Introduction à CVS

Manuel Bouyer

18 mai 1998

1 Introduction

Les problèmes posés par le travail en commun de plusieurs personnes sur un même ensemble de fichiers sont multiples. Outre les problèmes d'organisation et de communication, un certain nombre de problèmes techniques se posent :

- Comment faire si deux personnes éditent en même temps un même fichier ? dans le cas général, la dernière personne qui sauve le fichier écrase les changements fait par l'autre.
- Comment voir qui a modifié un fichier, et ce qui a été changé ?
- Pour quelle raison un fichier a-t-il été modifié ?

CVS permet de résoudre ces problèmes. Il conserve l'historique des modifications des fichiers, avec le nom de login de la personne ayant effectué la modification, et un commentaire entré par cette même personne. Chaque personne travaille sur sa propre copie des fichiers, et peut à tout moment demander à CVS de synchroniser cette copie avec les fichiers communs.

CVS est utilisé par la plupart des grands projets de l'Internet. Par exemple, le système FreeBSD (UNIX gratuit) est développé par plus de 60 personnes, répartis sur l'ensemble du globe, ayant le droit (au sens UNIX du terme) de modifier les sources du système. Dans ce cas, CVS permet d'assurer la cohérence des sources, une partie de la communication entre les développeurs et facilite grandement la correction des inévitables erreurs. Sans CVS, il faudrait probablement une personne s'occupant exclusivement de la gestion des sources de FreeBSD.

2 Fonctionnement de CVS

2.1 Fonctionnement interne

Afin de conserver l'historique des fichiers, CVS utilise sa propre arborescence, ayant la même structure (les même répertoires) que l'arborescence d'origine. Cette arborescence s'appelle le *repository*. Dans ce *repository*, CVS va stocker les fichiers, ainsi que l'historique de leur modifications. En principe l'utilisateur n'a pas à modifier directement ces fichiers.

A chaque version d'un fichier correspond un numéro de révision (`revision number` en anglais) unique. Ces numéros sont de la forme « 1.4 », « 1.3.2.7 », ou « 1.5.2.2.4.1 ». CVS n'est pas limité à un développement linéaire. L'arbre de développement peut être séparé en branches, permettant, par exemple, à un même logiciel d'évoluer dans plusieurs directions, sans être obligé d'avoir plusieurs *repository*. Un numéro de branche est constitué d'un nombre impair de nombres (« 1 », « 1.3.2 », ou « 1.5.2.2.4 » dans l'exemple précédent). La branche comportant un seul nombre (ce dernier vaut généralement « 1 ») s'appelle la branche principale (*main trunc*).

CVS permet d'associer un nom symbolique (*tag*) à une revision de fichier, ou une branche. Par exemple, un logiciel est commercialisé sous la référence «Version 2.1», on pourra attribuer le nom «v2_1» aux révisions des fichiers constituant le source du programme. On pourra ensuite continuer le développement, et utiliser le *tag* «v2_1» pour retrouver les sources de la version 2.1 du logiciel. CVS ne permet pas d'imposer un numéro de révision à un fichier, et en principe l'utilisateur se servira peu de ces numéros, il utilisera des *tags*.

2.2 Utilisation

Lorsqu'il désire travailler sur un ou plusieurs fichiers, l'utilisateur devra en obtenir une copie dans un autre répertoire (*checkout*). La racine du *repository* devra être spécifiée soit sur la ligne de commande, ou dans la variable d'environnement *CVSROOT*. Il peut obtenir soit la version courante, soit une version antérieure. Lorsque l'utilisateur a terminé les modifications qu'il avait à apporter, il demande à CVS d'intégrer ses changements au *repository* (*commit*). A ce moment là, CVS va demander à l'utilisateur d'entrer quelques lignes décrivant les changements. Cette description va être conservée dans le *repository*, et peut également être transmise aux autres développeurs, par courrier électronique par exemple. Ses modifications deviennent alors disponibles aux autres utilisateurs, qui peuvent mettre à jour leur propre copie des sources (*update*). Si au moment du *commit*, quelqu'un d'autre a également modifié un fichier dans le *repository*, CVS va essayer de résoudre les éventuels conflits, ou demandera à l'utilisateur de le faire s'il n'y arrive pas.

2.3 La chaîne $Id: $

Lors d'un *checkout* ou d'un *update*, CVS va rechercher dans le fichier la chaîne de caractères `$Id: $`, et va mettre entre le «:» et le «$» les informations sur le fichier: numéro de révision, date, et nom de login de la personne ayant exécuté le *commit* de cette révision. Cette chaîne peut être placée à n'importe quel endroit dans le fichier (commentaire en C, variable/chaîne de caractères, etc ...). Voici quelque exemples :

```
/* $Id: vm_pager.c,v 1.23 1996/05/18 03:38:05 dyson Exp $ */

char *CVSId = "$Id: strcpy.c,v 1.1.1.6 1996/05/30 21:53:03 bouyer Exp $"
```

Inclure cette chaîne dans un fichier permet de retrouver facilement la révision utilisée, lorsque ce fichier a été transmis à une tierce personne. Cela permet de préciser de manière très exacte de quoi l'on parle.

