Gestion de version :

centralisée ou décentralisée ?

Webinaire RI3

16 mars 2010

A. Pérus - LAL

Gestion de version : centralisée ou décentralisée ?

1. Gérer l'historique
2. Quelques pratiques
   - Branches et fusion de branches
   - Intégrité
   - Publication
   - Processus de développement
   - Recherche de bugs
   - Fichiers binaires
3. Quelques caractéristiques techniques
4. Conclusion

Diapo 1: Gérer l'historique avec un VCS ?

- Suivre l'historique et les évolutions.
  - pour chaque changement : **qui**, **quand**, **quoi** et **pourquoi**
  - inspecter ce qui a été changé et être capable de recréer le projet dans l'état au moment du changement.
- Travailler sur des parties distinctes du projet, indépendamment des modifications effectuées par ailleurs : **branches**
- Travailler simultanément sur plusieurs versions du projet
- Retrouver plus facilement les "bugs"
- Maintenir d'anciennes versions

Diapo 2: 'Version Control System' (VCS)

- VCS : *Version Control System*

  un des principaux outils au coeur de la gestion d'un projet

- Deux familles d'outils :

  - CVCS : *Centralized Version Control System*
  - DVCS : *Distributed Version Control System*

Diapo 3: Branches et fusion de branches

Dans un projet à plusieurs, gérer les développements concurrents est un point crucial.

- Dès qu'il y a branches, il faut savoir gérer les fusions
  - entre ses propres branches
  - entre développeurs

  Deux techniques :
  - Verrouillage
  - Fusion

Diapo 4: Branches et fusion de branches

Fusionner peut être dangereux : on peut réintroduire discrètement des "bugs" corrigés ou en créer de nouveaux :

- en travaillant dans des branches séparées, on peut modifier de différentes façons les mêmes parties de fichiers.
- gestion des fichiers renommés ou copiés dans une branche et modifiés sous leur ancien nom dans une autre branche.
- le code d'une des branches peut aussi dépendre d'une fonctionnalité qui a changé dans l'autre branche. Souvent, cette dépendance est évidente : elle casse la reconstruction. Mais parfois, les effets sont beaucoup plus insidieux ...

La façon dont les VCS gèrent les branches et les fusions entre branches est donc très importante.

Diapo 5: Branches et fusion de branches - SVN

Avec SVN on crée une nouvelle branche en copiant une branche existante, puis on récupère une copie locale.
SVN permet de travailler à plusieurs simultanément dans la même branche.

- Chacun récupèrer une copie de travail d'une même révision de la branche.
- Mais tant qu'on ne "commite" pas, le travail n'est pas sauvegardé ...
- Un 1er "commite"
  Mais, tant qu'on n'a pas fusionné son travail avec la nouvelle révision, on ne peut pas soi-même "commiter" ...
  Et si quelque chose se passe mal pendant la fusion ? On n'a pas pu sauvegarder ses modifications, ni leur historique : soit on perd son travail, soit on valide quelque chose d'incorrect.
- En fait, le risque est même plus subtil : on peut modifier quelque chose sans entrer en conflit textuel avec les modifications d'autrui; on peut "commiter", en produisant une nouvelle version que personne n'a vue ou testée.

C'est la façon la plus simple, *et donc naturelle*, de procéder avec SVN.

Diapo 6: Branches et fusion de branches - DVCS

DVCS : pas de serveur central qui stocke les metadata.
Chaque clone contient une copie autonome du projet avec tout l'historique et un espace de travail.

- Si on travaille ensemble sur un projet, on a chacun son dépôt
- Chacun peut cloner celui d'un autre, ou bien un dépôt placé sur un serveur "central".
- Quand on "commite", on le fait dans son dépôt.
- On peut publier ses modifications dans un dépôt ou bien proposer à un collègue de les récupérer directement depuis son propre dépôt.

La récupération de modifications est découplée de la fusion avec les modifications locales.

