant le tutorial mis en disposition par Zenodo (link). On a donc créé une clé qui permet d'authentifier l'utilisateur, et on a commencé par créer le dépôt pour héberger les données et de les déposer dedans avec des requêtes HTTP. On a aussi testé des autres opérations importantes telles que la suppression des fichiers, la récupération d'un fichier en utilisant son identifiant unique, la publication d'un dépôt, la création des nouvelles versions d'un dépôt publié. En effet, les deux dernières opérations sont très imporantes pour nous puisque l'archivage d'un fichier commence par sa publication (on obtient le doi) et la création d'une nouvelle version d'un dépôt permet de l'évoluer en gardant un identificateur pour chaque changement. Au final, on a pu trouver un flow des requêtes API à lancer pour pouvoir avoir un mécanisme logique qui permet un utilisateur de garder une évolution gracieuse au cours de sa recherche:

INSERT THE FLOW CHART

4.2.2 AnnexRemote: la bibliothèque python utilisée

Maintenant que l'on peut facilement communiquer avec Zenodo et que l'on a un blueprint de comment on veut structurer notre backend Zenodo, il faut commencer son implémentation. Afin d'implémenter un remote git-annex, il faut d'abord être sûr que son progrmeme implémente bien le protocole 'external special remote' de git-annex qui fait le lien entre git-annex et un remote externe. En effet, les deux bout de la communication échangent des requêtes et des réponses durant la période de l'exécution du programme, et donc pour ne pas avoir des soucis de confusion des intéractions, à chaque fois l'une des deux parties prend l'initiative en n'envoyant que des requêtes et l'autre partie répond alors avec des reponses à ces requêtes. C'est pour cette raison qu'il faut avoir un programme qui répond bien à ces contraintes. On utilise donc la bibliothèque **AnnexRemote** de python qui implémente la totalité du protocole et respecte toutes ses spécifications. Il faut donc juste importer cette bibiliothèque dans notre programme et définir une classe *ZenodoRemote* qui extend la classe *SpecialRemote* (implémentée par **AnnexRemote**). Ensuite, on implémente les fonctions de la classe avec les fonctionnalitées qui sont uniques à notre backend Zenodo. Toutes les fonctions du création du dépôt, suppression des fichiers, obtention d'un fichier, .. sont définies, mais pour tout ce qui reste (par exemple, la création d'une nouvelle version) c'est à nous d'ajouter.

4.2.3 Les opérations principales du remote

Chaque remote Zenodo doit être capable d'exécuter des opérations principales qui servent à envoyer les fichiers sur le remote, les manipuler, et les récupérer en local. Tout cela se fait avec les fonctions du programme principal git-annex-remote-zenodo. Voici les opérations essentielles que l'on a implémenté dans le programme principal:

Création du dépôt C'est la première étape du procès qui se fait une fois pour chaque remote, on l'implémente donc lors de l'initialisation du remote (dans la fonction `initremote` de la classe). On s'appuie sur la clé donnée par l'utilisateur, ainsi que son choix Sandbox (FN) ou non, pour envoyer une requête POST à l'API demandant la création du dépôt. On récupère ensuite l'identifiant unique du dépôt ainsi que d'autres informations (comme le lien à utiliser pour déposer les fichiers), et on les stoque dans le fichier des configurations de git-annex. On stoque aussi la clé unique de l'utilisateur pour ne pas lui demander à chaque fois de la donner.

Envoi d'un fichier Cette opération peut s'exécuter plusieurs fois par l'utilisateur lors de sa recherche, puisqu'elle permet de stoquer les fichiers dans un autre endroit où ils sont disponibles à tout moment pour être observés ou récupérés. On implémente cette fonctionnalité dans la fonction `transfer_store` de la classe. Pour commencer l'envoi des fichiers, il ne faut d'abord le lien vers le dépôt que l'on récupère facilement avec la fonction `getconfig` de l'annex. Après, on exploite le fait que git-annex donne à chaque fichier annexé un identificateur unique (une clé SHA1), et on utilise donc cet identificateur comme nom quand on dépose un fichier sur Zenodo. Ce choix d'implémentation nous permet de garder un lien direct entre git-annex et Zenodo sans devoir passer par autres étapes supplémentaires d'identification. On sait qu'un fichier File1 qui a un identificateur Key1 et qui est annexé en local est le même que le fichier Key1 qui est dans le remote. Et puisque git-annex s'appuie principalement sur les identificateurs des fichiers pour les manipuler, maintenant, quand veut chercher un fichier dans le remote, on peut faire ça directement sans devoir chercher le fichier qui est relié à cet identificateur.

