Les nouveaux systèmes de gestion de version (VCS)

Stéphane Bortzmeyer
AFNIC
bortzmeyer@nic.fr

6 décembre 2005

Exposé libre

Rappel : que fait un VCS

Rappel : que fait un VCS

1. C'est un outil de voyage dans le temps : il garde toutes les versions d'un fichier,

Rappel : que fait un VCS

1. C'est un outil de voyage dans le temps : il garde toutes les versions d'un fichier,
2. C'est un outil de travail en groupe : il gère les accès multiples.

Le paysage en 2000

Le paysage en 2000

- Le travail en équipe a fini par être admis.

Le paysage en 2000

- Le travail en équipe a fini par être admis.
- Le verrouillage préalable a fini par être abandonné (gros blocage psychologique).

Le paysage en 2000

- Le travail en équipe a fini par être admis.
- Le verrouillage préalable a fini par être abandonné (gros blocage psychologique).
- Domination écrasante de CVS.

Le paysage en 2000

- Le travail en équipe a fini par être admis.
- Le verrouillage préalable a fini par être abandonné (gros blocage psychologique).
- Domination écrasante de CVS.
- Grâce à SourceForge, tout le monde est passé à CVS.

Les limites de CVS

La liste est longue mais attention : toutes n'ont pas la même importance pour tout le monde.

Les limites de CVS

La liste est longue mais attention : toutes n'ont pas la même importance pour tout le monde.

1. Pas de `commit` atomique : si on *committe* deux fichiers, on peut n'en enregistrer qu'un. Le *patch* est par fichier, pas par `commit`.

Les limites de CVS

La liste est longue mais attention : toutes n'ont pas la même importance pour tout le monde.

1. Pas de `commit` atomique
2. Pas de renommage d'un fichier. Si on change de nom, on perd l'historique.

Les limites de CVS

La liste est longue mais attention : toutes n'ont pas la même importance pour tout le monde.

1. Pas de `commit` atomique
2. Pas de renommage d'un fichier.
3. Pas de versionnement des répertoires. Un répertoire est juste un container.

Les limites de CVS

La liste est longue mais attention : toutes n'ont pas la même importance pour tout le monde.

1. Pas de `commit` atomique
2. Pas de renommage d'un fichier.
3. Pas de versionnement des répertoires.
4. Pas de versionnement des méta-données. Exemple : les permissions.

Les limites de CVS

La liste est longue mais attention : toutes n'ont pas la même importance pour tout le monde.

1. Pas de `commit` atomique
2. Pas de renommage d'un fichier.
3. Pas de versionnement des répertoires.
4. Pas de versionnement des méta-données.
5. Code peu clair et pas documenté (ni structuré). Plus de vrai développement (OpenCVS ?).

Les limites de CVS

La liste est longue mais attention : toutes n'ont pas la même importance pour tout le monde.

1. Pas de `commit` atomique
2. Pas de renommage d'un fichier.
3. Pas de versionnement des répertoires.
4. Pas de versionnement des méta-données.
5. Code peu clair et pas documenté (ni structuré).
6. Travailler sur un code tiers est pénible (*vendor branch*).

Les limites de CVS

La liste est longue mais attention : toutes n'ont pas la même importance pour tout le monde.

1. Pas de `commit` atomique
2. Pas de renommage d'un fichier.
3. Pas de versionnement des répertoires.
4. Pas de versionnement des méta-données.
5. Code peu clair et pas documenté (ni structuré).
6. Travailler sur un code tiers est pénible (*vendor branch*).
7. Les branches sont lentes et très compliquées à utiliser.

Les limites de CVS

La liste est longue mais attention : toutes n'ont pas la même importance pour tout le monde.

1. Pas de `commit` atomique
2. Pas de renommage d'un fichier.
3. Pas de versionnement des répertoires.
4. Pas de versionnement des méta-données.
5. Code peu clair et pas documenté (ni structuré).
6. Travailler sur un code tiers est pénible (*vendor branch*).
7. Les branches sont lentes et très compliquées à utiliser.
8. Pas beaucoup de travail possible en déconnecté (même `cvs diff` demande le réseau).

Et la mère des limites de CVS

Système centralisé

Dépôt centralisé, avec nette séparation des *commiteurs* et des autres. C'est toute la question des systèmes de VCS **centralisés** comme CVS contre les systèmes de VCS **décentralisés** comme git ou darcs.

Regrouper les changements

L'unité de travail de CVS est le fichier. 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 reflète cet état de fait.

Unité de travail : le *changeset*

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

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

Nommer les changements

Le *patch* a un identificateur (**unique** ou espéré tel) :

Plusieurs commandes (par exemple l'affichage de l'historique) peuvent prendre cet identificateur en paramètre.

Les identificateurs servent aussi à la communication entre humains.

