Les nouveaux Systèmes de Gestion de Version

Stéphane Bortzmeyer
AFNIC
bortzmeyer@nic.fr

Résumé

Pendant très longtemps, l'unique logiciel libre et sérieux de gestion de versions était CVS. Il reste aujourd'hui la référence. Mais maintenant que ce vénérable logiciel atteint sérieusement ses limites et que la créativité bouillonne dans ce domaine, après dix ans de monopole absolu de CVS, il est temps de présenter la concurrence.

Pour succéder à CVS, il existe deux sortes de logiciels, centralisés ou décentralisés. La première catégorie est essentiellement représentée par Subversion, la seconde comprend de nombreux logiciels, dont la plupart ne sont pas encore très stables, mais elle représente probablement l'avenir de la gestion de versions.

Dans ces systèmes de gestion de versions décentralisés (comme darcs, mercurial ou cogito), il n'y a plus de dépôt de référence. Plusieurs dépôts coexistent et on peut lire et écrire dans celui de son choix.

Ils permettent ainsi un développement lui-même très décentralisé et collent donc au style de développement de la majorité des logiciels libres.

Mots clefs

patch, *changeset*, dépôt, *repository*, CVS, Subversion, darcs, git, cogito, Arch, tla, mercurial, monotone.

1 Introduction

Grâce entre autres à l'hébergeur SourceForge [1], l'utilisation de systèmes de gestion de versions (VCS, *Version Control System*) en réseau s'est largement répandue et ces systèmes sont désormais un des principaux outils de travail collaboratif. Fini, le travail solitaire du programmeur génial, dont le monde ne voyait rien avant la *release*.

Plus faciles d'utilisation et plus répandus, ces VCS sont utilisés par les développeurs bien sûr, mais aussi par les webmestres pour gérer les fichiers qui composent le site Web, par les ingénieurs système Unix pour gérer l'historique de `/etc`, par les auteurs de documentation ou de support de cours pour gérer leurs documents, articles ou exposés, etc[1].

Pendant très longtemps, l'unique logiciel libre réellement utilisable pour la gestion de versions était CVS. Il reste encore aujourd'hui la référence. Mais maintenant que ce logiciel atteint ses limites, cet article va présenter les concurrents.

CVS souffre en effet de plusieurs limites : les *commits* ne sont pas atomiques, les répertoires et les méta-données (le fait qu'un fichier soit exécutable, par exemple) ne sont pas versionnés, et le travail sur du code tiers, développé en amont[2], est très pénible.

Pour succéder à CVS, il existe deux sortes de VCS, centralisés et décentralisés (on trouve aussi le terme « répartis », traduction de *distributed*). Tous ont en commun de ne plus travailler par fichier, comme CVS, mais par *changeset* ou *patch*, un ensemble de changement sur plusieurs fichiers.

Dans les systèmes centralisés, il y a un dépôt de référence et les utilisateurs lisent et écrivent uniquement dans ce dépôt. CVS est l'archétype de ces VCS. Mais désormais, sur ce créneau, il cède du terrain à Subversion. Ce dernier, baptisé « CVS++ », tente de faire « CVS sans les problèmes ». Par exemple, il permet enfin de renommer un fichier. Ou de « versionner » les méta-données.

Dans les systèmes décentralisés, le sujet chaud du moment, on essaie de mieux coller au mode de développement en réseau si fréquent dans le logiciel libre.

Dans ces VCS, il n'y a plus de dépôt de référence. Plusieurs dépôts coexistent et on peut lire et écrire dans celui de son choix.

Cela met fin au dépôt central sacré, avec ses « commiteurs » privilégiés qui regardent les autres de haut. Le fait d'accorder ou de refuser le droit de « commiter » entraine souvent des crises qui sont désormais inutiles.

La décentralisation permet plus facilement de suivre un logiciel amont, auquel on ajoute quelques *patches* qui ne seront pas forcément intégrés. Les VCS décentralisés dédramatisent ainsi le *fork*, la scission, qui a secoué tant de projets de logiciel libre, en le rendant moins radical et moins définitif.

Elle permet enfin un travail avec le VCS en mode déconnecté, ce qui est important compte tenu de l'explosion de l'utilisation des ordinateurs portables (le portable contient un vrai dépôt). Avec CVS, si l'édition des fichiers est possible en mode déconnecté, toute opération sur le dépôt, même une simple lecture, demande une connexion au réseau.

[1] C'est pourquoi je n'utilise pas le terme de « Système de gestion de sources » car ces logiciels peuvent servir à bien d'autres choses que la gestion de sources, documentations Docbook ou pages HTML, par exemple. Voir par exemple [2].

[2] Par exemple, si je développe un logiciel qui teste qu'une zone DNS est correctement configurée, je peux avoir besoin d'une bibliothèque de bas niveau d'accès au DNS. Je peux vouloir lui apporter des changements, surtout si elle n'est pas très activement maintenue. Gérer ces changements est un scénario qui motive l'intégration de cette bibliothèque dans le VCS, alors qu'elle continue à évoluer en dehors, créant ainsi de futures difficultés.

2 Rappel sur la gestion de versions

Un système de gestion de versions (VCS) est un ensemble logiciel qui va ajouter une nouvelle dimension au système de fichiers : le temps. Un système de fichiers traditionnel est organisé de manière hiérarchique, avec ses dossiers ou répertoires qui contiennent d'autres dossiers ou répertoires et finalement des fichiers. Le système de fichiers garde en outre trace des méta-données, comme le nom du créateur du fichier, la date de création, ...

