SubVersion How To

Michael Jégat

Incubateur ESTIA
64210 Bidart
Tél. 05.59.43.85.21
www.corexpert.net

RÉSUMÉ : *présente l'utilisation de SubVersion ainsi que les outils associés.*

MOTS-CLÉS : *subversion, collaboratif, version, tag, branche, tortoise*

Table des matières

1 Le contrôle des versions : comment ca marche ?

La gestion des versions (en anglais : revision control) est une activité qui consiste à maintenir l'ensemble des versions d'une unité d'information. Elle est principalement utilisée dans les domaines de l'ingénierie et du développement de logiciels pour gérer les évolutions des documents informatiques, comme les codes sources, les dessins ou autres informations sur lesquelles travaille une équipe de personnes.

La gestion des versions se base donc sur le suivi des modifications apportées aux fichiers. Ainsi, imaginons *Clara* qui travaille seule sur une arborescence de fichiers et qui souhaite en faire la gestion des versions. Voici comment elle pourrait procéder :
- dans son arborescence, elle rajoute à la fin de chaque nom de fichier le numéro 1, correspondant à la première version,
- ensuite, elle va modifier un fichier,
- enfin, lors de la sauvegarde de ses modifications, *Clara* ne va pas écraser l'ancien fichier, mais va plutôt l'enregistrer sous le même nom avec un nouveau numéro de version. Ce numéro pourra simplement être incrémenté.

En suivant ce principe, *Clara* possèdera tout l'historique des modifications apportées à ses fichiers et sera donc capable de revenir en arrière. Elle pourra aussi, grâce à certains logiciels spécialisés (windiff, kdiff3), visualiser les différences entre les versions des fichiers.

Dans le cadre d'un travail en équipe, une solution serait de placer l'ensemble des fichiers sur un serveur accessible à toute l'équipe. Ensuite, lorsqu'une personne souhaite travailler sur un fichier, elle le récupère, l'édite et le sauvegarde sous le même nom en modifiant juste le numéro de version.

Il faut cependant faire attention aux accès concurrents sur les fichiers : lorsque deux personnes vont éditer le même fichier. Imaginons *Clara* et *Morgane* travaillant sur la même version d'un fichier. *Clara* vient de terminer et enregistre normalement ses modifications. Ensuite, *Morgane* termine son travail et souhaite sauvegarder le fichier. Là, elle va s'apercevoir qu'un fichier portant déjà le même numéro de version existe déjà (créé par *Clara*). Dans ce cas là, *Morgane* va devoir fusionner les modifications qu'elle a apporté avec celles effectuées par *Clara* . Lorsque cette fusion sera achevée, elle sauvegardera le fichier avec un numéro de version incrémenté une nouvelle fois.

Ces pratiques sont faisables et certaines entreprises travaillent comme cela. Cependant, ces méthodes sont très fastidieuses et relativement complexes, et demandent beaucoup de discipline. Ainsi un appui logiciel semble indispensable et c'est pourquoi il en existe un certain nombre.

Nous allons tout d'abord présenter les concepts essentiels de ces logiciels avant d'en présenter une étude comparative.

1.1 Fonctionnement

La plupart des logiciels utilisent l'approche énoncée ci-dessus :
- les données sont centralisées sur un serveur
- les utilisateurs récupèrent une copie en local pour effectuer des modifications
- les utilisateurs valident les changements en renvoyant leurs modifications au serveur
- toutes les modifications sont mémorisées par le serveur

Cette approche est dite "centralisée" car tous les fichiers sont situés sur un même site. Il existe aussi une technique décentralisée, ou distribuée, où chaque copie de travail représente un dépôt complet, avec tout l'historique, dans lequel on peut enregistrer et grâce auquel on peut distribuer les modifications.

Les deux méthodes fonctionnent très bien mais les logiciels centralisés sont plus fréquents. Un système décentralisé sera conseillé dans le cas où une connexion permanente au serveur n'est pas possible.

Dans la suite de ce document, nous nous intéresserons plus particulièrement à l'architecture centralisée car Subversion utilise ce modèle.