Nommer les changements

Le *patch* a un identificateur (**unique** ou espéré tel) :

- un numéro de version pour Subversion (“r42”),

Plusieurs commandes (par exemple l'affichage de l'historique) peuvent prendre cet identificateur en paramètre.

Les identificateurs servent aussi à la communication entre humains.

Nommer les changements

Le *patch* a un identificateur (**unique** ou espéré tel) :

- un numéro de version pour Subversion (“r42”),
- une chaîne de caractères pour darcs (“Remarques orthographiques de Machin”)

Plusieurs commandes (par exemple l'affichage de l'historique) peuvent prendre cet identificateur en paramètre.

Les identificateurs servent aussi à la communication entre humains.

Nommer les changements

Le *patch* a un identificateur (**unique** ou espéré tel) :

- un numéro de version pour Subversion (“r42”),
- une chaîne de caractères pour darcs (“Remarques orthographiques de Machin”)
- un résumé cryptographique (*hash*) pour monotone, git ou Mercurial (“d02c12f943af83659cf6b3fb2c94d2655b1ab15c”)

Plusieurs commandes (par exemple l’affichage de l’historique) peuvent prendre cet identificateur en paramètre.

Les identificateurs servent aussi à la communication entre humains.

Format de *patch* : exemple darcs

```
[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,
```

Subversion

Dit aussi CVS++ *"CVS is an excellent, proven model for version control ; it just wasn't implemented well."*

`http://subversion.tigris.org/`

Subversion

Dit aussi CVS++ *"CVS is an excellent, proven model for version control ; it just wasn't implemented well."*

`http://subversion.tigris.org/`

Subversion est délibérément conçu pour être proche de CVS. On remplace `cvs` par `svn` dans la commande et l'**utilisateur** de CVS est opérationnel en cinq minutes.

Exemple Subversion

```
AFGNIC/documentation % svn log Backup
------------------------------------------------------------
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
```

Notez que les numéros de révision sont par *patch*, pas par fichier. Et ils sont globaux au dépôt (d'où le saut ici, de 39 à 56).

Changements pour l'utilisateur

Changements pour l'utilisateur

- `svn diff` et `svn status` marchent en déconnecté.

Changements pour l'utilisateur

- `svn diff` et `svn status` marchent en déconnecté.
- `commits` atomiques (numéro de version par *patch*)

Changements pour l'utilisateur

- `svn diff` et `svn status` marchent en déconnecté.
- `commits` atomiques (numéro de version par *patch*)
- Répertoires et méta-données versionnées

Changements pour l'utilisateur

- `svn diff` et `svn status` marchent en déconnecté.
- `commits` atomiques (numéro de version par *patch*)
- Répertoires et méta-données versionnées
- `svn rename` !

Changements pour l'administrateur

Changements pour l'administrateur

- Choix du dorsal, ou les joies de DB.
  Par défaut, Subversion utilise Berkeley DB, pour avoir une "vraie" base de données.
  Mais il peut aussi utiliser un dorsal "fichier", nommé FS-FS.
  Dans les deux cas, plus question d'éditer le dépôt à la main.

Changements pour l'administrateur

- Choix du dorsal, ou les joies de DB.
  Par défaut, Subversion utilise Berkeley DB, pour avoir une "vraie" base de données.
  Mais il peut aussi utiliser un dorsal "fichier", nommé FS-FS.
  Dans les deux cas, plus question d'éditer le dépôt à la main.
- *Bindings* très pratiques (et *log* en XML).

Administration Subversion

Serveur réseau : Apache 2 (et WebDAV, RFC 2518 et 3253) par défaut. Mais aussi un simple serveur à lui, qui peut être tunnelé dans SSH.

Attention aux corruptions (permissions incorrectes du dépôt $\longrightarrow$ corruption) de la base.

Les systèmes décentralisés

Ici, on change de paradigme.

Les systèmes centralisés, comme CVS et Subversion se caractérisent par un dépôt **privilégié**.

Les systèmes décentralisés

Ici, on change de paradigme.

Les systèmes centralisés, comme CVS et Subversion se caractérisent par un dépôt **privilégié**.

Donc :

- Certaines opérations sont impossibles en déconnecté (pas de `svn log`) ou si on n'est pas *committeur* (problème de cvsup).

Les systèmes décentralisés

Ici, on change de paradigme.

Les systèmes centralisés, comme CVS et Subversion se caractérisent par un dépôt **privilégié**.

- Certaines opérations sont impossibles en déconnecté
- Les développements parallèles (stable vs. instable, par exemple) nécessitent des acrobaties (branches de CVS).

Les systèmes décentralisés

Ici, on change de paradigme.

Les systèmes centralisés, comme CVS et Subversion se caractérisent par un dépôt **privilégié**.

- Certaines opérations sont impossibles en déconnecté
- Les développements parallèles (stable vs. instable, par exemple) nécessitent des acrobaties
- Il faut désigner des **committeurs**, des privilégiés ($\longrightarrow$ disputes)

Les systèmes décentralisés

Ici, on change de paradigme.

