Mini-*HowTo* CVS sous Unix

(révision 1.19, 05 Octobre 2004)

Antoine MARIN*

Table des matières

*antoine!marin()jouy!inra!fr

Ce mini-*HowTo* CVS (*Concurrent Version System*) doit vous permettre de créer et participer à un projet avec CVS sous Unix (*a fortiori* Linux).

Je ne parlerai pas d'installation logicielle ici, mais uniquement de configuration et d'utilisation.

Ce mini-*HowTo* est issu de la lecture du manuel de référence [1] [1], ainsi que de nombreux liens du site CVS BUBBLES [2], qui comporte beaucoup d'informations utiles, comme les applications utilisant CVS, CVS pour Mac et Windows, des listes de diffusion, etc. CVS est actuellement développé par Cyclic Software et est sous licence GNU.

J'ai écrit ce document pour éviter aux lecteurs désireux d'utiliser CVS sous Unix, de lire les 170 pages du manuel de référence, et pouvoir ainsi avoir une prise en main rapide de CVS, sans forcément comprendre (ce qui n'est pas utile) **tout** des mécanismes sous-jacents (ce qui est toujours mon cas, comme la plupart des utilisateurs). Je recommande néanmoins **fortement** d'avoir le manuel de référence sous la main, ne serait-ce que pour l'annexe B qui est un descriptif rapide (avec toutes les options) de toutes les commandes CVS.

1 Introduction – la philosophie de CVS

CVS permet de garder l'historique des modifications de fichiers ASCII ou binaires. Il garde en mémoire **qui** a fait **quoi**, **pourquoi** et **comment** (si l'auteur a bien voulu le dire).

Cela permet bien sûr d'avoir toujours accès aux versions antérieures de son travail, et aussi de travailler à plusieurs sur le même projet, c'est-à-dire de modifier « en même temps » le même fichier. Ce « en même temps » existe réellement car chaque utilisateur travaille sur une copie de l'original.

En fait, personne ne travaille jamais sur les originaux (même l'administrateur du projet) ; toutes les modifications des originaux passent par CVS. Les fichiers originaux sont modifiés quand un utilisateur renvoie ses nouvelles versions des fichiers. Les anciens fichiers ne sont jamais détruits (ce qui évite l'écrasement d'un projet par un utilisateur peu scrupuleux) ; seules les différences sont enregistrées, il est alors toujours possible de revenir en arrière, et ce indépendamment pour chaque fichier.

Chaque fichier possède donc son propre historique, et il est possible (et recommandé, au moins pour l'administrateur) de garder un « souvenir », à l'aide d'une marque (*tag*), d'une version d'un projet (*release*) à un moment donné pour pouvoir le rappeler en une seule fois dans cet état, si cela est nécessaire.

[1] `http://www.loria.fr/cgi-bin/molli/wilma.cgi/doc.847210383.html`

[2] `http://www.loria.fr/\textasciitildemolli/cvs-index.html`

Toutes les commandes CVS utilisées dans ce document sont présentées avec seulement les options que je pense être les plus utiles (à mon humble avis), pour plus de détails, se référer à la doc complète [1].

Le changement de version d'un fichier est appelé révision *revision*. Notez la différence entre une révision, c'est-à-dire l'enregistrement de l'évolution d'un fichier par CVS (lors de la soumission des modifications), et une version, c'est-à-dire une étape importante de l'évolution d'un fichier ou ensemble de fichiers (projet), enregistrée par l'administrateur à l'aide d'un *tag*.

Chaque révision de fichier doit être accompagnée d'un commentaire de l'utilisateur qui fournit sa modification ; ce commentaire est très utile pour la relecture future des modifications passées, par lui-même comme par les autres. Il ne faut donc pas hésiter à fournir quelques détails qui apporteront plus d'informations que les modifications elles-mêmes.

Voici un exemple de commentaire qui n'apporte pas d'informations par rapport à la lecture des modifications du fichier **barbeau.c** :

– « Ajout de la variable X dans **barbeau.c** »,

il faudrait plutôt écrire :

– « Nouvelle fonctionnalité Y (brève description) grâce à l'ajout de la variable X dans **barbeau.c** ».

2 Créer un projet avec CVS

Ce chapitre est destiné aux personnes qui souhaitent être l'administrateur d'un projet, qu'il soit personnel ou commun. Les autres peuvent passer directement au chapitre 3, en ayant quand même jeté un coup d'œil à la section 2.2.

2.1 Créer le dépôt (*repository*)

La première chose à faire est de créer un dépôt à un endroit donné, par exemple dans le répertoire **/usr/local/cvs**. Il suffit pour cela de le créer :

```
mkdir /usr/local/cvs
```

Il faut ensuite initialiser sa variable d'environnement **CVSROOT** :

```
export CVSROOT=/usr/local/cvs
```

en shell Bourne (sh, bash, ksh, zsh), et :

```
setenv CVSROOT /usr/local/cvs
```

pour csh et tcsh.

Par la suite, il est bon de placer cette commande dans votre fichier .shellrc (.bashrc, .cshrc, etc.) pour éviter de la taper à chaque fois.

Dorénavant, tous les projets que vous administrerez se trouveront dans **$CVSROOT**, soit **/usr/local/cvs** dans notre cas.