Le VCS ajoute le temps : il garde trace de toutes les versions successives d'un fichier. Il peut afficher l'historique des modifications et récupérer une version quelconque d'un fichier. Cela permet, par exemple, de retrouver la dernière version stable, ou bien, si on a identifié la date d'apparition d'un problème, la modification effectuée à cette date là. Avec un VCS correct, il n'est plus nécessaire de « commenter » les bouts de code supprimés comme dans :

```
/* Removed on 2005-08-13: no longer
   necessary. SB
      if ((sigaction (SIGPIPE,
                      &mysigaction, NULL))
         < 0)
;
*/
```

Il suffit de les supprimer et de noter dans l'historique la raison de cette suppression. Si la suppression est regrettée, le VCS permettra de retrouver le code[3] : rien ne disparait jamais si on utilise un VCS.

Le VCS stocke également des méta-données comme le nom de l'auteur d'une modification. En effet, tous les VCS (comme les SGBD) permettent un accès partagé au système de fichiers versionné, et ce sont donc de véritables outils de travail en groupe.

Par exemple, supposons que les habituels Alice et Bob travaillent sur le même article, qu'ils écrivent en Docbook/XML. Alice écrit un premier jet qu'elle enregistre dans le VCS. Bob le récupère et fait des modifications pendant l'absence d'Alice : au retour de celle-ci, elle pourra facilement voir ce que Bob a modifié[4], quand et pourquoi (si Bob a pris soin de fournir un message explicite).

Avec CVS, leur travail apparaitra comme sur l'exemple 1.

3 CVS, succès et limites

CVS, *Concurrent Version System*, est de loin le plus connu des VCS. Utilisé partout, pour du logiciel libre ou non-libre, il est tellement présent qu'il est, pour beaucoup de gens, la référence des VCS, voire le seul outil de VCS connu.

CVS est largement documenté, par des livres (voir [3] ou [4]) ou en ligne ([5] ou [6]).

CVS a mérité cette place en fournissant un outil de travail en groupe souple et riche à beaucoup de développeurs. CVS a popularisé l'idée de non-réservation d'un fichier : tout développeur muni des droits appropriés peut éditer un fichier. Si deux développeurs essaient d'enregistrer le même fichier, CVS tente de fusionner les deux modifications. S'il échoue, ce qui arrive par exemple si les deux modifications sont situées sur la même ligne du fichier, il affichera un conflit, qu'il laissera les développeurs résoudre[5].

Avant CVS, il était courant d'avoir des VCS qui obligeaient à effectuer une réservation préalable d'un fichier qu'on voulait éditer. Par oubli ou par volonté de garder le contrôle, beaucoup d'auteurs oubliaient de terminer cette réservation et le fichier devenait non modifiable.

Mais avec le temps, CVS n'a pas évolué et ses limites semblent maintenant de moins en moins acceptables :

1. Les *commits*, ou enregistrements des modifications, ne sont pas atomiques. En cas de disque plein, par exemple, on peut ne commiter qu'une partie des fichiers. La raison fondamentale est que CVS gère des fichiers, pas des *patches*. Nous verrons que tous les outils ultérieurs considèrent que l'objet élémentaire est le *patch*, pas le fichier.
2. CVS ne connait pas le renommage d'un fichier : si on change un fichier de nom, on perd tout l'historique de ce fichier.
3. Le travail sur du code tiers, ou bien les expérimentations sur le code sont pénibles car ce qui permet ces travaux en parallèle, les branches, sont lentes et difficiles d'usage.
4. Les répertoires ne sont pas versionnés (un répertoire est juste un container[6]),
5. les méta-données ne sont pas versionnées : on ne peut pas attacher de propriétés (comme les permissions) à un fichier, par exemple.
6. Il n'y a guère de travail possible lorsqu'on est déconnecté du réseau (cas d'un portable). Même `cvs diff` demande un accès au réseau.

[3] Notons tout de même qu'avec la plupart des VCS, surtout les centralisés, retrouver un bout de texte qui a été supprimé est plus difficile que de trouver la date et les conditions d'apparition d'un texte qui est présent dans la version actuelle. Il faut en général itérer sur toutes les versions d'un fichier en remontant dans le temps.

[4] La commande CVS peu connue `cvs annotate` est une aide considérable sur ce point.

[5] Tous les VCS peuvent utiliser des outils externes pour aider à la résolution de ces conflits, comme l'excellent kdiff3, `http://kdiff3.sourceforge.net/`.

[6] Tous les nouveaux VCS n'ont pas traité cette limite : par exemple, Mercurial (section 6.5) a repris ce modèle.

[7] Ne craignant par le syndrôme de la réinvention de la roue, un projet OpenCVS [7] a récemment vu le jour, pour assurer la maintenance / réécriture de CVS.

```
# Alice a fini sa première version
alice>  cvs add article.db
alice>  cvs commit -m "Premiere version" article.db

# Bob récupère les changements
bob>  cvs update
# Il effectue des modifications puis :
bob> cvs commit -m "Meilleure explication des causes de panne" article.db

# Alice récupère les changements
alice>  cvs update
# Qu'a fait Bob, exactement ?
alice> cvs diff -r1.1 -r 1.2 article.db
```

Exemple 1 – *Alice et Bob travaillent ensemble grâce à CVS*

7. Enfin, le code, peu clair et pas documenté, n'est plus du tout maintenu[7].

4 Fichiers, `patches` et autres

4.1 Quelle unité de travail ?

L'unité de travail de CVS est le fichier[8]. Si un changement affecte deux fichiers (changement d'une API où on change le `.h` et le `.c`), CVS ne le sait pas et permet, par exemple, un *commit* incomplet.

Le numéro de version par fichier (comme 1.3 ou 2.8) reflète cet état de fait.

