BUT Info R2.03 - CM1 : Gestion de versions

Un travail collaboratif sûr

Un projet informatique implique presque le travail coordonné de plusieurs développeurs travaillant en même temps.

Une conception judicieuse permet la plupart du temps que personne ne travaille simultanément sur la même partie du logiciel :

- L'approche objet structure le système en éléments avec des responsabilités distinctes
- Chaque développeur s'attachant plus particulièrement à un aspect du système, ses modifications journalières sont souvent assez localisées

Même dans ce cas, des systèmes de gestion de codes sources partagés sont utiles pour gagner du temps et réduire les erreurs.

Partager en synchronisant

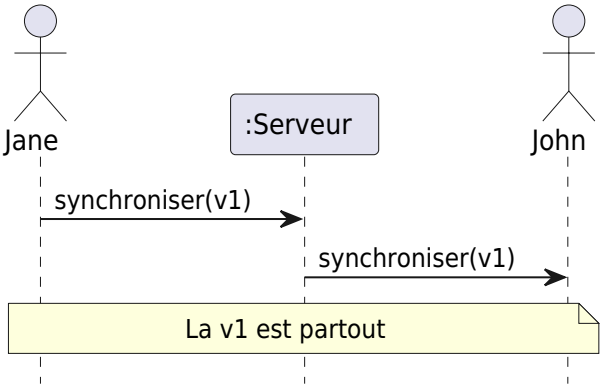
Les systèmes de synchronisation de fichiers sont populaires parce qu'ils sont très répandus pour une utilisation courante avec des fichiers quelconques. Mais s'agissant de partage de code entre plusieurs personnes, on atteint leurs limites en raison de problèmes de conflits.

Imaginons deux développeurs, Jane et John, travaillant chacun sur son poste de travail et ayant accès à un serveur pour synchroniser leur code.

1. Jane crée un fichier dans sa version v1 et le système de synchronisation le transmet au serveur.
2. La version sur le serveur est donc également v1.
3. Le système de synchronisation met à jour le poste de travail de John en ajoutant la version v1

Représentation UML du scénario

UML (Unified Modeling Langage) est un langage graphique de modélisation. Il ne propose pas que les diagrammes de classes *étudiés dans le cours de Conception et de Programmation Objets. UML spécifie la manière de représenter différents aspects d'un système, avec à chaque fois un type de diagramme adapté à ce que l'on souhaite représenter. Il y a 10 diagrammes principaux. Ici, c'est un diagramme de* séquences *qui est employé pour représenter une séquence de messages envoyés entre plusieurs participants interagissant les uns avec les autres.*



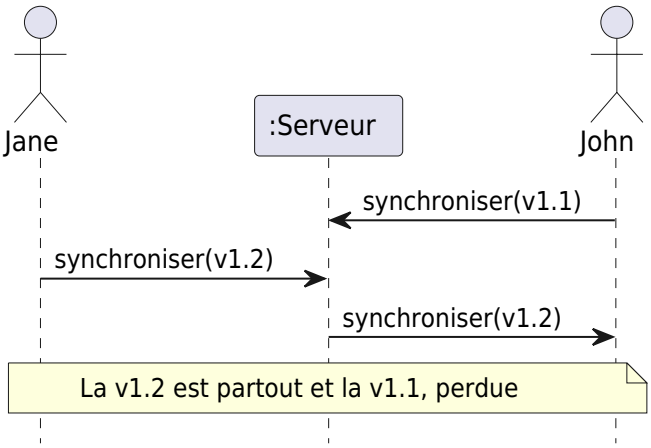
Conflits de synchronisation

A ce point, la version `v1` est sur les deux postes de travail. Supposons maintenant que que :

- John ouvre le fichier et le modifie pour produire la version `v1.1` à partir de la `v1`, sans l'enregistrer pour l'instant
- Simultanément, Jane continue a travaillé sur le fichier en `v1` et produit une `v1.2` différente de celle John