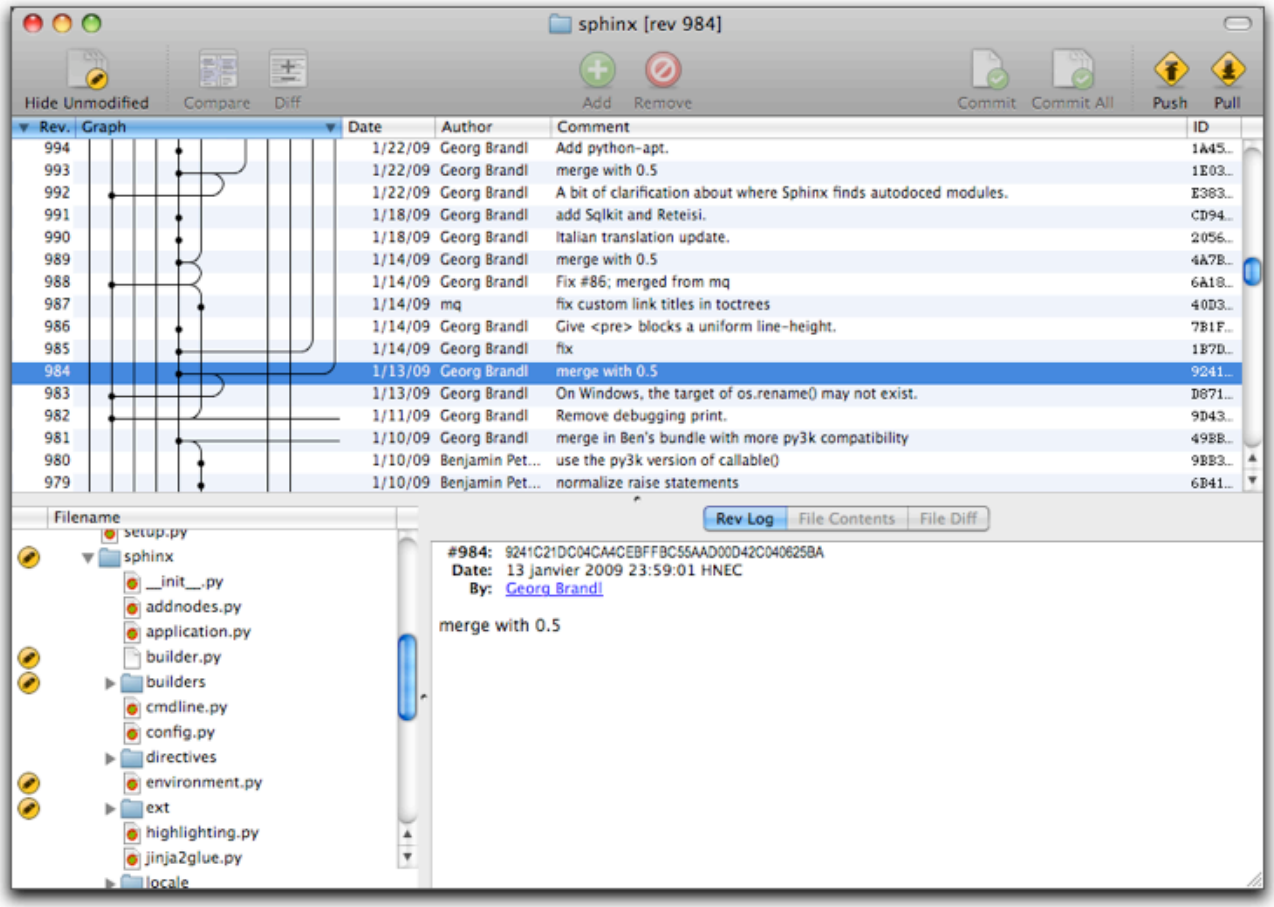
Diapo 7: Branches et fusion de branches

Avec un DVCS, chaque "commit" est potentiellement une branche en lui-même.
Si on démarre à plusieurs à partir de la même révision, alors, chacun crée en fait, lors du 1er commit, un petit "fork" du projet.

Branches et fusions sont complètement naturelles avec les DVCS et donc perçues différemment que dans le monde SVN.
Tant Mercurial que Git peuvent associer un nom aux branches que l'on veut faire durer.

L'historique d'un projet montre bien ce parallélisme et sa structure en branches, avec qui a fait quoi, quand et pourquoi.

Diapo 8: Branches et fusion de branches



Diapo 9: Integrité

SVN n'impose aucune organisation particulière des packages qu'il gère.

- Il n'y a pas de concept de branche autre que ce qu'il propose via un `svn copy`.
  Les branches sont quelque part dans la hiérarchie des fichiers selon les conventions adoptées par la collaboration.
- La plupart des commandes de SVN opèrent uniquement sur la portion du dépôt indiquée :
  - `$> svn co /branches/myfeature`
  - `$> cd myfeature/deep/in/subdir`
  - `$> svn ci`

  Nouvelle version uniquement pour la partie inférieure de la branche !
- `svn update` fonctionne de la même façon : il est tout à fait possible d'avoir des portions de son espace de travail synchronisées avec différentes révisions de l'histoire de la branche.

Diapo 10: Intégrité

Au contraire, les DVCS traitent la totalité du dépôt comme une unité de travail.

- Si on exécute `git commit -a` n'importe où dans le dépôt, toutes les modifications effectuées seront prises en charge et on créera une nouvelle version du projet.
- Avec Mercurial, de façon similaire, `hg update` mettra à jour la totalité de l'espace de travail par rapport à un moment donné de l'historique.

Aucun de ces outils ne peut mettre une branche dans un état inconsistant accidentellement.

Diapo 11 : Publication

C'est l'une des principales différences entre SVN et les DVCS :

- avec SVN, "commiter" une modification la rend public *implicitement* et *instantanément*
- alors qu'avec les DVCS, les 2 opérations sont parfaitement découplées.

Combiner le "commit" et la publication peut convenir si tous les développeurs ont un accès en écriture sur le serveur et si tous sont connectés en permanence sur le même réseau.

Les séparer rajoute une étape supplémentaire, mais ouvre la possibilité de travailler hors connexion et d'utiliser des techniques de publication originales.

Diapo 12: Publication

C'est l'une des principales différences entre SVN et les DVCS :

- avec SVN, "commiter" une modification la rend public *implicitement* et *instantanément*
- alors qu'avec les DVCS, les 2 opérations sont parfaitement découplées.