Tous les VCS modernes ont pour unité de travail le changement ou *changeset* (aussi nommé *patch*).

Il comprend des changements faits dans plusieurs fichiers et il est atomique.

Pour certains VCS, notamment Subversion, seule l'insertion du *changeset* est atomique : le code reste bâti sur la notion de *snapshots*, de « photographies » du dépôt à un moment donné (voir [8] pour une bonne discussion de ce sujet).

Dans la plupart des VCS modernes, le *patch* est conservé dans le dépôt, sous des formats divers (et malheureusement non standard[9]). L'exemple 2 montre par exemple un *patch* de darcs.

4.2 Les identificateurs de *patch*

Tout VCS doit fournir un moyen de désigner un *patch* particulier. Par exemple, tous les VCS ont une fonction (généralement nommée `diff`) pour comparer deux états du dépôt à des temps différents.

Subversion identifie ses *patches* par un numéro de révision, global au dépôt. Darcs utilise une chaîne de caractères choisie par l'utilisateur (qui doit donc éviter des identificateurs non significatifs, comme « bug fix »).

Les autres VCS ont des identificateurs de *patch* calculés, par un résumé SHA-1 ou équivalent. Mercurial (section 6.5) permet de n'indiquer que le début de l'identificateur, et si ce début n'est pas ambigü, cela fonctionne, ce qui évite de taper de longs résumés SHA. Ce calcul garantit leur unicité globale, même en cas de fusion de dépôts.

Ces identificateurs servent aussi à la communication entre humains, comme on le voit dans l'exemple 6 (« Cette bogue a été introduite par le *patch* cbca0ba5a800bfc6796122, je pense. »)

5 Le successeur de CVS, Subversion

Les concepts étant les mêmes, les simples utilisateurs passent de CVS à Subversion en cinq minutes, d'autant plus que les commandes sont identiques. Pour l'administrateur, c'est plus difficile, car le changement est plus profond. Le format du dépôt est complètement différent (il en existe au moins deux, avec DB et avec des fichiers plats). La configuration d'un serveur réserve quelques pièges, on peut utiliser Apache 2 et WebDAV ou bien le plus simple svnserve.

Subversion ([9]) est aussi surnommé « CVS++ » et son cahier des charges résumé par un de ses développeurs avec *« CVS is an excellent, proven model for version control ; it just wasn't implemented well. »*.

Subversion est délibérément conçu pour être proche de CVS. S'il remplace `cvs` par `svn` dans la commande, l'**utilisateur** de CVS est opérationnel dans la grande majorité des cas.

Subversion est largement documenté, par exemple dans [10] et [11].

Subversion introduit quand même quelques changements pour l'utilisateur de CVS :

– `svn diff` et `svn status` marchent en déconnecté
– Les répertoires et méta-données sont versionnés
– `svn rename` permet le renommage et n'a pas d'équivalent CVS

[8] L'unité de verrou est le répertoire, on a presque atomicité par rapport aux fichiers d'un répertoire donné, pour les accès concurrents, mais pas pour les pannes comme le disque plein. C'est quand un *commit* est multi-répertoire que le risque est le plus fort.

[9] Il est possible que git, section 6.7, devienne le format standard de stockage de beaucoup de VCS dans le futur. Voir la section 7.

```
[converted readme to docutils-style markup simons@cryp.to**20041004030453]  < > {
hunk ./README 10
- (1) "ADNS" provides access to the GNU ADNS library via the
-standard foreign function interface. Not all options are
-supported as of now, but you can do A, MX, NS, and PTR
-lookups. It has been tested with ADNS versions 1.0 and 1.1.
+ "ADNS"
+     provides access to the GNU ADNS library via
+     the standard foreign function interface. Not all
+     options are supported as of now, but you can do A, MX,
```

Exemple 2 – *Le format de `patch` utilisé par darcs. Ici, le `patch` consiste simplement en un changement dans le fichier README.*

```
------------------------------------------------------------
r56 | bortzmeyer | 2004-07-22 11:47:47 +0200 | 1 line

Binary databases dumped to an intermediate directory
------------------------------------------------------------
r39 | bortzmeyer | 2004-07-06 07:15:15 +0200 | 1 line

purge of old backups done
```

Exemple 3 – *Résultat d'un `svn log Backup`*

– Les *commits* atomiques (d'où le numéro de version par changement)

On voit ce dernier effet dans l'exemple 3, on notera le saut de la révision de 39 à 56. Cela ne résulte pas d'une erreur de copier-coller dans l'article mais du fait que le numéro de révision, dans Subversion, est global pour tout le dépôt. Ici, le fichier `Backup` n'a pas été changé entre r39 et r56.

Subversion introduit aussi des changements pour l'administrateur. Il permet à celui-ci de choisir le dorsal de stockage des données. Par défaut, Subversion utilisait Berkeley DB, pour avoir une « vraie » base de données.

Mais il peut aussi utiliser un dorsal « fichier », nommé FSFS, qui sera le choix par défaut dans la prochaine version.

Dans les deux cas, plus question d'éditer le dépôt à la main. Si on utilise Berkeley DB, il faut également penser à faire des sauvegardes en mode texte[10]

Pour servir les dépôts Subversion sur le réseau (un des points faibles de CVS), il y a deux approches : Apache 2 ou bien un simple serveur à lui.

Contrairement à une idée très répandue, Subversion n'impose pas Apache. C'est simplement l'un des mécanismes permettant l'accès distant au dépôt[11] Si on utilise ce mécanisme, il faut un serveur Apache 2. En effet, Subversion dépend de l'API d'Apache 2 pour son module mettant en oeuvre le protocole DeltaV, lui-même bâti au dessus du protocole WebDAV (RFC 2518[12] et RFC 3253).