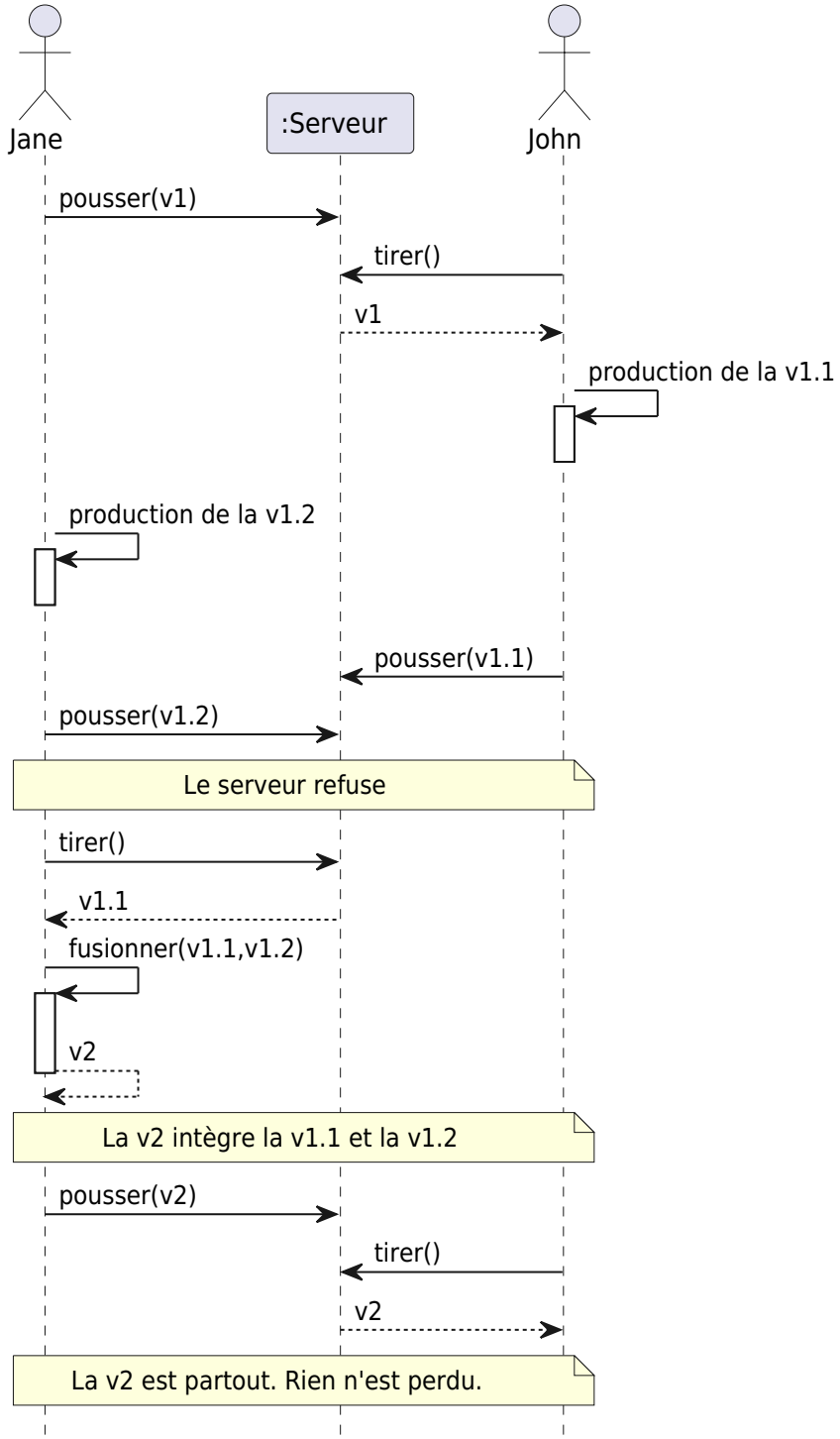
Avec un système de synchronisation comme ceux mis en œuvre dans les “drives” de fournisseurs variés :