Dans le cas d'une petite équipe soudée, toujours efficacement connectée, le choix de publier en "commitant" peut être le plus facile.

Dans le cadre d'une collaboration plus lâche et géographiquement très répartie, découpler le "commit" de la publication peut davantage convenir.

Diapo 13: Publication

C'est l'une des principales différences entre SVN et les DVCS :

- avec SVN, "commiter" une modification la rend public *implicitement* et *instantanément*
- alors qu'avec les DVCS, les 2 opérations sont parfaitement découplées.

Ainsi, les CVCS peuvent être bien adaptés au modèle de gestion de projet dans lequel une équipe très structurée est menée avec des règles très contraignantes. Les accès sont alors contrôlés par des responsables et non partagés par des pairs. Les droits d'accès à certaines parties du dépôt peuvent être restreints selon les équipes ou les développeurs.

Les DVCS n'offrent par défaut pas d'autres possibilités que de couper le dépôt en plusieurs sous-dépôts séparés.

Diapo 14: Processus de développement et VCS

Souvent on commence en utilisant un DVCS comme on le faisait avec SVN.

On clone l'un des dépôts principaux et y dépose en retour ses modifications.

Cette façon de travailler est familière et inspire confiance, mais elle n'utilise qu'une toute petite partie des façons d'interagir au sein du projet.

Diapo 15: Processus de développement et VCS

Puisque le modèle distribué s'appuie sur la récupération des modifications en les reportant dans un dépôt local, il favorise la revue de code.

1. X. gère le dépôt qui doit devenir la version 2.4 du projet.
2. Z. lui annonce qu'il a effectué un certain nombre de modifications et qu'elles sont prêtes; il lui donne son url et X. récupère ces modifications très simplement.
3. X. peut alors les analyser et éventuellement demander à Z. de les amender avant de les accepter, de les fusionner et de les publier.

Bien sûr, on peut faire de la "revue de code avant fusion" avec un CVCS; mais par défaut un CVCS n'y incite pas.
Il faut se fixer très explicitement ce type d'étapes et s'y contraindre, alors que c'est complètement naturel avec un DVCS.

Diapo 16: Processus de développement et "Hooks"

Les "hooks" permettent de lancer des scripts à des moments particuliers lors de certaines opérations du VCS : commit, changement de révision, lock (comparable à la notion de "trigger" en SQL).

Ils permettent d'introduire, au niveau du dépôt, des contraintes définies par la gestion du projet.

Les CVCS offrent une plus grande facilité de gestion des "hooks"; par exemple, il est possible de configurer un script en "pre-commit" qui rejettera tout "commit" introduisant une erreur détectée par une suite de tests automatique.

Avec un DVCS, ou pourra mettre en place une solution équivalente sur un dépôt jouant le rôle de dépôt central de référence, mais il sera plus difficile d'empêcher des développeurs d'échanger entre leurs dépôts des modifications à effets de bords malencontreux.

Diapo 17: Recherche de 'bugs'

Généralement, la 1è chose que l'on fait face à un nouveau bug, c'est de parcourir l'historique à la recherche de ce qui a changé récemment et de voir qui a modifié quoi, quand et pourquoi.

Ces opérations sont quasi instantanées avec un DVCS, car les données sont stockées localement sur la machine, alors qu'elles peuvent être relativement longues si on s'adresse à un serveur SVN lointain et occupé.

Diapo 18: 'Bisect'

Bien que le simple affichage de l'historique soit très utile, il est beaucoup plus intéressant encore d'avoir un outil qui permette de mettre directement le doigt sur un bug !

C'est la commande `bisect`.

L'utilisation est assez simple : on lance la commande `bisect` sur une révision dont on sait qu'elle n'a pas le bug et sur une révision contenant ce bug. Par itération et en questionnant l'utilisateur, l'outil est capable de trouver la révision qui a introduit le bug.

Procédure facilement automatisable. Il suffit :

1. d'écrire un script qui reconstruise et qui teste la présence du bug;
2. on lance `bisect`,
3. on revient plus tard et on a la révision ayant introduit le bug sans autre intervention manuelle !

Diapo 19: 'Bisect'

Par ailleurs, la commande opère en log(n) : pour une recherche sur 1000 révisions il posera environ 10 fois la question; si on élargit la recherche sur 10000 révisions, le nombre de questions passera à 14.

Non seulement cette commande `bisect` change complètement la façon dont on recherche des bugs, mais si on écrit systématiquement les scripts qui automatisent son utilisation, on fait d'une pierre deux coups, en développant une suite de tests de régression !

On peut objecter que l'examen de l'historique est beaucoup plus simple avec SVN plutôt qu'avec un DVCS puisque son historique est habituellement beaucoup plus linéaire. Mais cette commande `bisect` n'est disponible que pour Hg et Git …

Diapo 20: Cherry-Picking

Une fois qu'on a trouvé un bug, le corriger est rarement suffisant.

Imaginons qu'il existe depuis plusieurs mois et qu'il y a déjà 3 "releases" à corriger. Chacune de ces versions a elle-même éventuellement sa propre branche de corrections de bugs.
Si l'idée de copier une correction d'une branche à l'autre est simple, la mise en pratique, elle, n'est pas si simple.

