TP4 : Dépôts distribués

Premiers pas avec les dépôts distants

Clonage de dépôt

Le travail distribué avec git commence souvent par le clonage d'un dépôt existant. Un clone est une copie partielle d'un dépôt git, qui n'en reprend que les éléments suivantes :

- Les commits de l'historique du dépôt d'origine
- Ses références locales (branches, étiquettes), représentées par des références distantes au sein du clone
- Sa branche active, dont une version locale est créée au sein du clone

Parmi les éléments qui ne seront **pas** recopiés dans le clone, on peut mentionner le répertoire de travail, l'index, les connexions éventuelles du dépôt à d'autres dépôts distants, ou la configuration de git.

Il y a cependant aussi une chose qui est présente dans le clone mais n'existait pas dans le dépôt d'origine, c'est une connexion vers ledit dépôt, la *remote* `origin`. Celle-ci permettra de synchroniser le clone avec son parent.

Pour mettre ces concepts en pratique, commençons par créer un clone du dépôt que nous avons manipulé lors du TP précédent :

```
$ cd ~/Bureau
$ git clone MesBranches MonClone
$ cd MonClone
```

Si vous lancez `gitg` dans les deux dépôts, et basculez l'affichage en mode "Tous les commits", vous constaterez que l'historique du clone est identique à celui de son parent en termes de commits (y compris au niveau des *hashes*), mais que la structure des branches est un peu différente.

Les branches du parent sont reflétées dans le clone sous forme de branches distantes (nom en `origin/<branche>`, couleur verte dans gitg). De plus, la branche active du parent a ausi été recréée sous forme de branche locale dans le clone, et c'est cette branche-là qui est actuellement active et extraite dans le répertoire de travail :

```
$ git status
```

Notez aussi dans la sortie de `git status` qu'un lien a été créé entre cette branche locale et la branche distante. Cela permettra par la suite de synchroniser facilement ces deux branches avec les commandes `git push` et `git pull`.

Vous pouvez afficher la liste des dépôts distants associés au dépôt local actuel avec `git remote` sans arguments :

```
$ git remote
```

En mode verbeux, cette commande affiche aussi l'addresse du dépôt distant. Ici, cette addresse est un chemin d'accès sur le système de fichier local :

```
$ git remote -v
```

On peut aussi afficher des informations plus détaillées sur une *remote* avec `git remote show` :

```
$ git remote show origin
```

Récupération de changements depuis le parent

Maintenant que nous avons créé un clone, retournons donc un instant dans le parent pour créer un commit :

```
$ cd ../MesBranches
$ echo "Nouveau dans votre ville !" > ToutNouveau.txt
$ git add ToutNouveau.txt
$ git commit -m "Ajout d'un nouveau fichier"
```

Si vous retournez dans le clone, vous constaterez que ce commit n'est pas visible, ni dans la branche locale, ni dans la branche distante :

```
$ cd ../MonClone
$ cat ToutNouveau.txt
$ git diff origin/univers-parallele
```

C'est parfaitement normal : comme nous en avons discuté, git n'échange avec les dépôts distants que lorsqu'on lui demande explicitement de le faire. Pour récupérer les dernier changements du dépôt parent, il faut en faire la demande, par exemple avec `git fetch` :

```
$ git fetch origin
```

Notez que git mentionne une activité au niveau de la branche `univers-parallele`. A ce stade, le changement n'est toujous pas présent dans la branche locale, mais il est visible dans la branche distante :

```
$ cat ToutNouveau.txt
$ git diff origin/univers-parallele
```

Nous pouvons intégrer ce changement à notre branche locale par une fusion de branche, qui s'effectuera en avance rapide puisque notre branche locale n'a pas divergé par rapport à la branche distante :

```
$ git merge --ff-only origin/univers-parallele
$ cat ToutNouveau.txt
```

Le raccourci pull

Il est très courant de vouloir se mettre à jour par rapport à une branche distante, et `git` fournit donc un raccourci pour accélérer cette opération.