1. John enregistre son fichier en `v1.1` avant Jane et ferme son éditeur de texte
2. La version `v1.1` est synchronisée avec le serveur
3. Jane enregistre sa `v1.2`
4. La version `v1.2` de Jane est synchronisée avec le serveur et écrase la `v1.1` de John
5. Le version `v1.2` désormais sur le serveur est synchronisée vers le poste de travail de John
6. Le travail de John est perdu



Processus de gestion de versions

Pour un travail collaboratif , il est nécessaire d'employer des outils de partage adaptés : des *systèmes de gestion de version*. Le processus de travail collaboratif change. Sa modélisation en UML peut être formulée ainsi :



Le processus ci-dessus est approximatif et illustre le principe général de la gestion de version, en ignorant les mécanismes spécifiques de systèmes particuliers. Les flèches pointillées dans le diagramme UML sont les retours à la requête envoyée juste au dessus.

En langage naturel :

1. Jane produit la `v1` et la *pousse* vers le serveur
2. John demande à *tirer* la dernière version et récupère la `v1`. La mise à jour n'est pas automatique : c'est John qui la demande explicitement
3. John produit la `v1.1` et Jane la `v1.2`
4. John *pousse* la `v1.1` vers le serveur. Comme la `v1.1` est produite à partir de la `v1`, le serveur l'accepte
5. Jane tente de *pousser* sa `v1.2` mais le serveur refuse parce que cette `v1.2` n'a pas été produite

à partir de la dernière version, la `v1.1` de John
6. Jane est contrainte de *tirer* la `v1.1` et de la *fusionner* sa `v1.2` pour produire une `v2` intégrant toutes les modifications
7. Jane peut maintenant *pousser* la `v2` et John la *tirer*.

Fusion

La clé de voûte de ce processus est le mécanisme de *fusion*. Heureusement, dans l'immense majorité des cas, cette fusion est automatique. L'efficacité de ce processus dépend donc de celle des algorithmes de fusion. En pratique :

- La fusion automatique emploie des mécanismes similaires à ceux de la commande `diff`.
- Elle fonctionne très bien tant que les utilisateurs ne modifient pas exactement la même ligne d'un programme.
- Ce cas pathologique est assez rare si le travail est correctement réparti au sein d'une équipe.
- Si ce cas survient, la résolution manuelle d'un conflit reste simple mais requiert une communication entre développeurs pour être bien gérée.

En tout état de cause, **les mécanismes de fusion automatique ne fonctionnent que sur des fichiers textuels** tels que des codes source. On évitera de mettre sous contrôle de version des fichiers compilés.