Diapo 21: Cherry-Picking

Mercurial, Git et Subversion ont tous la possibilité de "sélectionner à la main" (*cherry-picking*) un ensemble de modifications dans une branche et de l'appliquer sur une autre. Mais c'est assez risqué.

Une modification ne flotte pas comme ça en l'air : elle existe dans un contexte et dépend de son environnement, du code autour.
Et certaines de ces dépendances sont sémantiques si bien que, malgré une fusion réussie, la modification donnera une erreur plus tard.

Lorsque ces dépendances sont simplement textuelles, il suffit d'inspecter la modification et de la corriger avec un éditeur adéquat. Les DVCS proposent un certain nombre de techniques pour résoudre efficacement ce type de problème.

Diapo 22: Cherry-Picking

Par exemple, avec Mercurial ou Git, il y a une alternative possible en utilisant des branches anonymes :

1. on identifie la révision introduisant le bug avec la commande `bisect`;
2. on récupère cette révision; on corrige l'erreur;
3. et on sauvegarde la correction dans une branche sous la révision fautive.

Cette nouvelle modification peut alors facilement être propagée dans toutes les branches touchées par le bug, sans avoir à se coltiner de sélection à la main, en utilisant alors les outils habituels de fusion et de résolution de conflits plus simples et plus fiables qu'une "sélection à la main".

Là encore, rien en théorie, n'interdit d'utiliser une technique similaire avec Subversion; mais l'utilisation de ses branches est beaucoup plus lourde que celle des DVCS et du coup, beaucoup moins utilisée.

Diapo 23: Fichiers binaires

Un des points forts des CVCS est la gestion des fichiers binaires, en particulier quand ils sont gros.

Les fichiers binaires sont difficiles à compresser et ne peuvent être fusionnés.
Chaque révision est pratiquement stockée telle quelle et leur accumulation peut vite devenir importante.

- Un CVCS n'aura à le faire qu'une seule fois,
- alors qu'un DVCS le répétera systématiquement dans chaque clone.

Diapo 24: Fichiers binaires

La fusion de 2 fichiers n'est pratiquement jamais possible - pratiquement aucune application à l'origine de ces fichiers ne fournit de solution pour résoudre ce type de conflits.

- Pour éviter ce type de problème, un CVCS peut bloquer l'accès au fichier en posant un "*lock*", si bien qu'une seule personne pourra l'éditer dans une branche donnée et à instant donné.
- Un DVCS ne pourra pas, de par sa nature même, offrir cette solution et il faudra s'en remettre à des règles de fonctionnement du groupe.

Diapo 25: DVCS et agilité

La possibilité de faire des "commits" locaux facilement encourage le développeur utilisant un DVCS à développer rapidement, par petites modifications.

Imaginons qu'on soit en cours de développement d'un truc compliqué et qu'on ait envie de tester un "refactoring" hasardeux …

Avec un DVCS :

1. on peut "commiter" dans l'état, sans risque d'importuner qui que ce soit et sans trop se préoccuper de savoir dans quel état est le projet et se lancer.
2. Si ça ne donne rien, il est facile de revenir en arrière et de reprendre le cours originel éventuellement en utilisant la commande `rebase` pour éliminer certains des "commits" effectués durant les tests.

Diapo 26: DVCS et agilité

Cette façon de procéder est certainement plus ou moins possible avec SVN, mais l'expérience montre qu'elle est beaucoup plus fréquente avec les DVCS.

Il est probable que le côté privé de la branche, sur son ordinateur, ajouté aux réponses instantanées des DVCS, encourage davantage à une utilisation avancée et généralisée de l'outil.

On peut observer un effet similaire avec les fusions. Dans la mesure où elles sont au coeur de l'activité des DVCS, elles apparaissent beaucoup plus fréquemment que dans les projets gérés avec Subversion.
Bien que toute fusion demande un effort et fait prendre des risques, lorsqu'on fusionne ses branches fréquemment, les fusions sont plus légères et moins périlleuses.

Diapo 27: Critères de comparaison possibles

- Facilité d'utilisation
- Sites d'hébergement de projets
- Plateformes supportées
- Outils externes (e.g. intégration dans des IDE, GUIs)
- Plugins / Extensibilité
- Rapidité
- Projets de référence
- Gestion des branches et des fusions

Diapo 28: Subversion

- C
- Centralisé
- Familiarité et simplicité du modèle
- Maturité et robustesse des outils
- Grand nombre d'outils annexe et d'interfaces graphiques
- Pas de branches autrement que par convention de nommage
- Sites : Google Code, Kenai, SourceForge

Diapo 29: Subversion aujourd'hui

- Subversion 1.5 / 1.6
  En plus de nombreux bugs corrigés, un certain nombre de nouvelles fonctionnalités, dont certaines très attendues :
  - La gestion des copies et déplacements de fichiers ont été totalement revues et améliorées (déplacement direct du fichier dans l'espace de travail)
  - Optimisation complète de l'ensemble donnant une exécution beaucoup plus rapide
  - Apparition du '*sparse checkout*' : on peut ne récupérer qu'une partie de projet, et non nécessairement l'intégralité
  - Implémentation du '*merge tracking*' : la fusion de 2 branches devient plus facile et ne demande plus de gestion manuelle des numéros de révision ...
  - Gestion de liste de modifications ("*changelist*") à "commiter"