Avec Apache, un seul processus et un seul utilisateur accèdent au dépôt. Les clients peuvent alors accéder le dépôt en donnant son URL (par exemple `svn co http://svn.example.org/foobar`).

En dehors d'Apache, une autre solution pour servir un dépôt Subversion sur le réseau est d'utiliser le protocole spécifique de Subversion (qui peut être tunnelé dans SSH, pour la sécurité), avec le serveur svnserve.

Dans le cas où plusieurs utilisateurs accèdent à la base, par des accès locaux ou bien par svnserve, attention aux corruptions (des permissions incorrectes du dépôt peuvent entrainer une corruption - réparable - de la base).

Pour l'administrateur du projet (et aussi pour tous les participants), une des grandes forces de Subversion est la disponibilité d'une API de programmation, accessible depuis beaucoup de langages. Là où, avec CVS, il fallait appeler des commandes Unix et analyser le résultat, Subversion fournit un moyen beaucoup plus propre de développer de petits utilitaires. Par exemple, le programme Python de l'exemple 4 permet de ne garder que les *commits* effectués par un utilisateur donné[13].

6 Un autre paradigme, les VCS décentralisés

Nous allons maintenant changer de paradigme. Le passage des VCS centralisés aux VCS décentralisés va nécessiter une même ouverture d'esprit que celle qui avait permis le saut des VCS à réservation et verrouillage vers CVS et son modèle « On édite d'abord et on fusionne après ».

Les systèmes centralisés comme CVS et Subversion se ca-

[10] Comme avec les autres logiciels qui utilisent un format binaire pour leurs données, par exemple OpenLDAP ou PostgreSQL.
[11] Subversion nécessite l'APR (*Apache Portable Library*) pour être compilé mais cela ne signifie pas qu'un serveur Apache soit obligatoire à l'usage.
[12] Pour voir le RFC de numéro NNN, `http://www.ietf.org/rfc/rfcNNN.txt`, par exemple `http://www.ietf.org/rfc/rfc2518.txt`
[13] Avec CVS, il aurait fallu appeler `cvs log` et analyser la sortie de cette commande.

```
import pysvn
...
client = pysvn.Client()
log = client.log(path)
for message in log:
    if user is None or user == message['author']:
        log_message = first_line(string.strip(message['message']))
        if not log_message:
            log_message = "EMPTY LOG MESSAGE"
        if user is not None:
            author = ""
        else:
            author = "by %s" % message['author']
        print "%s (%i): \"%s\" %s" % (date,
                                      message['revision'].number,
                                      log_message.encode(locale, "replace"),
                                      author)
```

Exemple 4 – *Exemple de programme Python utilisant l'API de Subversion*

ractérisent par un dépôt **privilégié**.

Donc :

– Certaines opérations sont impossibles en mode déconnecté (pas de `svn log`) ou si on n'est pas *committeur.*
– Les développements parallèles (stable vs. instable, par exemple) nécessitent des acrobaties (branches[14] de CVS).
– Il faut désigner des ***committeurs***, des privilégiés, ce qui peut entrainer des disputes.
– Les scissions (*fork*) sont binaires et donc douloureuses. Corollaire : il est difficile de suivre un projet amont en tentant des développements locaux (*vendor branch* de CVS).

Mais les VCS centralisés ont aussi leurs forces :

– On a un dépôt de référence, connu ;
– On simplifie l'implémentation et l'usage ;
– On peut appliquer des politiques globales (comme des tests de non-régression ou un envoi de courrier pour chaque *commit*).

D'innombrables articles et entrées de *blog* ont déjà été écrites pour comparer les deux approches (voir [12], [13], [14], ...).

Avec les VCS décentralisés, chaque développeur peut avoir son dépôt dans lequel il peut committer.

Il y a donc parfois deux *commits*, un dans son dépôt et un dans le dépôt des livraisons.

Il est donc nécessaire de temps en temps de **synchroniser** (au moins partiellement) deux dépôts. En général, l'opération de copie initiale d'un dépôt se nomme `clone` (darcs l'appelle `get`) et les synchronisations se nomment `push` et `pull` selon la direction où elles se font.

On notera que le VCS n'est qu'un outil, il ne définit pas une politique (comme tout bon outil de travail en groupe). Le choix d'avoir ou non un dépôt de référence, ainsi que le rythme de synchronisation, sont des questions d'organisation du projet, un VCS décentralisé n'impose rien à ce sujet.

Par exemple, on peut mettre en œuvre une politique centralisée, avec un dépôt de référence, au dessus d'un VCS décentralisé. L'inverse n'est pas vrai : si on choisit un outil centralisé, on ne pourra pas adopter ensuite une politique décentralisée.

6.1 Exemple avec darcs

Comme ces concepts sont assez nouveaux, nous allons les illustrer avec darcs (section 6.2), le plus simple à utiliser :

```
% ls
Makefile  aux.c  main.c
% darcs init
% darcs add *
Skipping boring file _darcs
% darcs record --all
What is the patch name? First import
Finished recording patch 'First import'
```

Sur cet exemple, un développeur C a commité (`darcs record`) trois fichiers. Il n'y a pas encore de différence avec un système centralisé, nous n'avons pas exploité toutes les capacités de darcs.

Voyons maintenant darcs avec plusieurs dépôts.

```
% ls
% darcs init
% darcs pull bortzmeyer@project.example.org/projet
...
Finished pulling.
% ls
Makefile  _darcs  aux.c  main.c
```

Mais ce n'est pas l'équivalent d'un `cvs co` ! La copie est un vrai dépôt, avec les mêmes droits que l'« original ».