Récupération d'un fichier Afin de récupérer un fichier qui sur Zenodo en local, on peut simplement faire une requête GET de l'API pour récupérer la liste des fichiers qui sont dans le dépôt. Après, on peut chercher le fichier dont le nom correspond à la clé git-annex que l'on veut récupérer. Une fois trouvé, on peut récupérer l'identificateur Zenodo donné à chaque fichier stoqué dessus, et on utilise cet identifiant pour télécharger ce fichier. On ne peut pas directement télécharger un fichier sans connaître son identificateur Zenodo unique. Cet identificateur est donné lors du stockage du fichier sur Zenodo et est différent de l'identificateur git-annex que l'on utilise pour renommer le fichier.

Vérification de l'existence d'un fichier Cette opération se fait plusieurs fois durant le procès puisqu'elle est exécutée par git-annex à chaque fois que l'on cherche à savoir l'état d'un fichier. Elle est donc lancée quand on veut déposer un fichier (pour être sûr qu'il n'y est déjà pas), quand on veut le récupérer, et quand on veut savoir où il est (la commande 'git-annex whereis' par exemple). En principe, on parcours la liste des fichiers qui sont disponibles sur le dépôt en comparant la clé git-annex donnée avec le nom des fichiers et on renvoie ua final un booléan pour informer git-annex de l'existence ou non de ce fichier dans le remote.

Suppression d'un fichier Afin d'envoyer un fichier, on s'assure déjà qu'il

est disponible sur le remote (s'il n'est pas là, on ne fait rien, et on ne considère pas ça comme erreur). On récupère donc la liste des fichiers disponibles dans le dépôt et on envoir une requête DELETE à l'API avec l'identifiant unique de ce fichier.

4.2.4 Les tests effectués

Après chaque opération effectuée, s'il y a eu des problèmes, on évoque une exception RemoteError avec le souci rencontré. On s'appuie sur les codes retournés dans les réponses de l'API pour savoir le status de la requête. Pour chaque opération, un code définit un état unique et donc on peut imprimer l'erreur dans les messages de debug pour l'utilisateur. C'est grâce à ces messages que l'on peut donc savoir la source du problème (si cela parvient juste de la requête ou si c'est un problème interne à Zenodo). Donc lors des tests de fonctionnement du backend, et grâce à l'inclusion d'un mode debug, on a pu s'assurer de la correction des opérations et de la cohérence entre git-annex et l'API Zenodo. Il y a eu des tests élémentaires pour chaque partie du programme pour gérer les petites tâches avant de passer aux tests complets où on a effectué toutes les opérations possibles sur le remote. Les traces qui informent le déroulement de ce procès peuvent être observées dans le fichier `journal.org` [fn: lien] où j'ai rédigé toutes les notes qui concernent ce projet et les tests effectués tout au long du stage avec les résultats trouvés.

4.3 Archivage

4.3.1 Archivage direct des données

Quand la première partie de la gestion des données finit, et on stocke tous les fichiers qui nous intéressent dans le remote, il faut maintenant passer à la deuxième partie de l'archivage qui se fait indépendamment de la première, et où on finalise son dépôt avec toutes les méta-données nécessaires avant de le publier. Dans notre programme d'archivage `git-annex-disableremote.py`, on a décidé de diviser les principales étapes de l'archivages en trois parties logiques: la publication du dépôt, la transformation des fichiers en remote web, et finalement la suppression du remote en local. Chacune de ces étapes joue un rôle intrinsèque et la succession des trois est ce que garantie l'archivage de notre dépôt.

La publication du dépôt Afin de publier un dépôt sur Zenodo, il faut d'abord donner des informations sur ce dépôt. On donne ainsi le choix à l'utilisateur de choisir la manière dont il veut fixer les méta-données:

soit il donne le path d'un fichier zenodo.json qui contient déjà les métadonnées, ou il les donne manuellement sur le terminal en répondant aux questions posées par le programme, ou il les configure directement sur Zenodo. On fait des tests après pour s'assurer que elles sont bien données, et on passe à l'étape suivante de la publication. C'est maintenant que l'on utilise l'opération *publish* de l'API pour finaliser la publication.

La transformation de fichiers en un web remote Cette étape est implémentée pour ajouter les fichiers que l'on vient de publier dans un deuxième remote avant de supprimer ce remote (On veut que git-annex aie au moins deux copies de chacun des fichiers). Si on passe pas par cette étape, l'utilisateur perdera le lien direct git-annex <-> Zenodo pour ces fichiers. C'est pour cette raison que l'on reprend la liste des fichiers (leurs noms ainsi que la clé git-annex) et que l'on récupère les liens de téléchargement de chacun des fichiers avant de les ajouter à un remote web avec la commande 'git-annex addurl'. Maintenant, et grâce à cela, tous les fichiers sont toujours enregistrés commes des copies dans l'annex même après la suppression du remote Zenodo.