2.2 Les variables CVSEDITOR et EDITOR

Profitons-en ici pour initialiser de la même façon la variable CVSEDITOR qui permet à CVS de savoir quel éditeur de texte ouvrir lorsqu'il vous demandera d'entrer un commentaire pour une modification. Si cette variable n'est pas initialisée, CVS cherchera ensuite dans la variable EDITOR puis, si cette dernière n'est pas initialisée non plus, il retournera un message d'erreur comme celui-ci :

```
cvs [commit aborted]: no editor defined, must use -e or -m
```

L'option **cvs -e editor** permet de spécifier un éditeur de texte lors de l'utilisation de la commande (remplacer **editor** par votre éditeur de texte préféré comme vi, emacs ou nedit ...) ; cette option prévaut sur les variables d'environnement.

L'option **cvs commande -m "commentaire"** permet d'éviter l'appel à un éditeur de texte en donnant le commentaire directement sur la ligne de commande, c'est pratique si le commentaire n'est pas long et peut être écrit sur une seule ligne.

2.3 Initialiser le dépôt

Il faut ensuite initialiser le dépôt par la commande :

```
cvs init
```

CVS crée alors un ensemble de fichiers dans un répertoire appelé **CVSROOT** qui se trouve dans **$CVSROOT** [3], qui seront modifiés au cours des projets et qui sont les fichiers de configuration, ainsi que les différents fichiers « souvenirs » de tout ce qui se passe quand CVS est invoqué sur ces répertoires (ces projets).

2.4 Importer ou créer un projet dans le dépôt

Comme nous l'avons dit dans l'introduction, il ne faut pas créer directement de fichiers ni de répertoires dans le dépôt. L'utilisateur peut alors se trouver dans deux situations :

1. vouloir importer sous CVS un projet déjà existant,
2. vouloir créer un projet *de novo*.

Nous allons voir que le second cas n'est pas très différent du premier.

Commençons par le premier cas : disons que votre projet actuel se trouve dans **/home/bossy/mon_projet/sources** (il ne doit contenir **que** des fichiers à importer, ou alors vous pouvez tout importer puis supprimer des fichiers au projet comme nous le verrons plus loin, mais une trace existera **toujours** de ces fichiers

[3] Ne confondez pas **CVSROOT**, la variable d'environnement qui contient le chemin vers le dépôt et **CVSROOT**, le répertoire que CVS crée dans **$CVSROOT** pour y stocker ses informations.

de départ), et que vous vouliez le mettre dans **$CVSROOT/bd/mercury_sources**, il suffit alors de faire :

```
cd /home/bossy/mon_projet/sources
cvs import bd/mercury_sources gueguen version1
```

ou bien :

```
cvs import -m "Premier import des sources"
bd/mercury_sources gueguen version1
```

(sur la même ligne).

L'option **-m** permet de spécifier directement le commentaire en ligne, sinon CVS va ouvrir votre éditeur de texte favori (voir section 2.2).

Les deux mots qui suivent le répertoire de destination sont :

1. **gueguen** qui est la marque du constructeur (*vendor tag*) et
2. **version1** qui est la marque de version (*release tag*).

La marque du constructeur est une notion avancée, pour cela je renvoie au manuel de référence [1] (mettez votre nom par exemple). La marque de version est la marque de la première importation, c'est par ce nom-là que vous apprendrez à rappeler votre toute première importation (ou toute première version du projet si vous voulez).

Si vous voulez créer un projet *de novo*, il suffit d'en faire l'ébauche dans un répertoire quelconque puis de l'importer comme nous venons de le voir (en fait il y a des moyens de le créer directement dans le dépôt, mais pour les opérations sur les répertoires je vous renvoie au manuel de référence [1]).

2.5 Quelques astuces avant de commencer

Pour des raisons pratiques, il est utile de donner un nom (pas trop long si possible) à votre projet, ce qui aura comme conséquence agréable de pouvoir l'appeler par son nom au lieu de sa localisation dans **$CVSROOT** (c'est une sorte d'*alias*). Pour cela, il faut modifier un fichier nommé **modules** qui se trouve dans **$CVSROOT/CVSROOT/**, ceci par l'intermédiaire de CVS.

Par exemple, si vous voulez donner le nom **JB007** à ce projet, il suffit de faire comme ceci (pour le moment contentez-vous de faire comme je vous dis, des explications viendront dans les chapitres suivants) :

– mettez-vous dans un répertoire temporaire et faites :
```
cvs checkout CVSROOT/modules
cd CVSROOT
```
[4]
– editez ce fichier, et après les commentaires (lignes qui commencent par « # ») tapez :
```
JB007 bd/mercury_sources
```

[4] La commande **checkout** ayant reconstruit l'arborescence.

– sauvegardez le fichier, quittez l'éditeur de texte, puis :
```
cvs commit -m "Ajout du module JB007" modules
cd ..
cvs release -d CVSROOT
```

Il est maintenant **fortement** conseillé de **détruire** votre ancien répertoire, pour ne pas être tenté de modifier votre projet en dehors de CVS, ce qui reviendrait à ne pas utiliser CVS pour votre projet !