On peut maintenant travailler en local.

```
% vi main.c
% darcs record --all
What is the patch name? Exit properly in main.c
Finished recording patch 'Exit properly in main.c'
```

[14] La fusion entre branches est généralement considéré comme l'un des pires cauchemars avec CVS.

On peut donc « committer » en local. D'autres développeurs connectés à l'Internet peuvent utiliser mon dépôt, c'est un dépôt de première classe.

On peut ensuite resynchroniser les dépôts

```
% darcs push --all
Pushing to bortzmeyer@project.example.org/projet...
Finished applying...
```

Mais on n'est pas obligé de resynchroniser : le principe du VCS décentralisé est de ne **pas** imposer une politique. Quand synchroniser et quoi envoyer sont des questions d'organisation, dont la réponse n'est pas imposée par le VCS.

Même si on synchronise, on peut choisir les patches à envoyer, de façon à ne pas tout fusionner. Voyons un exemple : supposons qu'on utilise RequestTracker[15] pour la gestion de tâches et de bogues. Le ticket #1354 est une bogue à réparer alors que le développement continue.

```
% darcs changes
Wed Jan 12 23:16:14 CET 2005  saroumane@isengard
  * New function foo()
Wed Jan 12 17:25:36 CET 2005  galadriel@lothlorien
  * #1354: fix main.c
Wed Jan 10 13:24:56 CET 2005  sauron@mordor
  * New brilliant idea: frobnicate before foobaring
```

On peut n'envoyer que les changements réellement liés à la bogue #1354.

```
% darcs push --patches '#1354' --all
Pushing to bortzmeyer@project.example.org/projet...
Finished applying...
```

Dans le dépôt `bortzmeyer@project.example.org/projet`, on ne verra que ce changement et pas les autres.

Ce qui apparaît dans la sortie de `darcs changes` est le nom du changement, pas juste un commentaire. Ce nom peut être utilisé dans les commandes de darcs. Avec darcs, une politique de nommage des changements est donc très recommandée (section 4.2).

6.2 darcs et la théorie des *patches*

darcs est à la fois très simple (comme CVS, on peut s'y mettre en quelques minutes) et très matheux avec sa « théorie des *patches* » qui tente de donner un cadre théorique solide à la gestion des conflits entre dépôts décentralisés qu'on essaie de synchroniser. L'algorithmique mise en jeu par les VCS décentralisés est en effet bien plus complexe.

C'est darcs qui pousse le plus loin la logique des VCS décentralisés : toute copie de travail est un dépôt de plein droit. L'équivalent des branches de CVS, par exemple, est juste un autre dépôt.

Le serveur a besoin de darcs, on ne peut pas mettre un dépôt darcs sur un simple serveur de fichiers.

darcs est écrit en Haskell (voir l'exemple 5). Il existe un compilateur Haskell libre répandu (ghc) mais qui échoue sur des architectures rares (Linux/mips, NetBSD/sparc).

Le principal inconvénient de darcs reste la lenteur lors des conflits multiples. Si deux dépôts ont une histoire très différente, la récupération de tout ou partie des *patches* d'un dépôt va entrainer de nombreux conflits potentiels, dont l'examen peut facilement occuper darcs pendant des heures. Malheureusement, avec darcs, le pire cas n'est pas borné[16].

En outre, darcs doit avoir une copie en mémoire du dépot complet, ce qui est ennuyeux dans certains cas comme le noyau Linux ou le `/usr/src` de FreeBSD.

darcs a de nombreuses possibilités qui ne sont qu'effleurées ici. Par exemple, il permet d'envoyer un *patch* facilement par courrier électronique.

La théorie des *patches*. Un *patch* est un ensemble de changements (*changeset*) sur un arbre de fichiers.

On peut le voir comme une **fonction** (ou un **opérateur**, si on a fait de la mécanique quantique) qui prend un arbre et rend un autre arbre.

$P(t) \rightarrow t'$

On peut appliquer plusieurs *patches* de suite (`darcs changes` affiche la liste) :

$P_1(P_2(P_3(t)))$

Tout arbre est l'application d'une série de *patches* à l'arbre vide.

Il existe l'inverse d'un *patch* : P^{-1}

Les *patches* peuvent parfois **commuter** :

$P_1 o P_2 = P_2 o P_1$

Important : la commutation dépend de la représentation des *patches* : si on ne garde pas de contexte, on ne peut pas souvent *commiter.*

De la notion de commutation, on passe à celle de dépendance : **Deux *patches* commutent si et seulement si ils sont indépendants**.

A quoi cela sert-il ? Par exemple, si j'ai l'arbre $A(B(C(t)))$ dans un dépôt et $C(t)$ dans un autre, je peux appliquer le *patch* A au second dépôt s'il est indépendant de B $\Leftrightarrow$ si A commute avec B.

darcs tente de fusionner les séries de *patches* en utilisant la commutation. S'il n'y a pas de commutation possible, on a un **conflit**.

De la même façon, darcs utilise la commutation pour voir s'il peut installer un seul *patch* dans une série.