1.1.1 Le dépôt

Le dépôt est un lieu central de sauvegarde des données, qui contient plus précisément une arborescence de fichiers et de répertoires. Un client peut se connecter au dépôt pour lire ou écrire ces fichiers :

– en écrivant des données, le client rend les informations disponibles aux autres utilisateurs,
– en lisant les données, le client reçoit les informations des autres.

Un tel dépôt est semblable à un serveur de fichier standard, mais possède l'avantage de mémoriser les modifications apportées aux données.

Ainsi, lorsqu'un client lit les données, il récupère normalement la dernière version disponible. Mais il a la possibilité de voir les états précédents des données.

Ces systèmes sont conçus pour mémoriser et suivre les changements des données sur le temps.

1.1.2 Le travail collaboratif

Une des missions principales des outils de gestion des versions est de permettre une édition collaborative et un partage des données. Cependant, tous ces systèmes se confrontent au problème suivant : comment le système va-t-il permettre aux utilisateurs de partager l'information sans se marcher sur les pieds ? En d'autres termes, comment le système va-t-il gérer les conflits ?

Rappelons qu'un conflit apparaît lorsque deux personnes travaillent sur les mêmes données. Voici un exemple concret :

1. *Clara* et *Morgane* lisent le même fichier
2. *Clara* et *Morgane* effectuent des modifications sur leur copie locale du fichier
3. *Clara* sauvegarde en premier ses données sur le dépôt
4. *Morgane* écrase la version de *Clara* lorsqu'elle effectue sa sauvegarde.

Généralement, deux méthodes existent pour gérer les conflits :
– *on les évite :* lorsqu'une personne travaille sur un fichier, celui-ci est verrouillé et personne d'autre ne peut l'éditer. Cette technique permet d'éviter les conflits mais apporte d'autres problèmes, notamment dans le cas où la personne garde trop longtemps le fichier.
– *gestion semi-automatique :* le logiciel essaie de fusionner automatiquement les modifications. Cette fusion automatique fonctionne généralement bien lorsque les modifications ont lieu dans des endroits éloignés du fichier. S'il n'y arrive pas, il demande à l'utilisateur de le faire. Cette approche est généralement celle utilisée par les logiciels.

En résumé, la gestion des conflits est un problème inévitable mais qui peut être réduit par :
– une bonne communication au sein de l'équipe : avertir les autres personnes lorsque l'on modifie un fichier ou lorsqu'on sauvegarde,
– éclater au maximum les fichiers : plus les données seront disposées dans des fichiers différents, plus le nombre de conflits va diminuer,
– une bonne répartition des tâches : définir la répartition des données dans les fichiers en fonction des tâches de chacun.

1.1.3 Les révisions

Comme nous venons de le voir, la plupart des logiciels de gestion des versions mémorisent l'ensemble des modifications apportées à chaque donnée. Pour pouvoir suivre ces modifications dans le temps, la notion de version, ou plutôt de révision, apparaît. En d'autres termes, une révision correspond à une modification, une modification pouvant correspondre à un ajout, une modification, une suppression ou une combinaison des trois sur une version donnée.

Schématiquement, on passera de la révision N à la révision N+1 en appliquant la modification M. Pour revenir en arrière, un logiciel de gestion de versions nous aidera alors à soustraire la modification M à la révision N+1 pour retrouver N.

Les révisions peuvent être gérées au niveau des fichiers ou de façon atomique. Plus concrètement, il y a deux approches possibles :
– révisions associées aux fichiers : un fichier possède plusieurs révisions. Il passe d'une révision à la suivante par une modification.
– révisions associées à un projet (un ensemble de fichiers) : le projet possède plusieurs révisions. Il passe d'une révision à la suivante par des modifications sur certains de ses fichiers.