Attention : avant d'effacer votre ancien répertoire (et toute sa descendance) vérifiez que votre projet a bien été importé en faisant ce qui suit (des explications viendront plus tard) :

– mettez-vous dans un répertoire temporaire et tapez :
```
cvs checkout JB007
diff -r /home/bossy/mon_projet/sources JB007
```
[5]

Vous ne devez avoir aucune différence à part un nouveau répertoire nommé « CVS » dans **JB007** (CVS le crée automatiquement lors du *checkout*), vous pouvez alors faire :

```
rm -fr /home/bossy/mon_projet/sources
```

sinon, recommencez doucement depuis le début du chapitre 2.

Pour finir avec 2 astuces pour le « checkout » :

– `checkout` peut s'écrire simplement `co`
– `checkout` peut se faire dans un répertoire de son choix avec :
```
co -d new_rep <projet>
```

3 Utiliser CVS sur sa propre machine

Vous voulez participer à un projet sous CVS qui se trouve sur votre machine. Soit vous venez de le créer (voir chapitre 2, vous savez donc qu'il s'appelle **JB007**), soit vous savez qu'il existe un projet appelé **JB007**. L'administrateur a dû faire bien attention que les droits d'accessibilité des répertoires CVS soient compatibles avec votre groupe (sinon, faites appel à lui).

Les premières choses à faire sont :

– initialiser sa variable d'environnement CVSROOT (section 2.1),
– initialiser ses variables EDITOR et/ou CVSEDITOR (section 2.2),
– se créer un répertoire de travail :
```
cd ~ ; mkdir cvswork
```
– y aller :
```
cd ~/cvswork
```

[5] Si vous n'aviez pas modifié le fichier modules, vous auriez dû taper : *cvs checkout bd/mercury_sources*

– récupérer le projet **JB007** par la commande :
  `cvs checkout JB007`
– se mettre dans le répertoire contenant le **JB007** :
  `cd JB007`

La commande **checkout** sans options vous permet de récupérer un projet dans sa version la plus récente. Peut-être voulez-vous travailler sur la version **lafontaine** (c'est son petit nom) ? Cela veut dire qu'à un moment donné de ce projet, les fichiers étaient dans un état jugé satisfaisant par l'administrateur du projet et que celui-ci a mis une marque (*tag*) sur tous les fichiers (indépendamment de leur numéro de révision), et il a appelé cette version **lafontaine**. Pour récupérer cette version tapez :

```
cvs checkout -r lafontaine JB007
```

Maintenant vous pouvez modifier les fichiers à votre guise, et quand vous voulez soumettre des modifications (vous êtes le seul juge de cet acte), vous le faites par la commande :

```
cvs commit bize.cpp
```

pour soumettre les modifications sur le fichier **bize.cpp**, ou bien, si vous avez modifié plusieurs fichiers :

```
cvs commit
```

vous permettra de soumettre toutes vos modifications d'un seul coup, et de n'écrire qu'un commentaire pour toutes ces modifications.

NOTE : même si vous soumettez un fichier vide, il sera pris en compte, mais ce ne sera qu'une nouvelle version du fichier précédent qui ne sera pas écrasé, vous risquez seulement que l'administrateur du projet vous contacte rapidement pour vous dire ce qu'il en pense . . .

Si jamais un message du genre :

```
cvs commit: Examining .
cvs commit: Up-to-date check failed for ‘bize.cpp’
cvs [commit aborted]: correct above errors first!
```

apparaît à votre écran, c'est que le fichier **bize.cpp** a été modifié depuis que vous l'avez récupéré, vous devez alors faire une mise à jour (*update*) avant de refaire un **commit**.

Comme la plupart des commandes CVS, **update** s'utilise en précisant ou non un nom de fichier après ; cette commande permet de se mettre à jour par rapport au dépôt, pour un fichier donné ou l'ensemble du projet, par exemple :

```
cvs update bize.cpp
```

Lors d'une mise à jour, CVS rassemble (*merge*) alors vos modifications, avec celles des dernières versions existantes. En général cela se passe sans mal, mais

si vos modifications sont importantes (d'où l'intérêt de ne pas soumettre ses modifications trop peu fréquemment) ou si celles qui ont été faites sur ce fichier par quelqu'un d'autre sont trop importantes, vous pouvez vous retrouver avec ce genre de message :

```
cvs update: Updating .
RCS file: /usr/local/cvs/ma_bd/ricard.sql,v
retrieving revision 1.5
retrieving revision 1.6
Merging differences between 1.5 and 1.6 into ricard.sql
rcsmerge: warning: conflicts during merge
cvs update: conflicts found in ricard.sql
C ricard.sql
```

et là, vous présentez un problème.

Quelques commentaires sur la syntaxe ci-dessus :

- **/usr/local/cvs/ma_bd/** est le chemin du projet,
- **ricard.sql** est le fichier problématique,
- **ricard.sql,v** est le nom sous lequel CVS sauvegarde les fichiers dans un dépôt,
- **1.5** et **1.6** sont les numeros de révision de **ricard.sql** concernés.

C'est le message d'un conflit, c'est-à-dire que CVS n'a pas pu rassembler les différences tout seul ; vous allez donc devoir le faire à la main.