[15] Request Tracker est l'outil libre de gestion de tâches le plus répandu. `http://bestpractical.com/rt/`

[16] Un gros travail d'algorithmique est actuellement en cours pour tenter de résoudre ce problème.

```
commute (p1, p2) -- Deal with common case quickly!
 | p1_modifies /= Nothing && p2_modifies /= Nothing &&
    p1_modifies /= p2_modifies = Just (p2, p1)
   where p1_modifies = is_filepatch_merger p1
         p2_modifies = is_filepatch_merger p2
commute (NamedP n1 d1 p1, NamedP n2 d2 p2) =
   if n2 'elem' d1 || n1 'elem' d2
   then Nothing
   else do (p2', p1') <- commute (p1,p2)
      return (NamedP n2 d2 p2', NamedP n1 d1 p1')
```

Exemple 5 – *Un exemple Haskell, tiré du source de darcs. La fonction* `commute`*, qui dit si deux* patches *commutent, est définie par* pattern matching *sur ses arguments, d'où les définitions multiples.*

6.3 svk

SVK ([15]) est un VCS décentralisé (écrit en Perl) qui réutilise une partie de Subversion (section 5) notamment ses dorsaux de stockage et son mécanisme d'accès aux dépôts distants.

Cette compatibilité partielle avec Subversion permet au développeur dont le dépôt officiel est sur Subversion de créer une copie locale du dépôt du projet (tout le dépôt, avec son historique, pas une simple copie de travail, qui ne stocke qu'une version) et travailler ensuite en local, avant de resynchroniser avec le dépôt « officiel » :

```
svk cp https://svn.example.org/ projet
```

Puis, à la fin, pour resynchroniser :

```
svk push
```

SVK est un logiciel stable et bien maintenu et sa compatibilité partielle avec Subversion aidera peut-être la migration de certains projets, qui hésitent devant le saut que représentent les autres VCS décentralisés.

6.4 Arch/tla

Arch et sa mise en oeuvre la plus connue, tla ([16]), est l'ancêtre des VCS décentralisés. tla est très complexe à utiliser, mais il a néanmoins déblayé le terrain et introduit le concept de décentralisation auprès de beaucoup d'utilisateurs, concurrençant sérieusement Subversion.

Il y a une subtile nuance entre GNU Arch, le protocole et tla, la mise en œuvre la plus connue. Désormais, il existe dans cette famille `larch`, la première implémentation en Bourne Shell, le tla historique, puis tla 2 (appelé `revc` qui n'est pas compatible avec le précedent), Bazaar (`http://bazaar.canonical.com/`) et Bazaar-ng (maintenant appelé Bazaar 2 et familièrement bzr `http://www.bazaar-ng.org/`) qui n'est plus compatible.

tla a longtemps été le VCS décentralisé le plus répandu. À bien des égards, c'est lui qui a lancé le concept dans le monde du logiciel libre, même s'il semble nettement sur le déclin aujourd'hui.

tla 1 est particulièrement difficile à utiliser, en partie en raison de sa curieuse « charte de nommage ».

tla divise le monde en **archives**, puis **catégories**, puis **branches**, puis **versions** et enfin **patches**, par exemple, `scott@netsplit.com--2005/dpkg--devel--1.13--patch-26`.

`scott@netsplit.com--2005` est l'archive (dépôt). Ensuite, les catégories, branches, versions et *patches*, séparés par --. Le but de cette charte de nommage est d'obtenir un nom unique pour une révision donnée dans une archive donnée et ça facilite la gestion des archives multiples en déplacant la complexité vers l'utilisateur.

Le serveur n'a pas besoin de tla : ce n'est qu'un serveur de fichiers, les accès peuvent se faire *via* HTTP ou WebDAV, SSH ou encore FTP.

Une archive n'est pas un répertoire de travail : si on veut une branche, il faut le demander explicitement. Autre point important, l'unité de base pour une branche est une catégorie et pas un répertoire de travail ou un dépot. L'implication principale est que créer une branche d'un projet complexe avec plusieurs catégories nécessitent l'utilisation des configurations, ce qui complique fortement l'utilisation de tla.

6.5 Mercurial

Mercurial ([17]), également connu sous le nom de `hg`, la commande à utiliser[17], est un des plus récents VCS décentralisés mais a rapidement suscité beaucoup d'intérêt, notamment en raison de ses grandes performances. Mercurial est écrit en Python.

Contrairement à darcs (section 6.2), il stocke plusieurs branches dans chaque dépôt et la fusion entre ces branches est souvent délicate.

Voici l'historique d'un dépôt Mercurial :

```
% hg log

changeset:   3:cbca0ba5a800
parent:      0:da413f7b0891
parent:      2:a9816f6ff2c1
user:        stephane@ludwigVI.sources.org
```

[17] Ce n'est pas une *private joke* d'informaticien mais une *private joke* de chimiste.

```
date:        Wed Sep  7 22:55:45 2005 +0100
summary:     Première version lettre Homelink
```

On note que les *changesets* ont deux identificateurs, un local au dépôt, le *revision number* (3, ici) et un global, le *changeset ID* (cbca0ba5a800, ici), moins convivial.

Le *changeset* vu ici a deux parents, qui sont deux branches différentes présentes dans le même dépôt.

6.6 monotone

Monotone ([18]) est un des « anciens » VCS décentralisés. Écrit en C++, il reste très utilisé. Il a été le premier à demander systématiquement la signature cryptographique de chaque *patch*. En effet, un VCS décentralisé est *a priori* très sensible aux chevaux de Troie puisqu'il va recevoir des *patches* d'origine très diverses. La confiance n'étant pas transitive (si je fais confiance aux *patches* du dépôt B et que le dépôt B faisait confiance aux *patches* du dépôt C, je vais me retrouver avec les *patches* de C, que je ne connais pas), la signature cryptographique, quoique pénible en pratique, peut apporter une utile traçabilité.

Monotone est, par contre, handicapé par son serveur spécifique, qui rend difficile la traversée des coupe-feux (les autres VCS utilisent en général HTTP et SSH).