3 Résumé des commandes

L'utilisation de CVS se fait depuis la ligne de commande, dans un shell. Une invocation de CVS prend la forme suivante :
cvs <options globales> command <options> arguments
cvs étant ici une commande UNIX. Les options globales sont les options communes à toute les commandes CVS. Vient ensuite la commande CVS (un mot), suivit des options spécifiques à cette commande, et de ses arguments.

3.1 Options globales

Les principales options globales sont les suivantes :

– **-d <repository root>** indique la racine du *repository* à utiliser, peut également être spécifié par la variable d'environnement **CVSROOT**.

– **-e <editor>** indique l'éditeur à utiliser pour entrer les commentaires lors du *commit*, peut également être spécifié par la variable d'environnement **EDITOR**.

– **-H** affiche une aide sur la commande, ou la liste des commandes disponibles si aucune commande n'est spécifiée.

– **-n** affiche les messages de la commande exécutée, mais ne change aucun fichier.

3.2 Commandes

Les principales commandes de CVS sont les suivantes :

- **add <file>** ajoute le fichier ou répertoire spécifié au *repository*. Le fichier ajouté doit être présent dans le répertoire courant.
- **remove <file>** attribue une révision spéciale au fichier spécifié, qui ne sera plus retrouvé lors d'un *checkout* ou *update*. Le fichier ne doit pas être présent dans le répertoire courant.
- **checkout [options] <directory>** crée un répertoire de travail contenant les fichiers sources du répertoire spécifié. On peut également utiliser les synonymes **co** ou **get**. Les principales options sont les suivantes :
  - **-D <date>** voir plus bas.
  - **-r <tag>** voir plus bas.
- **commit [file]** intègre les changements effectués sur la copie locale au *repository*. On peut utiliser le synonyme **ci**. Si aucun fichier n'est spécifié, CVS va traiter récursivement les fichiers depuis le répertoire courant.
- **diff [options] [-r rev1 | -D date1] [-r rev2 |-D date2] [file]** compare un ou plusieurs fichier, en affichant les différences, entre deux révisions, une révision et le(s) fichier(s) courant, ou les fichiers courants et le *repository*. Si aucun fichier n'est spécifié, CVS traite les répertoires récursivement depuis le répertoire courant. Les principales options (autres que -D ou -r) sont :
  - **-c** affiche le résultat au format `context diff`.
  - **-u** affiche le résultat au format `unified diff`.

  Ces options sont en fait passées au programme *diff(1)*.
- **import [options] repository vendortag releasetag** Cette commande est utilisée pour incorporer une arborescence déjà existante dans le *repository*. Elle est également utilisée pour créer un nouveau *repository*, il suffit pour cela d'exécuter cette commande depuis un répertoire vide.
- **log [file]** Affiche les messages entrés lors des *commit* pour le fichier spécifié. Si aucun fichier n'est spécifié, CVS traite les répertoires récursivement depuis le répertoire courant.

- **release [-d] <directory>** indique s'il reste des fichiers modifiés dans le répertoire, avant d'effacer celui-ci. Lorsque -d est spécifié, le répertoire est physiquement effacé après confirmation.

- **status [files]** Affiche l'état des fichiers spécifiés. Si aucun fichier n'est spécifié, CVS traite les répertoires récursivement depuis le répertoire courant.

- **update [options] [files]** Met à jour les fichiers, en accord avec le *repository*, en conservant les changements locaux effectués. Utilisé pour mettre à jour ses fichiers lorsqu'une autre personne a modifié le *repository*. Si aucun fichier n'est spécifié, CVS traite les répertoires récursivement depuis le répertoire courant. Les principales options sont :

  - **-r <tag>** voir plus bas.
  - **-D <date>** voir plus bas.

3.3 Les options -r et -D

Ces options ne sont pas des options globales, mais sont utilisées par plusieurs commandes.

-r <rev> permet de spécifier une révision particulière au lieu de la révision courante. La révision peut être spécifiée soit par un *tag* symbolique (fichier ou branche), soit par numéro de version (fichier ou branche également).