Pas d'affolement, en général cela n'arrive pas souvent, et quand cela arrive, les corrections sont triviales à effectuer pour un humain. CVS indique à l'intérieur du fichier **ricard.sql** les conflits comme ceci :

```
«««< ricard.sql
bla bla bla ...
=======
bli bli bli ...
»»»> 1.6
```

vous devez alors choisir (ou faire un mélange) entre « bla bla bla » et « bli bli bli ». Losque cela est fait, un **cvs commit ricard.sql** devrait marcher, sauf bien sûr, si quelqu'un d'autre a encore modifié le fichier pendant que vous effectuiez cette modification, auquel cas ... vous êtes bon pour recommencer (je vous rassure, cela n'arrivera quasiment jamais).

NOTE : si, lors d'une mise à jour qui se passe mal, vous voyez que vous êtes en train de modifier la même chose qu'un autre utilisateur, mais dans une voie totalement différente, le mieux est encore de prévenir l'administrateur du projet pour savoir que faire.

3.1 Ajouter et effacer des fichiers

Lorsque vous modifiez un projet dans votre répertoire de travail et que vous faites un **commit**, CVS n'ajoute et ne retire de fichiers **que** si vous le lui avez demandé explicitement (voir section 3.2 et annexe A pour les fichiers *unknown*).

Pour ajouter un fichier :

```
cvs add pothier.C
```

est la commande appropriée, et CVS va vous congratuler par un :

```
cvs add: adding pothier.C
cvs add: use 'cvs commit' to add this file permanently
```

comme indiqué, vous allez poursuivre par un :

```
cvs commit pothier.C
```

Pour retirer un fichier, utilisez la commande **cvs remove fichier** de la même manière. Cependant, il faut que ce fichier soit réellement absent de votre répertoire de travail, sinon CVS va vous envoyer un message de complainte.

RAPPEL : effacer un fichier n'est qu'un opération « virtuelle », il reste toujours un souvenir (succeptible d'être rappelé) d'un fichier qui a été effacé, ainsi que tout son historique de révisions.

3.2 Les préfixes des messages CVS

Lors des opérations sur les fichiers (*add*, *commit*, *update* ...), CVS vous affiche des messages où le nom des fichiers concernés sont précédés d'une lettre en majuscule. En général, vous n'avez pas à vous soucier de tout comprendre mais, pour information, voici les cas les plus courants :

- **U gregi.lev** = Updated (vous venez de récupérer la dernière version du fichier **gregi.lev**),
- **P oldies.f** = Patched (vous venez de mettre à jour la dernière version du fichier **oldies.f**),
- **M gibrat.tex** = Modified (vous avez modifié le fichier **gibrat.tex**),
- **C pierre.perl** = Conflict (le fichier **pierre.perl** contient un conflit),
- **T jeje.ml** = Tagged [6] (vous venez de marquer le fichier **jeje.ml**),
- **? lolo.cc** = Unknown (le fichier **lolo.cc** ne fait pas partie du projet).

3.3 Ajouter et effacer des répertoires

Pour ajouter un répertoire, c'est comme pour ajouter un fichier (une fois qu'il a été créé bien sûr) :

[6] Voir le chapitre 5 pour des détails sur la fonction **tag**.

```
mkdir newdir
cvs add newdir
cvs commit -m "un nouveau repertoire" newdir
```

Effacer un répertoire n'est pas possible !

C'est pourquoi il faut penser un minimum à l'organisation du projet avant de créer des répertoires qui pourraient être gênants par la suite.

Néanmoins, il est possible de retirer tous les fichiers qui sont dans un répertoire et d'utiliser l'option « `-P` » (pour *prune* : tailler, élaguer) lors d'un **checkout** ou d'un **update**, ce qui a pour conséquence d'ignorer les répertoires vides (ils n'apparaîtront pas).

Sinon, il est toujours possible de faire disparaître le répertoire en le retirant directement dans le dépôt, cette manœuvre demande aussi la modifications de fichiers d'administration de CVS. Je **déconseille** cette opération si vous n'êtes pas sûr de ce que vous faites ; il se peut que vous ne puissiez plus accéder à votre projet après une fausse manœuvre.

3.4 Voir ce qui a été fait : *log*, *status* et *diff*

Pour visualiser les révisions d'un fichier, c'est-à-dire : les numéros de révisions, qui les a faites, quand, les commentaires, etc., tapez :

```
cvs log fichier
```

Deux options me semblent utiles : **-d** et **-w**. La première permet de réduire le nombre de révisions à afficher en fonction de la date ; elle doit être suivie d'au moins une date et au plus de deux. Elle utilise les formats suivants de date, dans lesquels des champs peuvent être omis :

1. 1998-08-23 17 :05, ou
2. 23 aug 1998 17 :05.

Les opérateurs de comparaison permis sont : $>$ et $<$ (exclusifs). On peut, paraît-il, utiliser $>=$ et $<=$ (inclusifs). Personnellement, je n'ai pas observé de différences avec les opérateurs $>$ et $<$ ($>=$ et $<=$ sont aussi exclusifs !).

ATTENTION : il ne faut pas oublier de protéger l'expression par des guillemets (voir exemple ci-dessous), sinon elle va être interprétée comme une redirection du *shell* [7].