Le raccourci `git pull` est équivalent à un `git fetch` suivi d'un `git merge` de la branche amont (*upstream*) associée à la branche active. Dans notre cas, il s'agit de `origin/univers-parallele`.

Effectuons un autre commit dans le dépôt parent pour tester cette commande.

```
$ cd ../MesBranches
$ echo "Demandez le programme !" >> ToutNouveau.txt
$ git add ToutNouveau.txt
$ git commit -m "Du contenu supplémentaire"
$ cd ../MonClone
$ git pull
$ cat ToutNouveau.txt
```

`git pull` accepte en paramètre les mêmes options que `git merge`, il est donc notamment possible de lui indiquer qu'on ne souhaite fusionner que si il n'y a pas eu de divergence (`git pull --ff-only`), ou au contraire qu'on souhaite créer un commit de fusion même si git pourrait s'en passer (`git pull --no-ff`).

Comme `git pull` fait (notamment) appel à `git merge`, cette commande modifie le répertoire de travail. Il est donc recommandé de l'utiliser avec un répertoire de travail propre.

Notion de branche amont

La branche amont est un concept git qui vise à associer une branche locale avec une branche distante afin d'activer des raccourcis de synchronization comme `git pull`. Une branche amont est configurée automatiquement dans un certain nombre de cas, par exemple quand on effectue un clone.

Un autre raccourci bien pratique pour créer une branche locale connectée à une branche distante est de demander un `checkout` d'une branche locale qui n'existe pas encore, mais dont un homologue distant existe :

```
$ git checkout master
```

Quand on exécute cette commande, git commence par chercher une branche locale `master`, et constate qu'il n'en existe pas, mais qu'il existe une branche distante du même nom, `origin/master`. Il crée alors une branche locale `master` à l'emplacement actuel de cette branche distante, et configure la branche distante `origin/master` comme amont de la brance locale `master`. On peut dès lors utiliser sur master des raccourcis comme `git pull`, ou certaines formes de `git push` qu'on verra plus loin.

On peut visualiser les branches distantes paramétrées comme branches amont des branches locales avec le mode très verbeux de `git branch -list` :

```
1 $ git branch --list -vv
```

Dans les rares cas où il est nécessaire de configurer manuellement l'amont d'une branche locale, la commande git branch fournit des moyens de le faire. Consultez sa documentation pour plus de détails.

Soumission de changements avec push

De la même façon qu'il est possible de se synchroniser avec les branches et commits d'un dépôt distant avec `git fetch` et `git pull`, il est possible de transmettre des informations en sens inverse et d'ajouter des modifications à un dépôt distant avec `git push`.

Comme `git pull`, et contrairement à `git fetch`, `git push` est normalement utilisé pour travailler sur une branche ou une étiquette à la fois. On ne cherche généralement pas à modifier toutes les branches du dépôt distant, mais seulement celle sur laquelle on est en train de travailler. Cela rejoint un grand principe de la gestion de version distribuée, qui est qu'une synchronisation locale est en général préférable à une synchronisation globale.

La forme la plus explicite de `git push` prend en paramètre une *remote* et une référence (étiquette ou branche), puis essaie de créer une version distante de cette référence, ou de la mettre à jour si elle existe déjà :

```
1 $ git push origin master
```

En l'occurence, il ne se passe rien, puisque nous n'avons pas modifié la branche **master** depuis la dernière fois que nous nous sommes synchronisés avec le dépôt **origin**.