Ces deux approches ont des conséquences totalement différentes comme le montre l'exemple suivant :
– imaginons un projet composé de 3 fichiers $f1$, $f2$ et $f3$

– appliquons sur ce projet 2 ensembles de modifications :
  tout d'abord on modifie le fichier $f2$
  ensuite on change les fichiers $f1$ et $f2$ à nouveau.
– maintenant, prenons chacune des méthodes de gestion possible :
  révisions associées aux fichiers : au départ, chaque fichier possède la révision $r1$. A la fin de la première validation, les fichiers $f1$ et $f3$ sont toujours dans la révision $r1$ mais le fichier $f2$ est passé en révision $r2$. Ensuite, après la deuxième validation, le fichier $f3$ est toujours en révision $r1$, tandis que le fichier $f1$ est passé en révision $r2$ et le fichier $f2$ est en révision $r3$. Dans un tel exemple, on va dire que le fichier $f2$ passe de la révision $r1$ à $r2$ en effectuant telle modification, et de la révision $r2$ à $r3$ en ajoutant telle modification.
  révisions associées à un projet : dans ce cas, au départ, le projet est en révision $r1$, sachant que les fichiers n'ont pas de révision. Ensuite, après la première validation, le projet passe en révision $r2$. Enfin, après la deuxième validation, le projet sera en révision $r3$. Comme on le voit, c'est beaucoup plus simple, et on va dire que pour passer de la première révision à la seconde il faut effectuer telle modification sur $f2$, et pour passer de $r2$ à $r3$, il faut modifier $f1$ et $f2$. Cette méthode paraît plus naturelle. Cette méthode est utilisée dans Subversion.

1.1.4 La gestion des modifications

Afin d'économiser de l'espace mémoire, la plupart des logiciels de gestion des versions utilisent des algorithmes complexes pour sauvegarder l'ensemble des versions des fichiers. Généralement, la méthode utilisée se base sur des algorithmes différentiels qui ne mémorisent réellement que la première version du fichier et sauvegardent ensuite les changements.

1.2 Etude comparative

Le tableau 1 montre les caractéristiques des logiciels de contrôle des versions les plus répandus.

nom	cible	architecture	révisions	licence	tarif
CVS	tout	centralisé	fichier	GPL	gratuit
Subversion	tout	centralisé	ensemble	Apache/BSD	gratuit
Darcs	tout	décentralisé	ensemble	GPL	gratuit
Clearcase	gros projets	centralisé	fichier	propriétaire	4000 euros
Visual SourceSafe	petits projets	centralisé	fichier	propriétaire	500 euros
Perforce	gros projets	centralisé	fichier	propriétaire	utilisation commercialle : 750 euros/licence utilisation opensource : gratuit utilisation éducation : gratuit + support
BitKeeper	gros projets	décentralisé	fichier	propriétaire	environ 1000 euros
Arch	tout	décentralisé	ensemble	GPL	gratuit
Git	tout	décentralisé	fichier	GPL	gratuit

TAB. 1 – Avantages et inconvénients

Notons que pour des logiciels comme CVS ou Subversion, des outils externes leur permettant de développer une architecture décentralisée existent.

1.2.1 CVS

CVS (Concurrent Versions System) est un logiciel libre (licence GPL) de gestion de versions. Il peut fonctionner en local ou en mode client/serveur. Dans tous les cas, le dépôt contient un ou plusieurs modules, chaque module étant constitué d'une hierarchie de fichiers versionnés.

CVS existe depuis 1989, c'est un logiciel très utilisé et bien éprouvé. De nombreux outils et environnements de développement l'utilisent. Malheureusement, CVS souffre de quelques lourdeurs conceptuelles. Par exemple, on ne peut pas renommer un répertoire, il faut supprimer et recréer tout son contenu. Le logiciel Subversion a été conçu comme un clone de CVS sans ses lourdeurs.

CVS peut être utilisé pour n'importe quel type de projets. Cependant, pour débuter un nouveau projet, nous préférerons Subversion qui fait la même chose en mieux.

1.2.2 Darcs

Darcs (David's Advanced Revision Control System) est un système de gestion de versions décentralisé conçu par David Roundy et dans l'objectif de remplacer CVS. Plusieurs différences existent dans le fonctionnement. En effet, chaque copie du dépôt agit comme un véritable dépôt, permettant de gérer plusieurs versions dans des endroits différents.

Lors d'une utilisation simple, l'utilisateur récupère une copie du dépôt, effectue des changements, enregistre les changements, récupère les changements effectués sur les autres dépôts, et met à disposition ses propres changements.

Les autres dépôts peuvent être accédés via les protocoles SSH ou HTTP.