Diapo 30: L'avenir des VCS centralisés

- Subversion 2.x
  Il est annoncé par l'équipe de développement plusieurs évolutions s'inspirant de certaines caractéristiques des DVCS :
  - '*offline commits*' : pouvoir gérer ses commits dans des branches locales

    la gestion de branches locales synchronisées ultérieurement avec le dépôt central
  - la centralisation de la gestion des '*metadata*' en un seul endroit de la copie locale
  - tout en gardant son modèle centralisé et simple

Diapo 31 : Les VCS décentralisés

- Les points forts :
  - Pas de serveur à gérer
  - Travail déconnecté - accès aux dépôts sans réseau
  - Rapidité
  - Sauvegardes distribuées
  - Branches privées
  - Modèle de développement autorisé très souple
  - Encourage la programmation dite "agile"

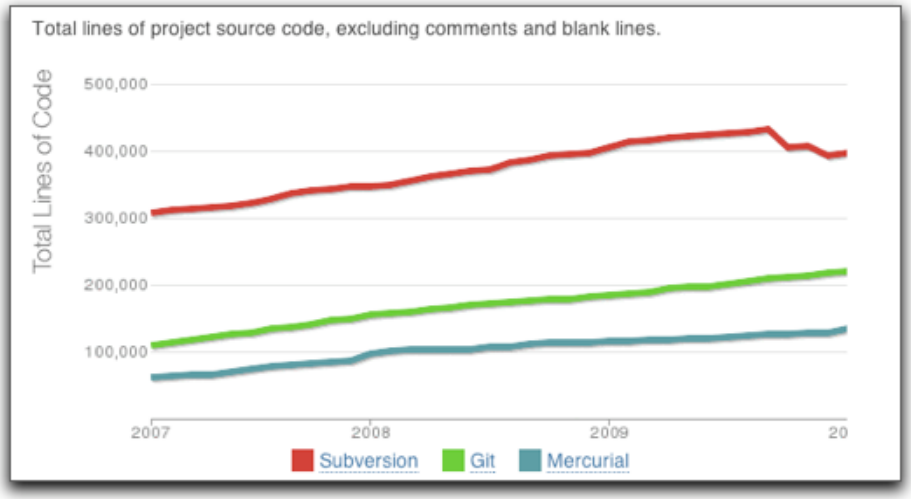
Diapo 32: Git

- C & Perl & Shell
- Origine : Linux Kernel Community
- D'abord la rapidité, ensuite la facilité d'utilisation
- Pas de version Windows native
- Puissant, mais nécessite un apprentissage
- Multiples branches par répertoire
- Sites : Kenai, github, gitorious, SourceForge
- Références : Linux Kernel, Ruby on Rails, jQuery, Android (Google), VLC, …

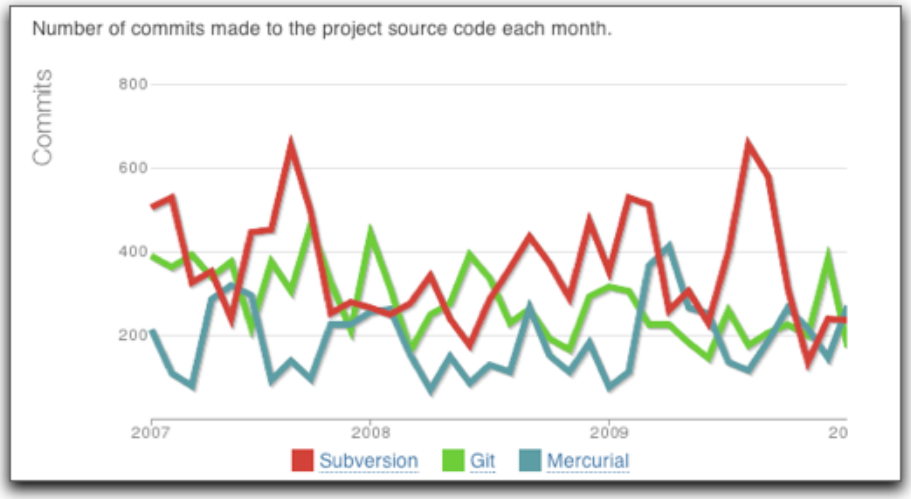
Diapo 33: Mercurial

- Python
- Facile à utiliser - très léger - très rapide
- Convient quelle que soit la taille du projet
- Plugins - API
- Sites : Bitbucket, Google Code, Kenai
- Multiples branches par répertoire (branches nommées)
- Références : NetBeans, OpenJDK, OpenOffice, OpenSolaris, Python, Mozilla, …