6.7 git et cogito

Ces deux logiciels sont bien séparés mais, en pratique, ils sont souvent utilisés ensemble. Leur utilisation massive pour le développement du noyau Linux leur a valu beaucoup d'intérêt, alors qu'ils sont tous les deux encore assez primitifs (git est quasi-inutilisable seul).

git. git a été développé spécifiquement pour le noyau Linux, par Linus Torvalds, l'auteur du noyau. Pendant très longtemps, Linux n'avait pas de VCS accessible à tous (chaque développeur utilisait son propre VCS mais rien n'était partagé). Du fait du style de travail très décentralisé, les VCS centralisés comme CVS ne convenaient pas.

Pendant plusieurs années, Linux a été développé avec un logiciel commercial, dont la licence interdisait explicitement toute utilisation pour développer un système de VCS concurrent !

Ce logiciel a changé brusquement sa licence et Linux, déjà handicapé par ce système non-libre, a dû se trouver un VCS libre en urgence. D'où le développement de git (`http://www.kernel.org/pub/software/scm/git/docs/` et `http://www.kernel.org/pub/software/scm/git/` pour le télécharger).

Les VCS décentralisés existants ont en effet pour la plupart reculé devant la taille du noyau Linux, le nombre de *patches* accumulés et le nombre de développeurs.

git n'est pas un vrai VCS et ne prétend pas l'être. C'est plutôt un simple gestionnaire de changements, une sorte de système de fichiers très simple, mais avec une dimension supplémentaire, le temps.

C'est un outil de très bas niveau, et il est donc très difficile à utiliser seul. En général, on passe par des interfaces situées au dessus de git, comme Cogito (section 6.7).

git travaille avec des objets dont le nom est un résumé cryptographique (l'algorithme SHA1 est utilisé). Ainsi, son contenu décrit complètement un objet. Pour git, si un fichier change de contenu, ce n'est plus le même fichier. Si on crée un fichier dans un répertoire, ce n'est plus le même répertoire. Cette technique est à la base de la gestion du temps dans git. git ne se demande pas « Quel était l'état du fichier foobar.c au 30 décembre ? » mais « Quel est le contenu de l'objet 092bc2b04126100878530888e6b1b306-02dce213 ? ».

git gère quatre sortes d'objets :
– les *blobs* qui sont des données non structurées (typiquement le contenu d'un fichier)
– les arbres, qui sont structurés (ils ont un lien vers leurs fils) et servent à modéliser les répertoires
– les *commits* qui désignent un arbre, le (ou les) *commit* précédent et le message associé. Les *commits* sont donc chainés entre eux.
– les *tags*

Voyons l'examen d'un répertoire géré par git :

Par convention, on stocke le nom (le résumé SHA1) du dernier commit dans le fichier `.git/HEAD`.

```
% cat .git/HEAD
e5d52c78555e5a24ba1b9a530ed763240003648d
```

Armés de cette information, on peut savoir le contenu de ce *commit* :

```
% git-cat-file commit e5d52c78...03648d
tree c638425d8eb1a...b37f444ce7b1f9
parent e5d52c78555e...d763240003648d
author Stephane Bortzmeyer
    <stephane@f.sources.org> 1116431886 +0200
committer Stephane Bortzmeyer
    <stephane@f.sources.org> 1116431886 +0200

Frobnication foobar()
```

Le *commit* référence un arbre, l'état du répertoire au moment du commit. On peut regarder cet arbre :

```
% git-ls-tree c63842...7b1f9
100644  blob    092bc2b...e213        foobar.c
```

Le « nom » (la signature SHA1) du fichier permettra de le lire tel qu'il était au moment de ce *commit* :

```
% git-cat-file blob 092b...ce213
void main(...
```

Si on veut remonter le temps, on peut combiner des commandes, puisque les objets *commit* sont chaînés. Je l'avais

dit, git est de bas niveau. Tout ce qu'on fait avec un VCS est possible avec git mais nécessite souvent la combinaison de plusieurs commandes et la manipulation de « noms » SHA1. Par exemple, pour voir la différence de deux arbres (un *patch*), il faut donner leurs noms :

```
% git-diff-tree 70c...eacb c63...7f4 | \
  git-diff-tree-helper
```

Les choses deviennent plus intéressantes lorsqu'on sait que les dépôts git sont accessibles de l'extérieur, par HTTP[18]. Et surtout que git permet à un *commit* d'avoir plusieurs parents, ce qui permet de représenter la fusion de deux lignes de développement. git permet de fusionner deux *commits*, après avoir trouvé leur plus proche parent commun.

cogito. cogito ([19]) est un ensemble de scripts shell qui se situe au dessus de git et permet de manipuler plus facilement des archives git.

Ainsi, cg-clone vous permettra de récupérer le dernier code de Linus sur le noyau 2.6[19] Vous pourrez ensuite regarder, exemple 6, ce qui a changé.

6.8 codeville

Codeville ([20]) est un des plus anciens VCS décentralisés. Écrit en Python, comme Mercurial, section 6.5 (qui semble susciter nettement plus d'intérêt aujourd'hui), il ne semble plus guère utilisé, mais il sert de plateforme de test pour des algorithmes de fusion.

7 Transition et coexistence

Il est probable que la coexistence de nombreux VCS différents est amené à durer plusieurs années. La simplicité du monde où CVS était le seul outil ne va pas revenir de si tôt.

Il est possible que git (section 6.7) fournisse une base commune à plusieurs VCS, darcs ayant déjà par exemple la possibilité d'accéder aux archives git. Mais cela ne résoudra pas tous les problèmes de coexistence.

Pendant longtemps, il est donc probable qu'on verra des équipes travailler sur le même projet avec des VCS différents et devoir donc synchroniser régulièrement, dans les deux sens, avec un outil comme Tailor ([21]).