Il est courant de vouloir pousser la branche active. Un premier raccourci de `git push` consiste donc à ne préciser que la *remote* cible. C'est alors la branche active qui sera soumise à cette *remote*.

```
1 $ echo "Hallo!" > HelloWorld.txt
2 $ git add HelloWorld.txt
3 $ git commit -m "Traduction␣allemande"
4 $ git push origin
```

Pour les versions récentes de git, l'erreur suivant peut apparaître :

```
1 ...
2 ! [remote rejected] master -> master (branch is currently checked out
3  error: failed to push some refs to '...'
```

L'erreur vient du fait que vous avez deux référentiels, l'un est **origin** que vous avez créé en premier et l'autre celui que vous aviez cloné. En ce moment, vous êtes dans votre référentiel de travail et utilisez la branche **master**. Mais vous êtes également "connecté" dans votre référentiel **origin** à la même branche master. Git craint que vous ne vous trompiez parce que vous travaillez peut-être sur **origin** au même temps. Vous devez donc revenir au référentiel **origin** et effectuer un `git checkout` sur une autre branche, et maintenant vous pouvez lancer la commande `git push` sans problème. Si nous nous rendons maintenant sur le dépôt **origin**, les changements y ont été propagés par `git push` :

```
1 $ cd ../MesBranches
2 $ git log master
```

Enfin, la forme la plus concise de git push ne précise même pas la remote cible. Celle-ci est alors déduite de la branche amont associée à la branche active.

```
$ cd ../MonClone
$ echo "Konnichi-wa!" > HelloWorld.txt
$ git commit -am "Traduction japonaise"
$ git push
```

Pour utiliser cette variante de `git push`, vous devez avoir défini une branche amont associée à la branche active. Si ce n'est pas déjà fait, un raccourci existe pour configurer une branche amont et pousser vers cette dernière : la commande `git push --set-upstream <remote> <branche>`, l'option `--set-upstream` pouvant s'abbrévier en `-u`.

Aller plus loin avec les *remotes*

Dépôts distants multiples

Rien n'interdit de travailler avec plusieurs *remotes* à la fois, et cela s'avérera très utile pour certaines pratiques de gestion de version distribuée que nous étudierons dans la suite de cette formation.

Créons donc un deuxième clone de notre dépôt parent, et ajoutons une remote qui pointe vers celui-ci avec `git remote add` :

```
$ git clone ../MesBranches ../MonAutreClone
$ git remote add autre ../MonAutreClone
$ git remote -v
```

Notons qu'ajouter une *remote* ne déclenche pas d'échanges immédiat. Il faut utiliser `git fetch` pour récupérer le contenu actuel de la nouvelle *remote*. Introduisons à l'occasion une variante de `git fetch` bien pratique quand on travaille avec plusieurs dépôts distants qui permet de récupérer les nouveautés de toutes les *remotes* à la fois :

```
$ git fetch --all
```

Vous noterez dans la sortie de `git fetch` que le nouveau clone ne contient que la branche `univers-parallele`, et pas la branche `master`. En effet, un clone ne contient par défaut que la branche active du dépôt d'origine, et l'on n'a accès qu'aux branches "propres" du clone, pas aux *remotes* auxquelles ce dernier est éventuellement connécté.

Pull et push face à la divergence

Même si `git push` et `git pull` sans arguments procèdent d'une logique très similaire (mettre à jour la branche amont par rapport à la branche locale, ou la branche locale par rapport à la branche amont), il y a une grande différence conceptuelle entre ces deux commandes : `git pull` tolère une divergence de la branche locale par rapport à la branche amont, alors que `git push` le rejette.

Mettons-le en évidence en créant une petite divergence d'historique :

```
$ cd ../MonAutreClone
$ git checkout master
$ echo "1" > Compteur.txt
$ git add Compteur.txt
$ git commit -m "Ajout d'un compteur"
$ git push
$ cd ../MonClone
$ echo "Prost!" > HelloWorld.txt
$ git add HelloWorld.txt
$ git commit -m "Disons bonjour plus joyeusement"
$ git pull
```

Nous venons de créer une divergence d'historique puisque la branche `master` de `MonAutreClone` contient désormais un nouveau commit ("Ajout d'un compteur"), la branche `master` de `MonClone` aussi ("Disons bonjour plus joyeusement"), et aucune de ces deux branches n'est ancêtre de l'autre.