-D <date> désigne la dernière révision avant la date spécifiée, d'un ou plusieurs fichiers. La date peut être spécifiée par l'une des formes suivantes :

- a month ago
- 2 hours ago
- 400000 seconds ago
- last year
- last monday
- yesterday
- a fornight ago
- 3/31/92 10:00:07 PST
- January 23, 1995 10:05pm
- 20:00 GMT

4 Exemple d'utilisation

Dans ce chapitre, nous allons voir un exemple de session CVS. Les parties en gras (**comme ceci**) sont les lignes de commande entrées. Les parties en italique (*comme ceci*) sont les messages affichés à l'écran. Les commentaires sont en écriture normale.

La première chose à faire est de mettre la racine du repository dans la variable d'environnement $CVSROOT :
setenv CVSROOT /home/bouyer/cvsroot
Ensuite, créer un répertoire dans le repository, qui conservera le futur programme :
mkdir toto (le nom n'a pas d'importance ici, c'est un nom temporaire)
cd toto
cvs import hello ensta v0_0 (le répertoire crée s'appellera hello, ensta étant le nom de la « vendor branch » et « v0_0 » le nom du « vendor tag », ces deux derniers étant peu important ici)
CVS lance alors un éditeur (vi par défaut) pour entrer le commentaire associé. Cela donne quelque chose comme ça :

```
debut du projet 'hello', developpe a l'ENSTA.
CVS: ----------------------------------------------------------------------
CVS: Enter Log.  Lines beginning with `CVS: ' are removed automatically
CVS:
CVS: ----------------------------------------------------------------------
```

CVS doit normalement répondre ensuite
No conflicts created by this import
On peut alors effacer le répertoire toto, et commencer à travailler sur le projet :
cd ..
rmdir toto
cvs checkout hello (on obtient une copie de hello)
cvs checkout: Updating hello
cd hello
On peut alors commencer le travail. Créons un fichier hello.c contenant :

```
/* $Id:$ */
main()
{
printf("hello world\n");
}
```

Il faut maintenant le mettre dans le repository :
cvs add hello.c
cvs add: use 'cvs commit' to add this file permanently
Le fichier n'est pas encore dans le repository, il y sera au prochain commit.
cvs commit
CVS lance alors l'éditeur. On pourra entrer par exemple :

```
Commit du premier fichier du projet hello.
CVS: ----------------------------------------------------------------------
CVS: Enter Log.  Lines beginning with `CVS: ' are removed automatically
CVS:
CVS: Committing in .
CVS:
CVS: Added Files:
CVS:    hello.c
CVS: ----------------------------------------------------------------------
```

RCS file: /home/bouyer/cvsroot/hello/hello.c,v
done
Checking in hello.c;
/home/bouyer/cvsroot/hello/hello.c,v <– hello.c
initial revision: 1.1
done
Le fichier a été ajouté au repository, avec la version 1.1.

Supposons qu'une deuxième personne commence à travailler sur le projet :
cvs checkout hello
cvs checkout: Updating hello
U hello/hello.c
cd hello
Le fichier hello.c contiendra alors:

```
/* $Id: hello.c,v 1.1 1996/12/12 20:23:06 bouyer Exp $ */
main()
{
printf("hello world\n");
}
```

(noter la chaîne de caractères suivant $Id) Il va ajouter un Makefile, permettant la compilation automatique du programme :
vi Makefile
cvs add Makefile
cvs add: scheduling file 'Makefile' for addition

cvs add: use 'cvs commit' to add this file permanently
cvs commit
Il entrera comme commentaires :

```
Makefile pour hello.
CVS: ----------------------------------------------------------------------
CVS: Enter Log.  Lines beginning with `CVS: ' are removed automatically
CVS:
CVS: Added Files:
CVS:    Makefile
CVS: ----------------------------------------------------------------------
```

RCS file: /home/bouyer/cvsroot/hello/Makefile,v
done
Checking in Makefile;
/home/bouyer/cvsroot/hello/Makefile,v <– Makefile
initial revision: 1.1
done

La première personne pourra alors récupérer le Makefile par :
cvs update
cvs update: Updating .
U Makefile
Il continue à travailler sur hello.c. Supposons qu'il le modifie comme suit :

```
/* $Id: hello.c,v 1.1 1996/12/12 20:23:06 bouyer Exp $ */
main()
{
printf("hello world\n");
printf("c'est fini\n");
}
```

cvs commit
(le message explicatif pourra être « ajout d'un printf »)
Checking in hello.c;
/home/bouyer/cvsroot/hello/hello.c,v <– hello.c
new revision: 1.2; previous revision: 1.1
done
Pendant ce temps, la deuxième personne a décidé de mettre une majuscule à « hello world » :

```
/* $Id: hello.c,v 1.1 1996/12/12 20:23:06 bouyer Exp $ */
main()
{
```

```
printf("Hello world\n");
}
```