Il est possible qu'apparaisse un format commun des *patches*, qui permettrait de faciliter l'écriture d'outils comme Tailor. Un tel format ne serait pas évident à définir, chaque VCS ayant une vision légèrement différente de ce qu'est un *patch* (par exemple, Subversion ou Mercurial stockent le caractère exécutable du fichier, une méta-donnée, mais pas darcs : une conversion Subversion ⇒ darcs ⇒ Subversion ferait donc perdre cette information).

Pour l'instant, chaque outil a son propre format d'échange de *patches*, même CVS, avec le non-officiel CVSps (`http://www.cobite.com/cvsps/`).

8 Conclusion

Les VCS ont accompagné les auteurs et développeurs depuis de nombreuses années. Les plus anciens étaient très contraignants par leur modèle de verrouillage préalable. CVS a libéré les auteurs et les développeurs et a montré depuis des années que le modèle où on édite d'abord et on fusionne (éventuellement) après était viable, alors que beaucoup d'experts prédisaient que cela mènerait au désordre.

Aujourd'hui, les VCS décentralisés peuvent inquiéter car il s'agit d'un modèle nouveau et qui semble moins contrôlé. Mais le développement du noyau Linux, où ce modèle est utilisé depuis plusieurs années, montre que le développement décentralisé est possible.

Si les outils sont encore primitifs (par exemple, les mécanismes de publication sur le réseau sont sommaires), et trop nombreux (une sélection darwinienne va être nécessaire) le paradigme des VCS décentralisés est déjà bien installé et représente certainement l'avenir.

Références

[1] Sourceforge, *Sourceforge software development hosting system*. `http://www.sourceforge.net/`.

[2] Stéphane Bortzmeyer et Olivier Perret. Versionnage : garder facilement trace des versions successives dun document - exemples avec un outil de contrôle de versions (cvs). Dans CHABIN Marie-Anne, éditeur, *L'archivage (Revue Document numérique Volume 4 Nř 3-4)*, pages 252–264, 2000.

[3] Karl Fogel et Moshe Bar. *Open Source Development with CVS*. Numéro 1-932111-81-6 dans ISBN. Paraglyph Press.

[4] Andy Hunt Dave Thomas. *Pragmatic Version Control Using CVS*. Numéro 0974514004 dans ISBN. Pragmatic Programmers, 2003.

[5] *Concurrent Versions System*. `http://www.nongnu.org/cvs/`.

[6] Per Cederqvist, *Version Management with CVS ('official' manual)*. `http://ximbiot.com/cvs/manual/`.

[18] `git-http-pull 667bb59b2d5b0a2e7fca5970d6f757790a6edd74 http ://rsync.kernel.org/pub/scm/git/git.git/` permet de récupérer git tel qu'il est au moment de l'écriture de ce texte. Sur le même site est distribué le noyau Linux en git.

[19] Traditionnellement, il n'existait aucun dépôt officiel du noyau Linux en cours de développement. Cela a changé avec git.

```
% cg-log
commit caf39e87cc1182f7dae84eefc43ca14d54c78ef9
tree e8caef545d8c97d839a085dac00f2dd7e2fd95c4
parent 34bb61f9ddabd7a7f909cbfb05592eb775f6662a
author Linus Torvalds <torvalds@g5.osdl.org> Wed, 07 Sep 2005 18:44:33 -0700
committer Linus Torvalds <torvalds@g5.osdl.org> Wed, 07 Sep 2005 18:44:33 -0700

    [SCSI] Re-do "final klist fixes"

    With the previous commit that introduces the klist enhancements, we can
    now re-do 2b7d6a8cb9718fc1d9e826201b64909c44a915f4 again.
```

Exemple 6 – *Résultat de `cg-log` (notez, dans le commentaire, la référence à un autre patch en donnant son identificateur)*

[7] OpenBSD, *OpenCVS, a FREE implementation of the Concurrent Versions System*. http://www.opencvs.org/.

[8] Martin Pool, *Integrals and derivatives*. July 2004. http://sourcefrog.net/weblog/software/vc/derivatives.html.

[9] The Subversion Team, *Subversion, a version control system*. http://subversion.tigris.org/.

[10] C. Michael Pilato Ben Collins-Sussman, Brian W. Fitzpatrick. *Version Control with Subversion*. Numéro 0-596-00448-6 dans ISBN. O'Reilly, 2004.

[11] Mike Mason. *Pragmatic Version Control using Subversion*. Numéro 0-9745140-6-3 dans ISBN. Pragmatic Programmers, 2005.

[12] Tom Lord, *Diagnosing Subversion*. February 2003. http://web.mit.edu/ghudson/thoughts/diagnosing.

[13] Greg Hudson, *A response to Tom Lord's 'Diagnosing svn'*. January 2004. http://web.mit.edu/ghudson/thoughts/undiagnosing.

[14] Ian Bicking, *Distributed vs. Centralized Version Control*. August 2005. http://blog.ianbicking.org/distributed-vs-centralized-scm.html.

[15] ChiaLiangKao, *svk, a decentralized version control system, using Subversion*. http://svk.elixus.org/.

[16] Tom Lord, *tla, a revision control system*. http://www.gnu.org/software/gnu-arch/.

[17] *Mercurial, a distributed SCM*. http://selenic.com/mercurial/.

[18] *Monotone, a free distributed version control system*. http://venge.net/monotone/.

[19] *Cogito, a version control system layered on top of git*. http://www.kernel.org/pub/software/scm/cogito/.

[20] *Codeville, a distributed version control system*. http://www.codeville.org/.

[21] Lele Gaifax, *Tailor.py, A tool to migrate changesets between VCS*. http://www.darcs.net/DarcsWiki/Tailor.