Les systèmes centralisés, comme CVS et Subversion se caractérisent par un dépôt **privilégié**.

- Certaines opérations sont impossibles en déconnecté
- Les développements parallèles (stable vs. instable, par exemple) nécessitent des acrobaties
- Il faut désigner des **committeurs**, des privilégiés
- 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).

Bons points avec les centralisés

Mais :

Bons points avec les centralisés

Mais :

- On a un dépôt de référence, connu,

Bons points avec les centralisés

Mais :

- On a un dépôt de référence, connu,
- On simplifie l'implémentation et l'usage.

Les VCS décentralisés

Principe : chaque développeur peut avoir son dépôt. Dans lequel il peut committer.

Les VCS décentralisés

Principe : 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.

Les VCS décentralisés

Principe : 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.

darcs

VCS décentralisé. Chaque copie est un dépôt complet. Chaque branche est un dépôt.

`http://www.darcs.net/`

Récent mais déjà utilisé en production.

L'identificateur de *patch* est une chaîne de caractères. Le message de *commit* n'est donc pas purement informatif, c'est un identificateur $\Rightarrow$ il faut une charte de nommage des messages.

Travail avec darcs

```
% 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'
```

Pas de différence avec un système centralisé encore.

darcs avec plusieurs copies

```
% 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".

Je travaille 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'
```

On peut donc “committer” en local. D'autres peuvent utiliser mon dépôt, c'est un dépôt de première classe.

On resynchronise les dépôts

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

Mais on n'est pas obligé.

Et on peut choisir les patches à envoyer.

darcs + RequestTracker

On utilise RequestTracker. 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
```

Sélection des *patches*

On peut n'envoyer que les *patches* 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 *patch* et pas les autres.

darcs pour l'administrateur

Écrit en Haskell.

Un compilo Haskell répandu (ghc) mais qui échoue sur des architectures rares (mips/Linux, sparc/NetBSD).

Le serveur a besoin de darcs.

Exemple Haskell

```
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')
```

La fonction commute est définie par*pattern matching* (d'où les deux définitions).

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.

$P(t) \rightarrow t'$

Série de *patches*

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}

Commutation

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**.

À quoi ça sert ?

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**.

Extraire un *patch*

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

Mercurial

Un autre VCS décentralisé.

`http://www.selenic.com/mercurial/`

Utilisé pour plusieurs gros projets.

Mercurial

Un autre VCS décentralisé.

`http://www.selenic.com/mercurial/`

Utilisé pour plusieurs gros projets.

Très rapide, même sur de gros dépôts.

Mercurial

Un autre VCS décentralisé.

`http://www.selenic.com/mercurial/`

Utilisé pour plusieurs gros projets.

Très rapide, même sur de gros dépôts.

Utilisation délicate : il peut y avoir plusieurs branches par dépôt, par exemple.

L'historique de Mercurial

```
% hg log -v
changeset:   14:74e3e4e6627523e00ed98bf9f8a045dff05653e2
user:        stephane@ludwigVI.sources.org
date:        Sun Oct  2 17:29:50 2005
files:       hack/obfuscate.c
description:
Even better code, with less comments
```

Chaque *changeset* a deux identificateurs, un local au dépôt (ici 14) et un résumé (ici 74e3...) qui est global.

git

Développé spécialement pour le noyau Linux par Torvalds.

Plutôt un système de fichiers temporel qu'un "vrai" VCS.

Bâti sur le principe des objets immuables, identifiés par leur résumé.

Interface très rude (mieux vaut utiliser cogito au dessus).

Historique de git

```
% cg-log -f
commit 8c36bcde3ed5c708f378b1cc72f5cf88d8510631
tree 48928a80c6e092777c076bf91ce4faf391aa8be2
parent f4c67c56c63ec452fb2f7fc7bf05abba631c5207
author Tobias Klauser <tklauser@access.unizh.ch> Sat, 26 Feb 200
committer Greg KH <gregkh@suse.de> Fri, 06 May 2005 13:51:54 -07
    ...
    As requested in the TODO file of hotplug-ng, here's a manpag
    ...
```

On note que tout est objet dans git : le *commit*, le répertoire (*tree*), le *commit* précédent (*parent*).

Coexistence

Pronostic : la variété des VCS décentralisés ne se calmera pas de si tôt.

Il faudra donc s'habituer à la coexistence : beaucoup de développeurs travailleront avec des dépôts gérés par des logiciels différents.

Donc, il faudra convertir, peut-être dans les deux sens, avec des outils comme Tailor :

`http://www.darcs.net/DarcsWiki/Tailor`

L'avenir

L'avenir

1. Sélection darwinienne parmi les nombreux VCS décentralisés,

L'avenir

1. Sélection darwinienne parmi les nombreux VCS décentralisés,
2. Puis marginalisation des VCS centralisés.