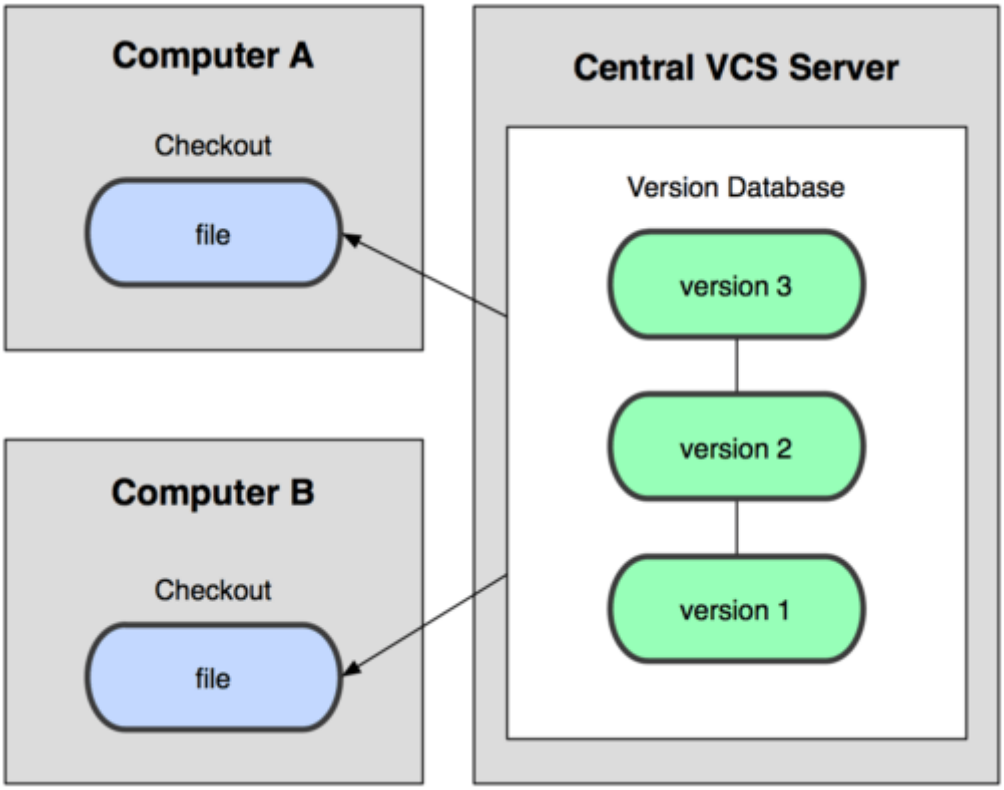
Gestion de versions

Un système de gestion de versions vise à :

- garder un historique des différentes versions des fichiers d’un projet ;
- permettre le retour à une version antérieure quelconque ;
- permettre un accès souple à ces fichiers, en local ou via un réseau ;
- permettre à des utilisateurs distincts et souvent distants de travailler ensemble sur les mêmes fichiers.

Approche centralisée

Chaque développeur ne dispose que d'une copie de travail. Les versions sont centralisées sur un serveur.



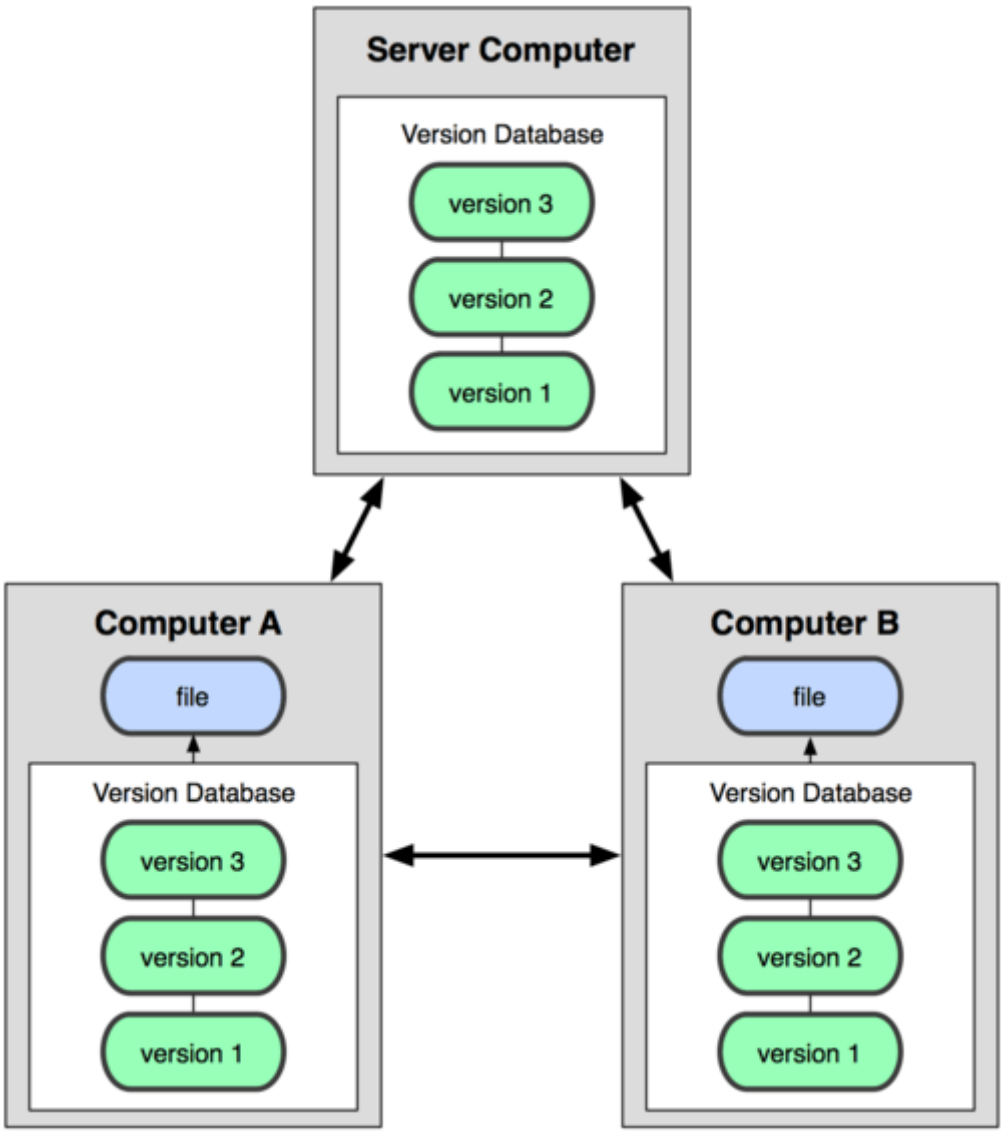
Cette architecture client/serveur est éprouvée, bien comprise et facile à administrer mais souffre de quelques inconvénients, en particulier de risques de corruption de données.