Cependant, `git pull` ne s'en est pas formalisé, et a simplement fusionné les changements de la branche `master` distante dans notre branche `master` locale. On peut observer l'apparition du commit de fusion correspondant dans `gitg`.

En revanche, `git push` a un comportement différent. Si on le met face à la même situation de divergence, il rejettera notre requête :

```
$ git reset --hard @{1}
$ git push autre
```

Pour qu'un `git push` soit accepté, il faut que la branche locale soit un successeur de la branche distante. La divergence n'est pas tolérée, contrairement au cas de git pull où elle est gérée par une fusion. Pour se sortir de ce genre de situation, il faudra d'abord effectuer une fusion ou un *rebase* avant de pousser sa branche. Nous allons bientôt voir pourquoi.

Importance des méthodes de travail

Les ennuis liés à la divergence ne s'arrêtent pas là. Même avec `git pull` et sa fusion implicite, nous pouvons avoir des soucis si la fusion se passe mal. Pour nous en convaincre, aggravons notre divergence par un petit conflit de fusion :

```
$ echo "2" > Compteur.txt
$ git add Compteur.txt
$ git commit -m "Conflit de fusion avec le compteur de 'autre'"
$ git pull
```

De tels conflits sont particulièrement désagréables à gérer, puisque l'on doit arbitrer entre différentes versions d'une même branche, qu'on ne peut plus synchroniser tant que le conflit n'est pas résolu.

Et malheureusement, sur un projet distribué suffisamment gros, on ne peut pas empêcher que ce genre de chose arrive si tout le monde est autorisé à travailler librement sur toutes les branches.

Des méthodes de travail ont donc été dévelopées pour éviter ce genre d'accident. Elles visent notamment à éviter les divergences d'historique entre versions locales et distantes d'une même branche. Nous présenterons une de ces méthodes en détail par la suite, mais mentionnons préalablement quelques principes que toutes les méthodes ont en commun :

1. A tout instant, une seule personne ou machine doit être responsable d'écrire sur une branche, les autres ne pouvant que lire l'état de celle-ci.
2. Si il existe la moindre chance que d'autres personnes lisent le contenu d'une branche, la personne qui modifie celle-ci doit s'abstenir d'effectuer toute modification d'historique qui pourrait créer des divergences avec ces clients, comme par exemple pousser une branche après un `git rebase`.

La restriction de `git push` décrite ci-dessus permet d'aller dans le sens de ce second objectif, puisqu'elle évite à une divergence locale de se propager sur le dépôt distant, qui est potentiellement partagé avec autrui.

Conclusion

Dans ce TP, nous avons vu comment git, en tant que gestionnaire de versions distribué, est capable de gérer des communications entre plusieurs dépôts, sans qu'il y ait besoin de communiquer à chaque opération.

Il est intéressant de mettre en parallèle la relative simplicité du vocabulaire de commandes utilisé avec la complexité conceptuelle conséquente qu'ajoute la synchronisation de dépôts distribués.

Nous conclurons cette formation en montrant comment l'utilisation de quelques outils supplémentaires, couplés avec des méthodes de travail adaptées, permet de mater cette complexité.

Exercices

1. Créez un clone du dépôt `MonDepot` que vous avez utilisé au TP1 et au TP2
2. Comparez le contenu du clone à celui de dépôt original avec `ls -al` et `gitg`. Comprenez-vous toutes les différences observées ?
3. Créez une branche `test` au sein de `MonDepot` et ajoutez-y quelques commits.
4. Récupérez la branche `test` dans le clone et basculez dessus.
5. Ajoutez quelques commits sur `test` au sein du clone.
6. Ajoutez le clone comme *remote* de `MonDepot` et mettez à jour sa branche `test` locale par rapport aux nouveaux commits du clone.

7. Basculez sur la branche `master` de `MonDepot` et créez-y quelques commits.
8. Fusionnez les nouveautés de la branche `master` de `MonDepot` au sein de la branche `test` du clone.