Par exemple pour connaître les révisions postérieures au 12 janvier 2000 du fichier **debrev.ern** tapez :

```
cvs log -d "> 12 feb 2000" debrev.ern
```

Vous pouvez aussi écrire des commandes comme :

```
cvs log -d "date1 < date2" debrev.ern
```

[7] Si vous ne savez pas ce que c'est, mettez des guillemets, sinon mettez-en aussi.

ce qui correspond à tout ce qui est antérieur à **date2** et postérieur à **date1**, en d'autres termes, toutes les révisions entre la **date1** et la **date2**.

La deuxième option permet de sélectionner les révisions d'un auteur en particulier ; par exemple pour voir les révisions faites par l'utilisateur **bossy**, il suffit de taper :

```
cvs log -wbossy debrev.ern
```
[8]

Pour connaître l'état d'un fichier (voir annexe A) tapez :

```
cvs status -v fichier
```

l'option **-v** ajoute en plus les *tags*, c'est-à-dire les versions du projet pour lesquelles une révision de ce fichier a été sélectionnée.

Pour afficher à l'écran les différences entre une version d'un fichier et une autre, utilisez **cvs diff**. Par exemple, vous voulez savoir quelles ont été les modifications entre la révision 1.2 et 1.5 du fichier **laurent.gg**, tapez :

```
cvs diff -r 1.2 -r 1.5 laurent.gg
```

si vous omettez le second **-r** la différence sera faite avec votre copie de travail actuelle (même si vous l'avez modifiée). La même chose est possible avec une ou des dates, en spécifiant l'option **-D**.

4 Utiliser CVS sur une machine distante

Il est très pratique de pouvoir travailler sur une machine distante pour plusieurs raisons : le dépôt se trouve sur une machine où des sauvegardes sont effectuées régulièrement (je recommande vivement cette option), ou encore, plusieurs personnes distantes travaillent sur un même projet et une machine externe contient le dépôt.

4.1 Utilisation avec rsh

Cela n'est possible que si vous avez un compte sur la machine qui contient le projet (ou si quelqu'un vous donne le droit de se connecter sur son compte), et s'il n'y a pas de *firewall* entre les deux machines ; dans le cas contraire, allez directement à la section 4.3.

Vous êtes l'utilisateur **bossy** sur la machine **mars.lune.br**, et vous voulez accéder au projet **JB007** sur la machine **pain.vin.fr**. Pour cela, vous devez avoir un fichier **.rhosts** sur la machine **pain.vin.fr**, dans votre répertoire de base (*home directory*) ; il suffit que ce fichier contienne la ligne suivante :

```
mars.lune.br bossy
```

vous pouvez tester rsh par la commande suivante :

[8] Notez bien que le nom de l'utilisateur est **collé** au w.

`rsh -l tonio pain.vin.fr ls`

c'est-à-dire que vous vous loguez en tant qu'utilisateur **tonio** sur la machine **pain.vin.fr** pour exécuter la commande **ls**. Si votre nom de *login* est identique sur les deux machines, vous pouvez retirer l'option **-l tonio** (vous vous connecterez en tant qu'utilisateur **bossy**).

Assurez-vous que vous êtes capable de lancer CVS sur la machine distante. Pour cela vous pouvez taper :

`rsh -l tonio pain.vin.fr 'which cvs'`

si vous avez une réponse du genre :

`which: no cvs in (.../.../...)`

où **(.../.../...)** est une liste de répertoires, c'est que vous n'avez pas accès à CVS (contactez l'administrateur du projet, il devrait savoir comment résoudre ce problème), sinon vous verrez apparaître un message comme :

`/usr/bin/cvs`

qui est un bon signe.

Placez-vous dans votre répertoire de travail, et accédez aux commandes CVS en tapant :

`cvs -d :ext:pain.vin.fr:/usr/local/cvs checkout JB007`

si vous avez le même *login*, sinon :

`cvs -d :ext:tonio@pain.vin.fr:/usr/local/cvs checkout JB007`

où **/usr/local/cvs**, est le chemin absolu du dépôt sur la machine distante : **pain.vin.fr**. Cette commande est un peu barbare, mais avec le rappel des commandes (tcsh et bash), ce n'est pas un problème. De plus, une fois que vous avez effectué le *checkout*, et que vous vous trouvez dans le répertoire correspondant, CVS enregistre les informations précédentes (où se trouve le dépôt, etc.), et vous n'avez plus à retaper toute cette ligne, mais seulement : **cvs commande option(s) fichier(s)**.

4.2 Utilisation avec ssh

C'est la même chose qu'avec rsh mais il faut configurer ssh pour ne plus vous demander votre mot de passe (il faut utiliser `ssh-agent`). Pour cela voir : `http://docs.mandragor.org` → operating systems → linux → formation debian → chapitre VI G « Se logguer par ssh sans taper de mot de passe ». Où encore : `http://www.debian.org/devel/passwordlessssh.fr.html`.

N'oubliez pas de mettre ssh dans votre variable d'environnement **CVS_RSH** :

`setenv CVS_RSH ssh`

pour csh et tcsh et :

`export CVS_RSH=ssh`

pour sh, bash, ksh et zsh.

4.3 Utilisation du serveur CVS (pserver)

Pour utiliser ce protocole, il y a deux aspects de la configuration : le côté serveur (la machine sur laquelle se trouve le projet CVS), et le côté client (la machine de laquelle vous voulez accéder au projet).

