Voyage dans le passé d'un code source avec un VCS

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 13 avril 2007. Dernière mise à jour le 7 mai 2007

https://www.bortzmeyer.org/voyage-temporel-code-avec-le-vcs.html

---

Les VCS ("*Version Control System*" ou Systèmes de Gestion de Versions) sont indispensables au développeur de logiciel pour garder trace des étapes successives du développement. Mais ils ne permettent pas facilement de retrouver une version particulière par son contenu.

Tous les VCS permettent de facilement extraire une **version** particulière d'un fichier si on connait des métadonnées comme le numéro de version ou comme la date. Par exemple, avec Subversion, voici comment récupérer la version **104** de `icp.h` :

```
% svn update -r104 icp.h
UU   icp.h
Updated to revision 104.
```

Et, avec CVS, voici comment récupérer la version du 1er octobre 2002 :

```
% cvs update -D '2002-10-01' nameserver.rb
P nameserver.rb
```

Le problème survient lorsqu'on veut remonter dans le temps en utilisant le **contenu** du fichier et pas ses métadonnées. Exemples typiques : « Qui a introduit la fonction `foobar()` et pourquoi ? » ou bien « Qui a supprimé le test `CHECK_THING` et pourquoi ? » Il n'y a pas cette fois d'option toute faite et chaque VCS va nécessiter une méthode particulière.

Il y a quatre fonctions à implémenter :
— 1) À quelle date est apparue telle chaîne de caractères dans tel fichier ?
— 2) À quelle date a disparu telle chaîne de caractères dans tel fichier ?
— 3) À quelle date est apparue telle chaîne de caractères (on ne connait pas le fichier) ?
— 4) À quelle date a disparue telle chaine de caractères (on ne connait pas le fichier) ?
Avec les conditions suivantes :

— On n'a pas accès au dépôt (pour les VCS centralisés comme Subversion) qui, de toute façon, est souvent en binaire (CVS ou Subversion-FSFS font exception), donc pas très lisible. C'est le cas d'un développeur d'un projet hébergé à Sourceforge, s'il n'est pas administrateur du projet.
— On suppose qu'il n'y a qu'une seule apparition ou disparition (pas de chaîne de carctères qui serait ajoutée puis supprimée puis réajoutée)

Pour Subversion, le plus simple est d'utiliser son API très complète, ici depuis un programme Python qui récupère successivement les différentes versions d'un fichier (en remontant vers le passé) jusqu'à trouver le motif recherché :

```
% grephistory.py -a getaddrinfo echoping.c
"getaddrinfo" was added (as "getaddrinfo") in echoping.c at revision 152

% svn log -r 152 echoping.c
------------------------------------------------------------------------
r152 | bortz | 2002-09-25 15:00:29 +0200 (Wed, 25 Sep 2002) | 2 lines

First IPv6 version

% grephistory.py -l gethostbyname echoping.c
"gethostbyname" was deleted (as "gethostbyname") from echoping.c at revision 152
```

Le programme en question (en ligne sur `https://www.bortzmeyer.org/files/grephistory.py`) (qui ne met en œuvre que les deux premières fonctions listées plus haut, celles où on connait le nom du fichier) utilise l'interface Subversion de Python <`http://pysvn.tigris.org/`>.

Vu l'algorithme utilisé, le programme est parfois très lent. Kim Minh Kaplan me fait remarquer qu'il serait préférable d'utiliser une recherche dichotomique. Si la révision actuelle est la 1000, l'algorithme actuel teste successivement les versions 999, 998, etc. Avec une recherche dichotomique, pour un ajout, il regarderait si la chaîne de caractères était déjà là à la révision 500 (1000 / 2), si oui, si elle était déjà présente à la révision 750, si non, à la 250, etc. La recherche serait donc en O(log(n)) au lieu d'être en O(n).

Mais la recherche dichotomique n'est pas stable. Si le motif cherché a été ajouté à la révision 400, retiré à la 600 et re-ajouté à la 800, l'algorithme bête trouvera toujours la 800. La recherche dichotomique peut donner 400 ou 800 selon la façon dont se fait le partitionnement. Donc, si la deuxième condition donnée plus haut (pas d'ajouts et de retraits multiples) est remplie, la recherche dichotomique est efficace. Sinon, elle est dangereuse.

Pour CVS, le même algorithme est possible, mais sa mise en œuvre est plus compliquée car il n'existe pas d'API. Il faut appeler les commandes CVS comme `cvs log` et `cvs export` successivement et analyser le résultat. Certains cas sont plus simples. Par exemple, pour trouver l'apparition d'une chaîne de caractères (fonction 1 dans notre liste) `cvs annotate $FICHIER | grep -A3 -B3 $CHAINE` est presque suffisant.

Une autre approche, si on a accès au dépôt CVS, est de regarder directement le fichier `${FICHIER},v` (un simple fichier texte) qui contient tout l'historique.

Pour darcs, le cas est un peu différent. Comme darcs est un VCS décentralisé, on a forcément tout le dépôt sur sa machine, avec tout l'historique. Donc, `zgrep -- $CHAINE _darcs/patches/*` peut être un bon point de départ.

Kim Minh Kaplan suggère une meilleure solution, la commande `trackdown` (renommée `test` dans les dernières versions <`http://darcs.net/Using/Test`>) par exemple pour chercher l'apparition de la chaîne "atom" dans le fichier `Site.tmpl` (avec l'aide de David Roundy car cette commande n'est pas d'un usage facile) :

```
% darcs trackdown -v '! grep -i atom Site.tmpl'
...
Trying without the patch:
Fri Dec 16 10:57:11 CET 2005  bortzmeyer@batilda.nic.fr
  * Auto-discovery of the Atom feed
Success!
```

Un autre VCS décentralisé, Mercurial, fournit une commande toute faite pour chercher dans l'historique, nommée logiquement `grep` (merci à Mathieu Arnold).

```
% hg grep --all -i sarkozy presidents.txt
presidents.txt:18:+:Nicolas Sarkozy
```

Le 18 est le numéro de la révision locale. `hg log -r 18 presidents.txt` nous en dira plus sur cette révision.

Pour les autres VCS, les contributions des lecteurs de ce blog sont les bienvenues.