Systèmes emblématiques :

- CVS
- Subversion (SVN) : svnbook.red-bean.com/nightly/fr/svn-book.html
- Perforce

Approche distribuée

Chaque développeur dispose en local d'un historique complet des versions en plus de sa version de travail.



Ceci permet la mise en place de chaînes de traitement qui ne sont pas réalisables avec les systèmes centralisés.

Système emblématique :

- Git : Git-scm.com/book/fr (documentation très complète et lisible)

> Dans le cadre de ce cours, c'est Git qui sera étudié

Git

Serveur Git

Git est distribué et chaque poste de travail dispose d'une copie complète des versions du projet. Ainsi, il est possible de mettre en place un contrôle de version sur une base de pair à pair (P2P). Chaque poste de travail peut ainsi se comporter comme un client ou comme un serveur.

Toutefois, certaines plateformes de partage de code sont très populaires et sont souvent employées comme des serveurs Git, auxquels les différents postes de travail se connectent. Parmi ces plateformes :

- https://github.com/
- https://gitlab.com/

Chaque développeur dispose d'un compte sur la plateforme (comme https://github.com/pierre-gerard) avec l'ensemble des projets auxquels il participe (comme https://github.com/pierre-gerard/hello-world). Chaque projet fait l'objet d'un sous-répertoire sur le compte de l'utilisateur.

Un utilisateur peut participer à un projet existant si un administrateur l'y autorise. Il peut également créer un projet et permettre à d'autres de participer.

> L'utilisation de GitHub est traitée dans le TP correspondant à ce cours.

Poste de travail

Les postes de travail se connectent à des serveurs pour récupérer et ajouter de nouvelles versions. A chaque fois qu'un poste de travail récupère de nouvelles versions, il récupère en fait la base entière avec la totalité des versions. On appelle cette base un *dépôt*.