Soit le serveur : **pain.vin.fr** et le client : **mars.lune.br**.

Occupons-nous d'abord du serveur. Il faut éditer (en tant qu'administrateur de la machine) le fichier **/etc/inetd.conf** pour lui ajouter un service[9] : le **cvspserver**.

Vous devez donc insérer la ligne suivante dans le fichier **inetd.conf** :

```
cvspserver stream tcp nowait root /usr/bin/cvs[10] \
cvs -f --allow-root=/usr/local/cvs pserver
```

NOTE : si vous avez plusieurs dépôts, répétez l'option `--allow-root=`.

Et aussi la ligne suivante dans le fichier **/etc/services** (si ce n'est pas déjà fait) :

```
cvspserver  2401/tcp
```

Vous pouvez normalement relancer **inetd**, sous Linux, par la commande :

```
/etc/rc.d/init.d/inet restart
```

Mais un *bug* existe (dû à l'utilisation d'une variable d'environnement) ; et en général cela ne marche pas, vous avez alors au moins 2 solutions :

1. Relancer la machine.
2. Modifier la ligne :
   ```
   cvspserver stream tcp nowait root /usr/bin/cvs \
   cvs -f --allow-root=/usr/local/cvs pserver
   ```
   par :
   ```
   cvspserver stream tcp nowait root /usr/bin/env \
   env -i cvs -f --allow-root=/usr/local/cvs pserver
   ```
   et relancer **inetd**.

Pour les utilisateurs de **xinetd** (un remplacement plus sécurisé de **inetd** paraît-il) les même problèmes se posent, et peuvent être résolus par l'ajout d'un fichier nommé **cvspserver** dans **/etc/xinetd.d/**, qui contient les entrées suivantes :

```
service cvspserver
{
 flags          = REUSE NAMEINARGS
 socket_type    = stream
 protocol       = tcp
 wait           = no
 user           = root
 log_on_success += HOST USERID EXIT DURATION
```

[9] Inetd est un programme qui déclenche la plupart des services réseaux : telnet, ftp, rsh, etc.

[10] Mettez ici la localisation du programme cvs (tapez : « which cvs » pour le savoir).

```
 log_on_failure += HOST USERID ATTEMPT RECORD
 server         = /usr/bin/env
 server_args    = env -i cvs -f --allow-root=... pserver
 only_from      = pain.vin.fr
}
```

Où « ... » correspond à l'emplacement de votre dépôt, et **only_from** permet ici de restreindre l'accès à une seule machine : **pain.vin.fr**.

Il faut ensuite créer un fichier **$CVSROOT/CVSROOT/passwd** qui contiendra le mot de passe de l'utilisateur encrypté. Comme les mots de passe vont circuler en texte non crypté sur le réseau, il est bon de ne pas mettre de mot de passe correspondant à un utilisateur de la machine.

NOTE : c'est un des rares fichiers à créer SANS passer par CVS, c'est-à-dire à éditer directement dans l'arborescence du dépôt.

Le format de ce fichier est :

```
nom_de_login:mot_de_passe_encrypté:utilisateur_local
```

Quelques explications :

- **nom_de_login** est le nom sous lequel l'utilisateur va se connecter ;
- **mot_de_passe_encrypté** est le mot de passe correspondant au *login* ;
- **utilisateur_local** est le *login* d'un utilisateur local que vous aurez défini (par exemple l'utilisateur **cvs**) et qui donnera ses privilèges à la personne qui se connecte avec le **nom_de_login** correspondant.

Il est possible de mettre un mot de passe crypté dans ce fichier en le recopiant d'un autre endroit (par exemple /etc/passwd). Mais ce n'est pas la solution la plus sûre, car il faut éviter de mettre le même mot de passe qu'un compte utilisateur pour des raisons évidentes de sécurité.

J'ai reçu diverses solutions pour créer des mots de passe par des utilisateurs sympatiques, je ne peux pas les mettres toutes ici, mais je vous propose les plus simples à mon goût.

Pour les heureux possesseurs d'apache :

```
htpasswd -bc passwd nom_de_login mot_de_passe
```

Cela crée un fichier nommé « passwd » avec le *login* et le mot de passe crypté.

Sinon, il est possible de compiler ce petit programme en C (en appelant le fichier « mycrypt.c ») :

```
#include <stdio.h>
#include <unistd.h>
#include <crypt.h>

/*
 * COMPILATION: cc -lcrypt -o mycrypt mycrypt.c
```

```
 * UTILISATION: ./mycrypt
 */

int main() {
  printf("%s\n",crypt(getpass("password:"),"rb"));
  return 0;
}
```

Pour le côté client, vous devrez vous connecter sur le serveur par la commande suivante :
`cvs -d :pserver:bossy@pain.vin.fr:/usr/local/cvs login`
si vous avez comme *login* **bossy**. L'emplacement du dépôt est :**/usr/local/cvs**.

Ensuite, CVS vous demandera votre mot de passe, vous accéderez aux commandes CVS comme ceci :
`cvs -d :pserver:bossy@pain.vin.fr:/usr/local/cvs commande`
par exemple, pour récupérer le fameux projet **JB007**, mettez-vous dans votre répertoire de travail et tapez :
`cvs -d :pserver:bossy@pain.vin.fr:/usr/local/cvs`
`checkout JB007` (sur la même ligne),
… vous connaissez la suite (au travail !).