Un autre aspect de Darcs est qu'il est écrit avec le langage Haskell.

1.2.3 Clearcase

ClearCase est un logiciel de gestion des versions commercialisé par Rational (IBM). Il peut fonctionner en local ou en mode client/serveur. Il est principalement utilisé dans les moyennes et grandes entreprises et peut gérer des projets sur lesquels travaillent des milliers de développeurs. Cependant, son coût est trop élevé pour des petites sociétés.

ClearCase est bien entendu compatible avec les autres produits de Rational (Rose...).

1.2.4 Visual SourceSafe

SourceSafe est un outil de gestion de versions commercialisé par Microsoft. Ce logiciel est plutôt destiné aux petits projets. Son fonctionnement est semblable aux autres outils centralisés : une copie en local doit être récupérée, puis l'utilisateur effectue des modifications avant de propager son travail.

Son principal avantage est d'être parfaitement intégré dans Visual Studio. Cependant, il souffre d'un certain manque de stabilité.

1.2.5 Perforce

Perforce est un autre outil de gestion de versions commercialisé par la société Perforce. Il utilise le protocole SCCS de Microsoft et est donc compatible avec tous les sytèmes l'utilisant (Visual Studio...). Il propose une gestion des modifications par liste de changements afin de rendre les modifications unitaires.

Chose intéressante, il permet aussi la création de tâches à accomplir sur le produit, et offre à l'utilisateur l'association de ces tâches avec les listes de changements.

Un connecteur ODBC permet d'utiliser des outils de métrique logicielle ou autres outils complémentaires de gestion de code. Enfin, il intégre la gestion de triggers permettant de déclencher des actions sur mise à jour de fichiers dans Perforce.

1.2.6 Bitkeeper

BitKeeper est un logiciel de gestion de version destiné au code source. Conçu comme un système distribué sophistiqué, BitKeeper se positionne comme un logiciel comparable à des systèmes professionnels tels que ClearCase ou Perforce. BitKeeper est produit par BitMover Inc., une compagnie privée basée à San Francisco en Californie, détenue par son PDG Larry McVoy (il est également à l'origine de TeamWare).

BitKeeper reprend plusieurs concepts de TeamWare. La fonctionnalité mise en avant est la facilité avec laquelle les équipes de développement peuvent disposer d'un dépôt des sources local, tout en travaillant avec un dépôt centralisé.

BitKeeper est un logiciel propriétaire (ses sources ne sont pas ouvertes au public) et est en principe vendu ou loué (comme composant dans une offre de support plus large) à des grandes ou moyennes entreprises. Le prix de la licence par développeur varie selon le client, mais il est estimé à plus de mille euros.

1.2.7 Arch

Arch est un système gratuit de gestion des versions. Il repose sur une architecture répartie. Il propose une approche basée sur les ensembles de modifications (tout comme Subversion). Il travaille aussi sur les arborescences et non fichier par fichier.

Il peut être comparé à Subversion, mais en version distribuée.

1.2.8 Git

Git est un système de fichiers versionné réparti proposé par Linus Torvalds pour gérer le développement du noyau Linux. C'est un logiciel gratuit et libre, distribué sous la licence GPL. Il fonctionne à bas niveau, ce qui limite son utilisation à des développeurs très techniques. Cependant, des "front ends" existent, comme par exemple Cogito. Il est aussi destiné à fonctionner sous Linux et propose une faible compatibilité avec les autres systèmes.

C'est un outil très récent, dont le développement a démarré début Avril 2005. Son objectif n'est cependant pas de devenir un outil classique de gestion de versions, comme CVS ou Perforce.

1.2.9 Conclusion

En conclusion, dans le domaine professionnel, nous retrouvons principalement les logiciels commerciaux ClearCase, Perforce, BitKeeper et SourceSafe. Les trois premiers sont plutôt destinés aux grandes entreprises tandis que SourceSafe est plutôt utilisé dans les petites sociétés.

Concernant les outils libres, CVS occupe une grande place et est utilisé par de nombreux projets (sourceforge, freshmeat...). Cependant, son avenir est devenu incertain avec l'arrivée de Subversion qui offre des fonctionnalités plus poussées. D'ailleurs, des projets comme KDE ont déjà basculé de CVS vers Subversion.