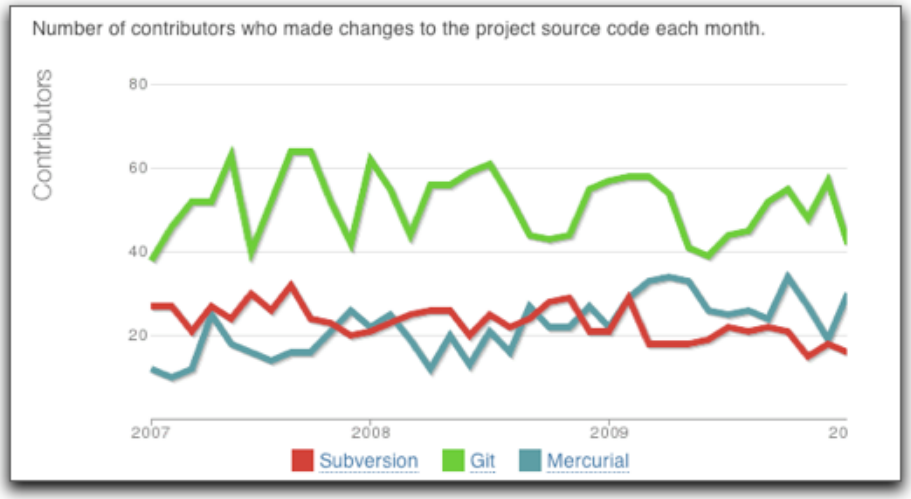
Diapo 34: Activités respectives - Nombre de loc/mois



Diapo 35: Activités respectives - Nombre de commits/mois



Diapo 36: Activités respectives - Nombre de contributeurs/mois



Diapo 37: Conclusion ?

Il n'y a pas de réponses absolues. Il faut considérer :

- avec quels types de données va-t-on travailler ?
  Si on manipule beaucoup de données binaires, un DVCS peut ne pas convenir
- de quelle manière veulent interagir les membres du groupe ?
  Si agilité, liberté et travail distant sont importants, un CVCS peut ralentir le groupe.

Et éventuellement :

- la gestion d'un firewall ?
- SVN permet de contrôler l'accès au niveau du fichier, contrairement à Hg ou Git.
- Hg et SVN ont des interfaces assez proches et simples - Git est potentiellement beaucoup plus compliqué
- tous les 3 sont facilement scriptables et intégrables dans d'autres outils de gestion de projets

Diapo 38: Conclusion ?

Chaque outil de gestion de version a sa propre approche et sa vision du travail en projet collaboratif.

Et en retours, il influence la façon de travailler de chacun au sein du groupe.

Un outil donné ne conviendra pas nécessairement à une équipe donnée : chaque outil vient avec ses avantages et ses contraintes qu'il faut essayer d'évaluer.

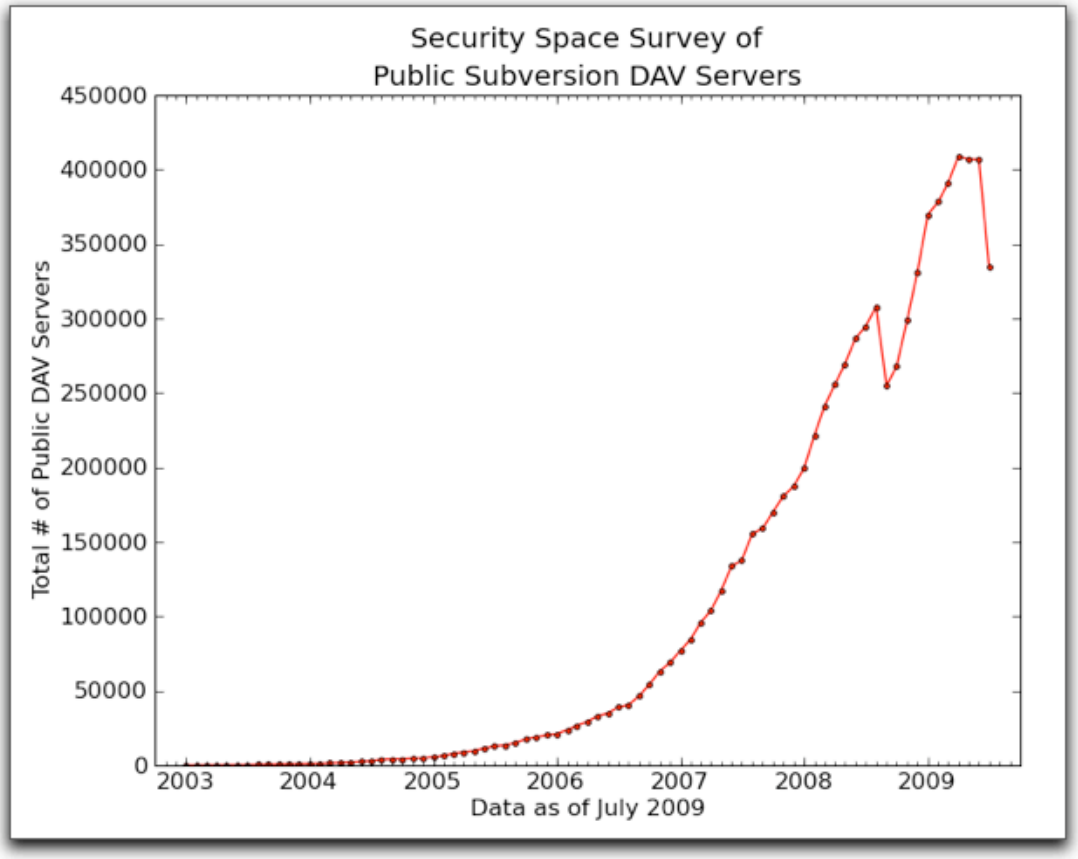
Merci !