NOTE : de même qu'avec **rsh**, une fois le *checkout* effectué, vous n'avez plus qu'a taper les commandes CVS normalement.

Pour terminer une session, tapez :
`cvs -d :pserver:bossy@pain.vin.fr:/usr/local/cvs logout`
Cela n'est vraiment utile que si vous ne travaillez plus vraiment sur ce projet.

5 Administrer un projet CVS

Ce chapitre comporte le mot un peu pompeux « administrer », mais ce ne sont que les opérations vitales (et plutôt d'ordre conceptuel) pour quelqu'un qui veut gérer un projet avec CVS, que je vais exposer ici. Je n'entrerai pas dans les fonctions avancées de CVS, comme :

- la structure des données dans le dépôt,
- les problèmes de sécurité avancés (utilisation de GSSAPI ou kerberos),
- les fonction avancées des rassemblements de branches (*magic branch numbers*),
- la gestion des fichiers binaires,
- les fonctions de **cvs watch** (contrôle automatique de certaines actions),
- l'utilisation des mots-clés (*keywords*),
- l'utilisation de vendeurs multiples.

Comme vous pouvez le voir (vous ne comprenez peut-être pas tout), CVS permet de gérer des développements extrêmement complexes ; il est utilisé couramment dans les projets *Free Software* ayant des centaines de développeurs dans le monde entier. Ce n'est pas de tout cela que je veux parler ici, mais des problèmes et tâches courants que devront résoudre et accomplir un administrateur « de base » avec CVS.

Il est temps maintenant de parler des *tags* et des branches.

5.1 Les *tags*

Une chose assez importante pour l'évolution d'un projet est d'apposer des marques (*tags*) aux fichiers d'un projet, lorsque la version vous semble satisfaisante à un moment donné. Cela est réalisé de la manière suivante :

```
cvs tag lerat
```

Ainsi, tous les fichiers du projets auront une marque appelée **lerat** [11] associée à leur numéro de révision actuel.

Pour rappeler le projet à cette version-là, vous avez deux solutions ;

1. vous n'avez aucune version de travail :

   ```
   cvs checkout -r lerat JB007
   ```

2. vous travaillez sur une version de **JB007**, et vous voulez revenir à la version « `lerat` » :

   ```
   cvs update -r lerat
   ```

Il est bien sûr possible de mettre un *tag* sur un seul ou plusieurs fichiers, et non pas tout le projet ; pour cela spécifiez le(s) nom(s) du(des) fichier(s) après :

```
cvs tag nom_de_tag fichier1 fichier2 ...
```

Il est aussi possible de retirer un tag :

```
cvs tag -d nom_de_tag [fichiers]
```

5.2 Les branches

J'ai mis quelque temps avant de me décider à écrire une partie sur les branches, une notion qui m'avait l'air difficile au début, mais qui ne l'est pas tant que ça. Je vais essayer de vous montrer que c'est facile et utile.

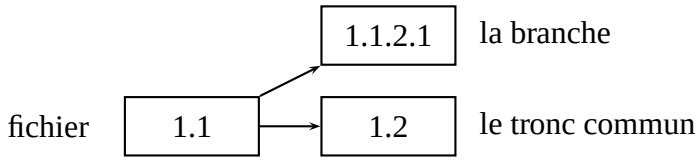
[11] Pour le nom des *tags*, vous pouvez utiliser les caractères suivants : majuscules, minuscules, « - » et « _ », le nom doit commencer par une lettre.

Qu'est-ce que les branches ?

Les branches sont ... des *tags*, avec CVS tout n'est que « *diff* » et « *tags* », mais ne vous souciez pas de comment CVS procède, faites-vous une représentation claire de ce que cela mime.

Une branche peut être vue comme une bifurcation par rapport au développement principal d'un projet, permettant de faire une parenthèse, de tester quelque chose de différent avant de décider de l'inclure dans le développement principal ou non.

Voici comment peut se représenter une branche (c'est trivial) :



Les nombres dans les cases sont les numéros de révision des fichiers (donnés à titre indicatif, mais CVS ne garantit pas cette numérotation).

Le tronc commun est l'état « standard » de développement des fichiers dans votre projet, les modifications sont normalement effectuées à ce niveau-là.

La branche représente une version marquée par l'utilisateur (dont je n'ai pas affiché le *tag*), mais **aussi** par CVS : c'est le numéro de révision `1.1.2.1`. Si vous développez dans une branche, alors vos modifications porteront sur cette branche uniquement et n'affecteront pas le tronc commun.

Pourquoi utiliser les branches ?

Les branches sont utilisées essentiellement pour deux raisons :

1. réparer un *bug* sur une ancienne version de votre programme. Par exemple, un utilisateur vous fait part d'un *bug* sur une ancienne version et vous voulez, dans un premier temps, réparer ce *bug* puis pouvoir répercuter ces modifications sur la nouvelle version.
2. vous voulez créer un remaniement important sur la dernière version du programme sur laquelle vous travaillez, mais cela prendra du temps, et vous n'êtes pas sûr que cela donnera de bons résultats. En tout cas, cela risque de destabiliser le programme pendant un certain temps et vous ne voulez pas appliquer ces modifications avant d'être sûr que cela marche vraiment, ou laisser tomber parce que cela n'a pas donné les résultats escomptés.