Les systèmes Perforce, ClearCase, BitKeeper, Subversion et Arch semblent les plus performants, mais le choix final d'un système de gestion de version devra s'effectuer en fonction de la taille du projet, de l'accès aux données (centralisé, réparti) et du budget alloué.

2 Présentation de Subversion (SVN)

Subversion est un outil de gestion des versions Opensource, distribué sous licence Apache/BSD. Il a été créé pour pallier les manques de CVS et conçu pour le remplacer.

Il s'appuie délibérément sur les mêmes concepts et considère que le modèle de CVS est le bon, seule son implémentation est en cause.

Du point de vue des auteurs de Subversion, les manques les plus criants de CVS sont :
- les "commits", ou enregistrement des modifications, ne sont pas atomiques,
- CVS ne connaît pas le renommage des fichiers : si on change le nom d'un fichier, on perd tout l'historique associé,
- les méta-données ne sont pas versionnées : par exemple, on ne peut pas attacher de propriétés (comme les permissions) à un fichier.

Du point de vue du simple utilisateur, les principaux changements lors du passage à Subversion, sont :
- les numéros de révision sont désormais par révision et non plus par fichier : chaque patch a un numéro de révision unique, quel que soient les fichiers touchés,
- "svn rename" permet de renommer un fichier,
- les répertoires et méta-données sont versionnés,
- subversion ne fait aucune distinction entre un label, une branche et un répertoire. C'est une simple convention de nommage entre ses utilisateurs. Aussi, il devient très facile de comparer un label et une branche ou autre croisement.

2.1 Fonctionnement

Subversion permet de gérer des fichiers et des répertoires à travers le temps. Une arborescence de fichiers est placée dans un dépôt central. Ce dépôt est semblable à un serveur de fichier, excepté qu'il mémorise toutes les modifications effectuées sur les fichiers et les répertoires. Ceci vous permet de récupérer facilement les anciennes versions de vos données, ou bien d'examiner l'historique des modifications pour voir comment vos données changent.

2.2 Fonctionnalités

Voici la liste des principales fonctionnalités proposées par Subversion :
- *les fonctionnalités de CVS :* la plupart des fonctionnalités de CVS sont intégrées dans Subversion. De plus, lorsque c'était possible, l'interface de Subversion a été calquée sur celle de CVS.

- *les répertoires, renommages et les propriétés des fichiers sont versionnés :* l'historique des versions ne se fait pas uniquement sur le contenu des fichiers comme le fait CVS. Cet historique se fait également sur les répertoires, les copies et les renommages. Les propriétés (méta-données) des fichiers sont aussi versionnées.

- *les commits sont atomiques :* lorsqu'un commit est fait sur un ensemble de ressources, il faut que la globalité du commit se termine correctement pour que celui-ci soit valide. Les numéros de révision ne sont plus sur les fichiers individuellement mais sur le commit lui-même.

- *Apache comme serveur réseau, WebDAV/DeltaV comme protocole :* Subversion utilise le protocole WebDAV/DeltaV basé sur HTTP ainsi que le serveur Web Apache. Grâce à cela, Subversion tire avantage de ceux-ci pour l'authentification, l'autorisation d'accès basique, la compression à la volée ou le parcourt du référentiel. Pour les personnes désirant utiliser un tunnel ssh, il existe également une version standalone de Subversion.

- *gestion des branches et des tags simplifiée :* les branches et les tags sont assimilés à des opérations de "copie". Comme la création de copies, les branches et les tags ne sont que des références à des révisions dans le référentiel et nécessitent donc qu'un faible espace sur le disque.

- *nativement client/serveur, architecture logicielle réfléchie :* Subversion a été pensé dès le début en tant que client/serveur, évitant ainsi les problèmes de maintenance qu'a pu rencontrer CVS. Le code est structuré sous forme de modules dont les points d'entrée sont bien définis ce qui permet de les intégrer facilement dans des applications tierces.

– *le protocole envoie le différentiel dans les 2 directions :* le protocole réseau n'envoie que les parties différentes des ressources entre les versions du client et du serveur (CVS envoie les différences uniquement du serveur vers le client, mais pas du client vers le serveur).