Antisèche

Créer un clone d'un dépôt (`<chemin>` peut être un chemin local ou une URL) :

```
$ git clone <chemin> [<nom>]
```

Afficher les dépôts distants (*remotes*) et les chemins associés :

```
$ git remote -v
```

Afficher des informations détaillées sur une *remote* :

```
$ git remote show <remote>
```

Ajouter une *remote* au dépôt :

```
$ git remote add <nom> <addresse>
```

Récupérer les changements d'une *remote* :

```
$ git fetch <remote>
```

Récupérer les changements de toutes les *remotes* :

```
$ git fetch --all
```

Récupérer et fusionner les changements de la branche amont :

```
$ git pull
```

. . . sauf si il y a eu divergence entre la branche locale et la branche distante :

```
$ git pull --ff-only
```

Créer ou mettre à jour des références distantes :

```
$ git push <remote> <branche1> [<branche2> ...]
```

Pousser une branche en marquant la branche distante comme branche amont :

```
$ git push --set-upstream <remote> <branche>
```

Mettre à jour l'homologue distant de la branche active sur une *remote* :

```
$ git push <remote>
```

Mettre à jour la branche amont associée à la branche active :

```
$ git push
```

Afficher les branches amont associées aux branches locales du dépôt :

```
$ git branch --list -vv
```

Informations complémentaires

Forcer la mise à jour d'une référence distante

Certaines méthodes de travail couramment utilisées avec Git requièrent de modifier l'historique d'un dépôt distant. Par exemple, il existe des projets qui exigent que toute soumission de modifications soit basée sur la version de développement actuelle, ou que l'historique des modifications soit simplifié pour en faciliter le passage en revue.

Toutes ces modifications impliquent l'utilisation de `rebase`, et donc la création d'un historique local qui diverge de la version distante d'une branche. Si vous tentez d'appliquer ces méthodes, vous constaterez donc rapidement que `git push` rejette vos soumissions, en raison du risque de perturber des collaborateurs qui auraient construit du travail par-dessus les versions distantes de vos branches.

La solution, après vous être assuré que vous êtes bien dans un cas où c'est raisonnable, est de forcer `git push` à accepter la soumission, détruisant ce faisant la version distante précédente de la branche :

```
$ git push --force
```

Détruire une branche distante

Lorsqu'on a fini de travailler sur une branche distante, par exemple parce qu'elle a été intégrée à la version officielle du projet, il est d'usage de détruire celle-ci. Cela évite une prolifération des branches inactives qui peut rendre l'historique d'un dépôt incompréhensible.

On peut le faire en utilisant `git push --delete`, ou sur les anciennes versions de Git son prédécesseur moins lisible, le signe '`:`'.

```
$ git push --delete mondepot mabranche
$ git push mondepot :mabranche
```

Bien entendu, comme avec `--force`, il est préférable de n'effectuer cette opération que lorsqu'on est sûr qu'aucun collaborateur n'est actuellement en train de travailler par-dessus la branche qui est sur le point d'être supprimée.

Branches, étiquettes, et références

Dans ce TP, j'ai principalement parlé de branches. Cependant, il est possible de pousser sur un dépôt distant tout type de référence git, concept qui inclut aussi les étiquettes (*tags*).

```
$ git push <depot> <etiquette>
```

Les étiquettes sont couramment utilisées pour représenter des versions publiées d'un projet. Par exemple, dans un projet logiciel, on utilise couramment des étiquettes "v1.0", "v1.1", "v2.0"... qui correspondent aux versions stables.

La notion d'étiquette annotée, que nous avions brièvement décrite précédemment, permet dans ce contexte d'ajouter des notes de version qui décrivent les nouveautés de chaque version publiée par rapport à la précédente. Certaines interfaces git comme GitHub prennent cela en compte en traitant les étiquettes annotées de façon spéciales: elles sont appelées "Releases", leur description peut être mise en forme avec le langage Markdown, et il est possible d'y attacher des paquets sources ou pré-compilés.