Comment utiliser les branches ?

Une branche est créée grâce à l'option « `-b` » (*branch*) de la commande `cvs tag` pour créer une branche sur la version sur laquelle vous travaillez ou `cvs rtag` pour créer une branche sur une version antérieure du programme.

Attention : lors de la création d'une branche, vous ne vous retrouvez pas automatiquement dans cette branche, pour cela il faut y accéder (voir paragraphe suivant).

On accède à une branche grâce à la commande `cvs update` en spécifiant le *tag* de la branche à laquelle on veut accéder, exactement comme pour accéder à une ancienne version d'un projet. Cette commande invoque la révision la plus récente des fichiers dans cette branche.

En général, après un travail dans une branche, il est utile de pouvoir répercuter les modifications faites dans la branche pour le reste du projet. Cela est possible grâce à l'option « `-j` » de la commande `cvs update`. Elle permet de fusionner la version courante avec la version portant le *tag* ou le numero de révision qui suit le « `-j` ».

Voici un exemple de session sous CVS avec création d'une branche, modification du travail dans la branche et répercussion dans le tronc commun.

Nous avons actuellement la dernière version du programme dans notre répertoire de travail, nous voulons créer une branche à l'époque ou nous avions marqué notre projet avec le nom `rel-1-1` (nous en sommes actuellement au *tag* `rel-1-5`).

Créer la branche nommée `rel-1-1-bug0` :

```
cvs rtag -b -r rel-1-1 rel-1-1-bug0
```

Se mettre dans cette branche :

```
cvs update -r rel-1-1-bug0
```

Faire ses modifications, puis :

```
cvs commit -m "mes modifs"
```

Revenir à la version la plus récente :

```
cvs update -A
```
[12]

Appliquer les modifications précédentes :

```
cvs update -j rel-1-1-bug0
```

Et voilà, le tour est joué, il ne reste plus qu'à prier pour qu'il n'y ait pas trop de conflits et vous pourrez alors faire un `cvs commit` de ces modifications, maintenant que vous êtes à nouveau sur le tronc commun.

[12] L'option -A permet de revenir à la version la plus récente, en se débarrassant de tous les *tags* « *sticky* ».

Une autre chose utile est la mise à jour d'une branche avec le tronc principal de développement, pour cela il suffit de faire ceci dans la branche :
`cvs update -j HEAD`

NOTE : il est aussi possible de retirer des modifications passées ; il suffit pour cela d'invoquer la commande « `-j` » 2 fois, par exemple :
`cvs update -j 1.36 -j 1.34 toto.c`
a pour effet de retirer de votre version courantes les modifications faites entre la révision 1.34 et 1.36 de `toto.c`, **attention** de respecter l'ordre inverse ci-dessus (1.36 avant 1.34). Pour utiliser cette stratégie sur plusieurs fichiers, il suffit d'utiliser des *tags* à la place des numéros de révision.

5.3 Derniers conseils

Je terminerai par quelques conseils, qui sont de l'ordre du bon sens, mais que l'on peut vite oublier. Gérer un projet c'est d'abord **communiquer**, c'est-à-dire qu'avant de mettre un projet sur pied, il est bon de bien penser à la structure qu'il aura dans le dépôt pour éviter de nombreux remaniements (qui peuvent s'avérer problématiques). Il est aussi important que les différents utilisateurs (dans la plupart des cas) sachent sur quelle(s) partie(s) du projet ils vont travailler pour éviter trop de conflits (même si cela n'est en fait pas si courant) et, bien sûr, recommander l'écriture de commentaires clairs et précis de ce qui a été fait.

Bon courage et bons projets avec CVS.

Remerciements

Je remercie vivement ma sœur Charlotte pour ses relectures qui ont apporté à ce document une lisibilité vitale ; tous les amis et amies qui ont lu ce document pour m'en indiquer les points obscurs et, bien sûr, les utilisateurs qui ont daignés m'écrire pour m'envoyer leurs astuces et remarques qui ont aidé à l'amélioration de ce document.

Merci !

A Les états des fichiers

Up-to-date : le fichier est identique à la dernière version du dépôt.
Locally Modified : le fichier a été modifié localement mais non soumis.
Locally Added : le fichier a été ajouté localement mais non soumis.

Locally Removed : le fichier a été effacé localement mais non soumis.
Needs Checkout : le fichier a été modifié par quelqu'un au dépôt (mais vous n'avez pas modifié ce fichier localement) ; malgré ce message, c'est un **update** que vous devez faire (et non pas un **checkout**).
Needs Patch : presque identique au message précédent (seule une partie du fichier sera échangée).
Needs Merge : vous avez modifié le fichier localement **et** quelqu'un l'a modifié au dépôt ; une mise à jour s'impose.
File had conflicts on merge : le fichier contient toujours un conflit (que vous n'avez pas encore résolu).
Unknown : CVS ne sait rien de ce fichier, comme un fichier temporaire que vous auriez créé.

Références

[1] C. *et al*, *Version Managment with CVS for CVS 1.10*.