Diapo 39: Références

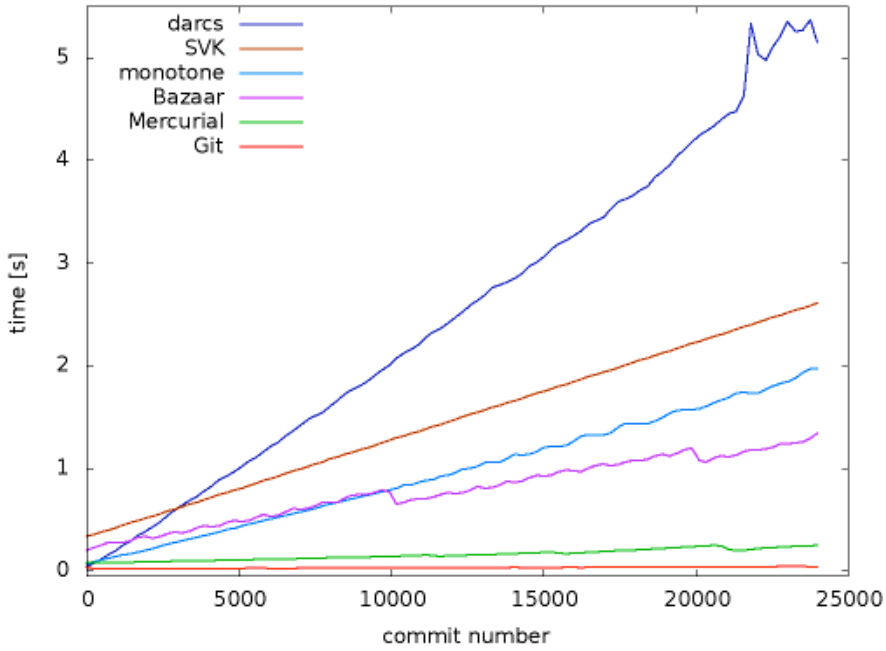
- Subversion
  Ben Collins-Sussman - "Version Control with Subversion"
  http://svnbook.red-bean.com/nightly/en/index.html
- Mercurial
  Bryan O'Sullivan - "Mercurial: The Definitive Guide"
  http://hgbook.red-bean.com/
- Git
  "Git Community Book"
  http://book.git-scm.com/
- Vitalité de Subversion :

  https://svn.apache.org/repos/asf/subversion/branches/python-3-compatibility/www/svn-dav-securityspace-survey.html
- Comparaison des activités :
  http://www.ohloh.net/p/compare
- Graphiques performance :
  https://ldn.linuxfoundation.org/article/dvcs-round-one-system-rule-them-all-part-3

Diapo 40: Subversion n'est pas encore mort !