Dans la mémoire de masse d'un poste de travail (disques durs, SSD etc) :

- Un projet correspond à un dossier/répertoire. On y trouve les fichiers de la version sur laquelle on travaille : le répertoire de travail.
- Dans le cas d'un répertoire existant non encore versionné, il faut indiquer (à l'aide d'une ligne de commande) que le répertoire courant doit être traité par Git

```
[répertoireProjet]$ git init
```

- Dans le cas où on souhaite importer un projet existant sur un server, on parle de *clonage*, et la ligne de commande s'exécute à partir du répertoire parent de celui qui contiendra les fichiers du projet. Cloner un dépôt distant installe la dernière version dans un sous-répertoire qui deviendra le répertoire de travail.

```
[répertoireParent]$ git clone https://github.com/pierre-gerard/hello-world
```

- Il est possible, sur une même machine, d'avoir plusieurs répertoires de travail du même projet distant.

Structure d'un répertoire Git

- Dans tous les cas, les fichiers immédiatement visibles dans un répertoire de travail constituent une copie de travail dans une version particulière.
- Toutes les autres versions sont stockées dans une base de données qu'on nomme *dépôt*. Le dépôt local est stocké dans un répertoire `.git`

> Lorsque l'on modifie un fichier indexé et qu'on crée une nouvelle version du projet, les modifications restent **locales** : la nouvelle version est enregistrée dans le répertoire `.git` et pas directement sur le serveur.

Le répertoire `.git` stocke aussi des fichiers de configuration (authentification, etc.) et des informations sur l'état des fichiers du projet.

Processus Git nominal

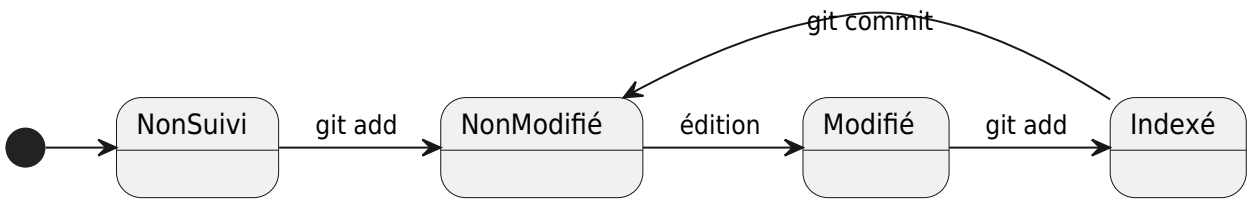
Versionner un fichier

Tous les fichiers dans un répertoire de travail ne sont pas automatiquement pris en compte par Git. Pour qu'un fichier dans un répertoire de travail puisse être ajouté dans le dépôt local et fasse l'objet de nouvelles versions, il faut le spécifier avec la sous-commande `add`.

```
[répertoireProjet]$ git add *.java
```

Etats d'un fichier

La commande `git add` permet de prendre en compte un fichier la première fois, mais elle permet aussi d'*indéxer* un fichier modifié sur le disque. Ce faisant, le fichier est placé dans une *zone d'index* (*staging area* en anglais). L'indexation est une étape intermédiaire avant archivage dans le dépôt local. L'état des fichiers peut être connu à l'aide de la commande `git status`.



Ici, c'est un diagramme d'états en UML qui est employé pour la modélisation. Les états sont ceux d'un fichier et les flèches correspondent à des changements d'état motivés par les événements associés).

Création de nouvelles versions

Si la sous-commande `commit` fait passer un fichier dans l'état *non modifié* c'est parce qu'elle enregistre la nouvelle version du fichier dans le dépôt (la base de données des versions). A chaque fois qu'on l'emploie, il faut spécifier un message pour identifier la version.

```
[répertoireProjet]$ git commit -m "correction du bug 342"
```

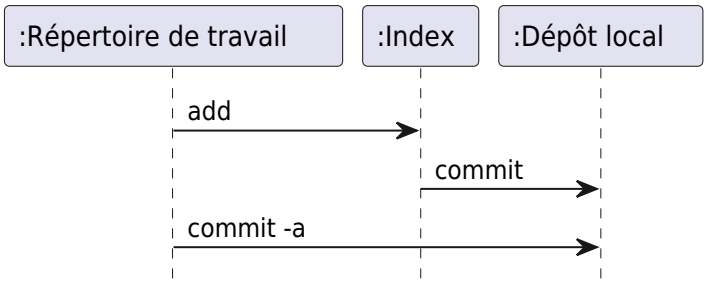
Seuls les fichiers indexés sont traités par `git commit`. Si un fichier est modifié, il faudrait donc en principe faire des `git add` avant chaque `git commit`. L'option `-a` avec `git commit` permet de tout faire en une seule commande, en ajoutant automatiquement à l'index tous les fichiers versionnés qui ont été modifiés.

```
[répertoireProjet]$ git commit -a -m "correction du bug 342"
```