La suppression du remote en local On s'appuie sur un fichier `remote.log` de git-annex pour récupérer le nom du remote afin de le supprimer. Ce fichier est accessible depuis la branche git-annex de Git et est utilisé pour stocker toutes les informations concernant les remotes git-annex. On peut retrouver le nom du remote (que l'on lui a donné lors de l'initialisation) grâce à l'identificateur du dépôt Zenodo. Une fois trouvé, on utilise la commande 'git remote remove' pour supprimer le remote.

4.3.2 Création d'une nouvelle version après l'archivage d'un dépôt

Cette opération n'est pas possible que si on essaye de créer un nouvelle version d'un dépôt déjà publié. C'est l'outil qui permet de faire évoluer ses fichiers même après publication. On peut donc créer une nouvelle version d'un dépôt quand on finit toutes les étapes de publications simplement en initialisant un nouveau remote et en donnant l'identificateur du dépôt que l'on veut utiliser pour créer la nouvelle version. L'option à utiliser est newversion=id et notre programme prend soins de toutes les opérations possibles comme il aurait fait avec un nouveau dépôt.

4.3.3 Stockage d'une archive de données dans un autre dépôt comme copie

Il y a aussi une étape que l'on fait au début de l'opération de l'archivage qui est la création d'une archive contenant les fichiers et le stockage de cette archive sur un nouveau dépôt Zenodo. Cette opération se fait indépendamment de git-annex et permet ainsi d'avoir un autre copie des données dans un dépôt accessible par l'utilisateur seulement sur le site web de Zenodo. L'utilité de cette opération est de permettre à l'utilisateur de garder une copie qu'il peut récupérer quand il veut sans passer par git-annex. Il y a des autres fichiers qui sont stockés dans ce dépôt autre que l'archive: `git-annex-info.json` et `restore_archive.py`. Le premier fichier contient des informations sur les fichiers tels que leurs liens de téléchargements, leurs identifiants, et leurs noms. Tandis que le deuxième est un script python à lancer par l'utilisateur pour restorer les fichiers de l'archive.

4.4 Restauration d'une archive

La restauration d'une archive se fait grâce au script `restore_archive` que l'utilisateur peut télécharger depuis le dépôt Zenodo avant de le lancer. Comme options, il faut fournir la clé Zenodo, l'option de restauration, et il faut aussi indiquer si c'est le site officiel Zenodo qui est utilisé ou le sandbox. Au début, le programme télécharge l'archive et le fichier `git-annex-info.json` depuis le dépôt avant d'extraire son contenu. Maintenant, on se trouve avec des lien symboliques cassés (git-annex utilise des liens symboliques pour pointer à où les fichiers sont stockés). Il faut donc restorer le contenu des fichiers maintenant. Les trois options possibles de restauration sont:

L'option simpledownload On supprime les liens symboliques en les remplaçant par les fichiers que l'on télécharge grâce aux liens stoqués dans le fichier json. Au final, l'utilisateur se retrouve avec ses fichiers qui sont maintenant disponibles dans un dossier simple.

L'option rebuildannex Dans ce cas, au lieu de remplacer les liens symboliques par les fichiers, on crée des dossiers dont les paths sont ceux où pointent les liens symboliques. On peut récupérer les paths grâce au fichier json où on a stocké les informations. Au final, les liens qui étaient cassés sont maintenant fonctionnels de nouveau et ils pointent vers des fichiers qui sont stockés ailleurs.

L'option usegitannex Cette option est pour un utilisateur qui compte repasser à git-annex lors de la récupération des fichiers. L'idée est

donc d'initialiser un répertoire Git et git-annex où on ajoute tous les fichiers après leurs restauration (la restauration se fait de manière simple comme en **simpledownload**). Une fois les fichiers ajoutés en annex, on les ajoute aussi à un remote web pour garder une dexuième copies en externe. Puisque l'on initialise git-annex, les fichiers donc auront des nouvelles clés git-annex.

Quand on finit la restauration des fichiers, le programme supprime l'archive et les deux autres fichiers utilisés.

5 Evaluation (TODO)

6 Méthodologie et compétences développées (TODO)

6.1 Méthodologie

6.1.1 Documentation de l'ensemble du processus

6.1.2 Gestion du projet

6.2 Compétences développées

7 Conclusion (TODO)

8 Bibliographie (TODO)

9 Annex (TODO)