cvs commit
cvs commit: Examining .
cvs commit: Up-to-date check failed for 'hello.c'
cvs [commit aborted]: correct above errors first!
La version de 'hello.c' présente dans le repository est plus récente que celle du répertoire courant. Il faut mettre la copie locale à jour avant de pouvoir faire un commit :
cvs update
cvs update: Updating .
RCS file: /home/bouyer/cvsroot/hello/hello.c,v
retrieving revision 1.1
retrieving revision 1.2
Merging differences between 1.1 and 1.2 into hello.c
rcsmerge: warning: conflicts during merge
cvs update: conflicts found in hello.c
C hello.c
Une même zone d'un fichier a été modifiée par 2 personnes différentes : c'est ce que l'on appelle un conflit. Si le fichier avait été modifié à 2 endroits différents (séparés par plus de 10 lignes), CVS aurait pu recréer le fichier tout seul. Le fichier hello.c du répertoire courant contient maintenant:

```
/* $Id: hello.c,v 1.2 1996/12/12 20:53:36 bouyer Exp $ */
main()
{
< < < < < < < hello.c
printf("Hello world\n");
||||||| 1.1
printf("hello world\n");
=======
printf("hello world\n");
printf("c'est fini\n");
> > > > > > > 1.2
}
```

La zone de conflit se trouve entre <<<<<<< et >>>>>>>. Entre <<<<<<< et ||||||| se trouve le morceau de fichier tel qu'il était dans le répertoire courant, entre ||||||| et ======= tel qu'il était dans le repository lors de la dernière mise à jour (commit ou update), et entre ======= et >>>>>>> tel qu'il est actuellement dans le repository. La bonne version est évidemment :

```
/* $Id: hello.c,v 1.2 1996/12/12 20:53:36 bouyer Exp $ */
main()
{
printf("Hello world\n");
printf("c'est fini\n");
}
```

On peut alors faire le commit :
cvs commit
Checking in hello.c;
/home/bouyer/cvsroot/hello/hello.c,v <– hello.c
new revision: 1.3; previous revision: 1.2
done

On peut maintenant obtenir diverses informations sur les fichiers :
cvs log hello.c
RCS file: /home/bouyer/cvsroot/hello/hello.c,v
Working file: hello.c
head: 1.3
branch:
locks: strict
access list:
symbolic names:
keyword substitution: kv
total revisions: 3; selected revisions: 3
description:
————————————-
revision 1.3
date: 1996/12/12 21:12:11; author: bouyer; state: Exp; lines: +2 -2
h->H.
————————————-
revision 1.2
date: 1996/12/12 20:53:36; author: bouyer; state: Exp; lines: +2 -1
Ajout d'un printf.
————————————-
revision 1.1
date: 1996/12/12 20:23:06; author: bouyer; state: Exp;
Commit du premier fichier du projet hello.
===

On obtient ici, outre diverses informations internes à CVS, l'historique du

fichier, avec un commentaire sur chaque changement. La commande suivante donnera les différences entre la version 1.1 du fichier et la version actuelle (au format context diff) :
cvs diff -c -r1.1 hello.c

```
Index: hello.c
===================================================================
RCS file: /home/bouyer/cvsroot/hello/hello.c,v
retrieving revision 1.1
retrieving revision 1.3
diff -c -r1.1 -r1.3
*** hello.c     1996/12/12 20:23:06     1.1
--- hello.c     1996/12/12 21:12:11     1.3
***************
*** 1,5 ****
! /* $Id: hello.c,v 1.1 1996/12/12 20:23:06 bouyer Exp $ */
  main()
  {
! printf("hello world\n");
  }
--- 1,6 ----
! /* $Id: hello.c,v 1.3 1996/12/12 21:12:11 bouyer Exp $ */
  main()
  {
! printf("Hello world\n");
! printf("c'est fini\n");
  }
```

Le même résultat aurait pu être obtenu en utilisant l'option -D à la place de -r (par exemple: **cvs diff -c -D"2 days ago" hello.c** si le fichier hello.c avait le numéro de révision 1.1 il y a 2 jours). La commande suivante donnera les differences entre les versions 1.1 et 1.2 :
cvs diff -c -r1.1 -r1.2 hello.c

```
 Index: hello.c
===================================================================
RCS file: /home/bouyer/cvsroot/hello/hello.c,v
retrieving revision 1.1
retrieving revision 1.2
diff -c -r1.1 -r1.2
```

```
*** hello.c     1996/12/12 20:23:06     1.1
--- hello.c     1996/12/12 20:53:36     1.2
***************
*** 1,5 ****
! /* $Id: hello.c,v 1.1 1996/12/12 20:23:06 bouyer Exp $ */
  main()
  {
  printf("hello world\n");
  }
--- 1,6 ----
! /* $Id: hello.c,v 1.2 1996/12/12 20:53:36 bouyer Exp $ */
  main()
  {
  printf("hello world\n");
+ printf("c'est fini\n");
  }
```