Processus local

Le diagramme de séquences UML est ici employé un peu abusivement. En effet, dans diagramme de

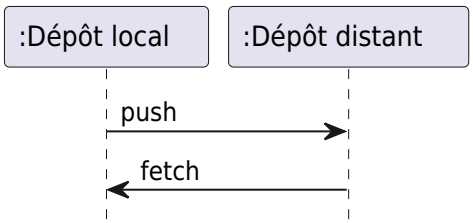
séquences, l'ordre (du haut vers le bas) définit une séquence alors que dans cette synthèse, l'ordre ne compte pas.



Les moyens d'annuler ces commandes sont détaillés dans la documentation :
http://git-scm.com/book/fr/v2/Les-bases-de-Git-Annuler-des-actions

Serveur distant

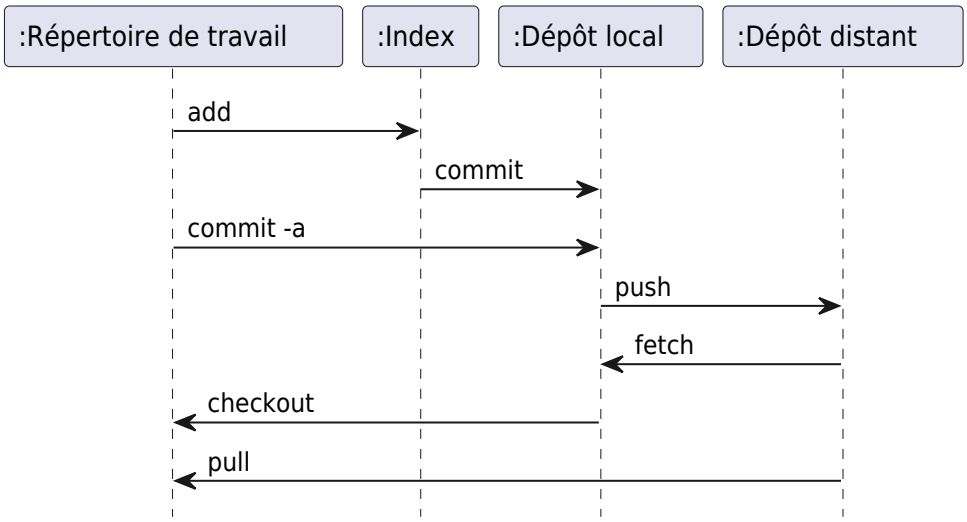
Les versions locales peuvent être propagées vers un serveur distant et les nouvelles versions sur un serveur peuvent être importées localement. Les deux commandes symétriques correspondantes sont respectivement `git push` et `git fetch`.



La commande `git fetch` ne change pas les fichiers du répertoire de travail, mais uniquement le dépôt local (la base des versions) en y intégrant les nouvelles versions du dépôt distant.

Pour rendre les fichiers directement accessibles dans le répertoire de travail, la commande `git checkout HEAD` permet d'y installer la dernière version dans le dépôt local.

Synthèse



Avec les paramètres adéquats, la sous-commande `checkout` permet également d'installer n'importe quelle version dans le répertoire de travail.

La commande `git pull` est un raccourci pour effectuer à la fois un `git fetch` et un `git checkout HEAD`.

Gestion des conflits

Automatique

Dans le cas où on souhaite faire un `push` vers un serveur distant qui aurait déjà intégré dans son dépôt les modifications d'autres personnes, Git détecte un conflit.