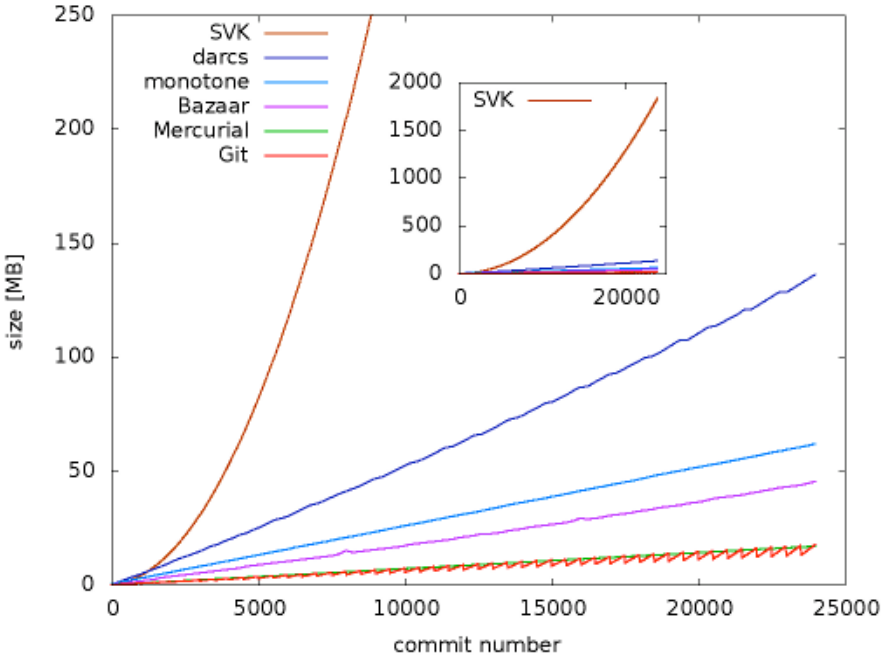


Diapo 41: Performances respectives de quelques DVCS



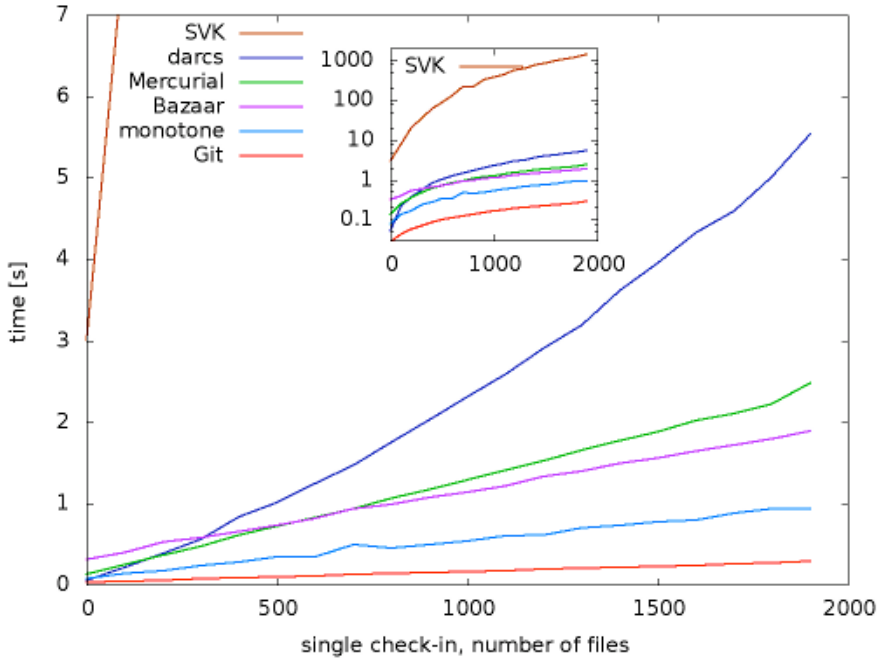
Évolution du temps de réponse d'un commit avec le temps (nbre de ci)
(4000 fichiers de 4050 bytes chacun)

Diapo 42: Performances respectives de quelques DVCS



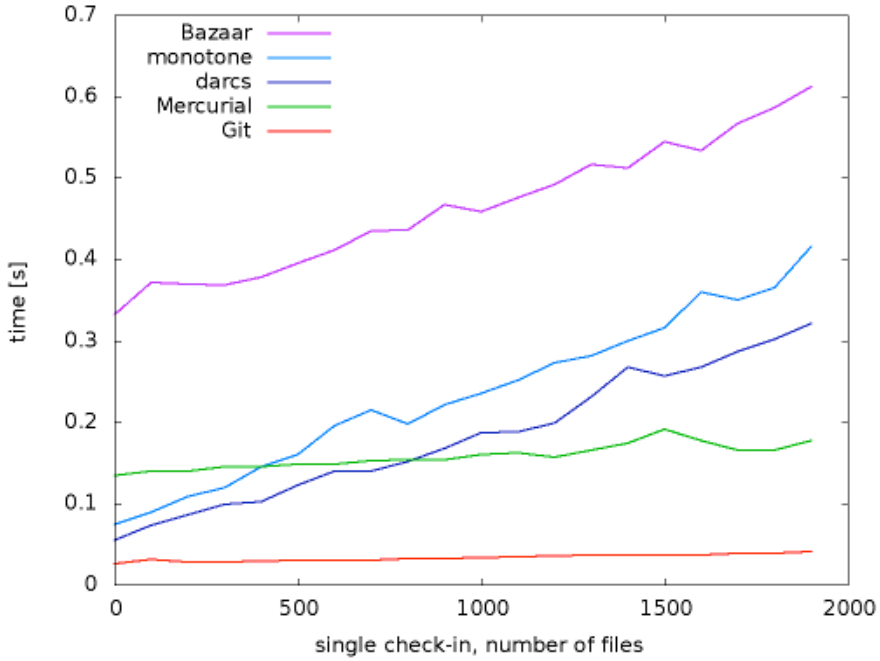
Évolution de la taille du dépôt avec le temps (nbre de ci)
(4000 fichiers de 4050 bytes chacun)

Diapo 43: Performances respectives de quelques DVCS



Évolution du temps de réponse en fonction du nombre de fichiers "commités"

Diapo 44: Performances respectives de quelques DVCS



Évolution du temps de réponse du "commit" d'un fichier en fonction du nombre de fichiers "commités" précédemment

Diapo 45: Renommage

Il n'est pas étonnant que Mercurial et Git ait une approche similaire face à la fusion des modifications, alors que Subversion s'y prend tout à fait différemment.

Comme les opérations de fusion sont réalisées très fréquemment avec Mercurial et Git, ces derniers sont particulièrement bien équipés pour ça.
Un cas d'utilisation délicat à gérer lors des fusions est le cas où des dossiers ou des fichiers ont été renommés.

- Aussi bien Hg que Git le font correctement.
- La machinerie de SVN pour fusionner est compliquée et fragile (aux dires même de ses auteurs - Ben Collin-Sussman et Daniel Berlin).
  Par exemple, les fichiers renommés disparaissaient lors des fusions;
  ce bug a été partiellement corrigé, les fichiers sont bien renommés lors de la fusion, mais leur contenu n'est pas toujours correct.

Diapo 46: Renommage

Renommage dans les environnements multiplateformes.

Windows, Mac OS X et les systèmes Unix n'ont pas les mêmes conventions de gestion de la casse des noms de fichier (FOO.TXT versus foo.txt).

Mercurial est capable de détecter et de gérer correctement le passage d'un environnement ignorant la casse à un environnement sensible à la casse.