> Les conflits sont toujours réglés localement

1. On fait d'abord un `pull` pour récupérer localement les modification distantes
2. Si tout se passe bien, Git fusionne automatiquement les modifications distantes et les modifications locales
3. A ce point, la copie de travail est en avance sur le dépôt distant puisqu'elle intègre les modifications locales en plus des modifications tirées du serveur
4. On peut faire un `commit` puis un `push` pour propager la version fusionnée

Manuelle

Dans le cas où Git ne parvient pas à fusionner automatiquement deux versions, il faut se résoudre à régler le conflit en éditant le fichier manuellement. C'est heureusement assez rare et ne survient que quand deux personnes ont modifié la même ligne du même fichier.

1. On arbitre le conflit manuellement
2. On réindexe le fichier problématique avec la sous-commande `add`
3. On peut désormais faire un `commit` puis un `push`

Branches

Git gère les branches de développement, au sein desquelles on peut créer des versions :

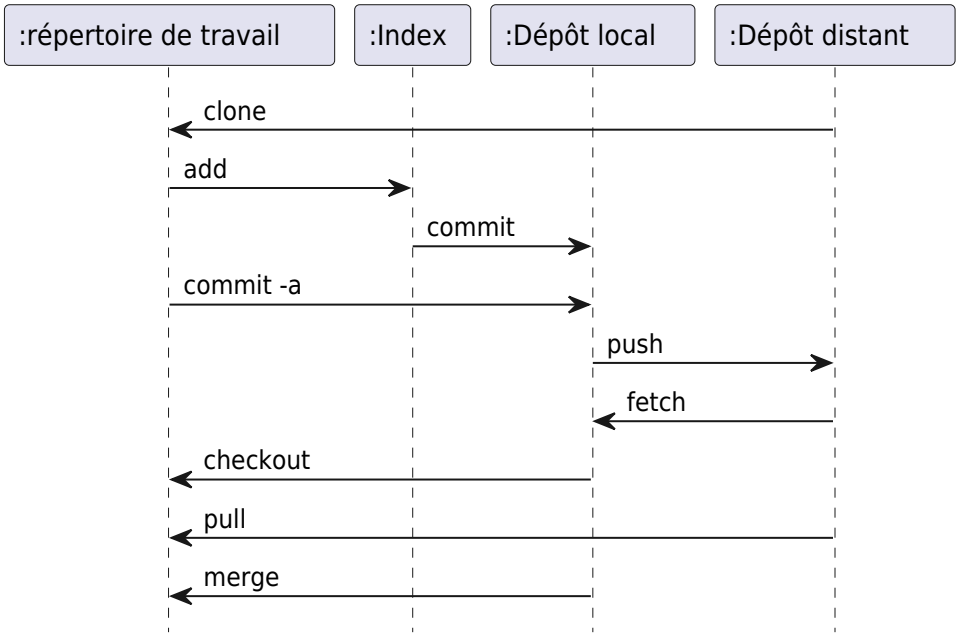
- il y a toujours une branche principale (`master` ou `main`) avec la version nominale du projet et ses versions officielles
- il est d'usage d'utiliser une branche spécifique pour le développement, avec de multiples versions dans cette branche
- la résolution d'un bug peut entraîner la création d'une sous-branche de `master`
- pour développer de nouvelles fonctionnalités, on emploie souvent des sous-branches de la branche de développement (une par nouvelle fonctionnalité)
- etc

Les commandes liées aux branches sont les suivantes :

- `git branch` pour créer une branche à partir de la version courante
- `git checkout` pour installer une branche particulière dans le répertoire de travail
- `git merge` pour fusionner deux branches

Ces commandes n'ont un effet que sur le dépôt local (utiliser `git push` pour propager).

Synthèse des principales commandes



Il existe des interfaces graphiques pour Git mais pour les employer efficacement, il est indispensable de comprendre les mécanismes sous-tendus pas les appuis de bouton.