– *les coûts sont proportionnels à la taille des changements et non des données :* en général, le temps nécessaire à une opération Subversion est proportionnel à la taille des changements qui résultent de l'opération et non de la taille du projet dans lequel les changements interviennent. C'est une des propriétés du modèle de référentiel de Subversion.

– *manipulation efficace des fichiers binaires :* Subversion est aussi efficace avec les fichiers binaires qu'avec les fichiers texte, car il utilise un algorithme différentiel pour transmettre et stocker les révisions successives.

– *sorties interprétables :* toutes les sorties en ligne de commandes du client Subversion sont humainement compréhensibles et interprétables automatiquement par un programme ; l'utilisation de Subversion avec des scripts est une priorité importante.

2.3 Installation

2.3.1 Procédure

Subversion est conçu sur une couche portable appelée APR (Apache Portable Runtime Library). Ceci signifie que Subversion peut marcher sur n'importe quel système d'exploitation sur lequel fonctionne le serveur Apache (httpd) : Windows, Linux, BSD, Mac OS X et autres.

La façon la plus facile d'installer Subversion est d'utiliser les paquetages binaires spécifiques au système souhaité. Ils sont téléchargeables gratuitement sur le site de Subversion (`http://subversion.tigris.org`). Ce site contient aussi des installeurs graphiques pour les utilisateurs de Microsoft. Si vous utilisez Linux, vous pouvez utiliser les systèmes de paquetages spécifiques, comme les RMPs, DEBs...

Vous pouvez aussi compiler Subversion directement depuis ses codes sources. Pour cela, il faut télécharger les sources et suivre la procédure indiquée dans le fichier INSTALL.

2.3.2 Composants

Une fois Subversion installé, vous aurez à votre disposition les composants suivants :
– *svn :* le programme client en ligne de commande
– *svnversion :* un programme pour récupérer l'état (en terme de révision) de la copie de travail
– *svnlook :* un outil pour inspecter un dépôt Subversion
– *svnadmin :* un outil pour créer ou réparer un dépôt
– *svndumpfilter :* un programme pour filtrer les flux de déchets d'un dépôt
– *mod_dav_svn :* un plug-in pour le serveur HTTP Apache, utilisé pour rendre un dépôt disponible aux autres à travers un réseau
– *svnserve :* un programme autonome, exécutable comme un démon ou par un appel SSH. Il offre une autre manière de rendre le dépôt accessible depuis un réseau

2.3.3 Serveur par Svnserve

Cette application sert de serveur Subversion, accessible via le port 3690. Du côté client, l'utilisateur devra utiliser les paramètres *svn ://* ou *svn+ssh ://* pour accéder au dépôt.

Vous avez trois possibilités pour faire tourner ce serveur :
– à l'aide d'**inetd**
– sous forme de démon
– tunnel ssh

Svnserve par Inetd

Si votre système utilise **Inetd**, vous pouvez ajouter cette ligne au fichier */etc/inetd.conf* :

```
svn stream tcp nowait svnowner /usr/bin/svnserve svnserve -i
```

Attention, vous devez être certain que le "svnowner" possède les droits appropriés pour accéder au dépôt. Maintenant, lorsqu'une connexion client arrive au serveur sur le port 3690, **inetd** créera un serveur **svnserve** pour la gérer.

Le dépôt sera accédé en utilisant l'option *svn ://*.

Svnserve sous forme de démon

Une seconde option pour faire marcher **svnserve** est de l'exécuter sous forme de démon. Pour cela, il faut utiliser l'option $-d$ comme suit :

```
$ svnserve -d
$   svnserve is now running, listening on port 3690
```

Vous pouvez aussi personnaliser le démon à l'aide des options *–listen-port* et *–listen-host*.

Le dépôt sera accédé en utilisant l'option *svn ://*.

Svnserve et le tunnel SSH

Vous pouvez utiliser **svnserve** dans un mode "tunnel", avec l'option *-t*. Ce mode assume qu'un programme tel SSH a authentifié avec succès un utilisateur et invoque maintenant un processus privé **svnserve** pour l'utilisateur. Le serveur fonctionne normalement et assume que le traffic est automatiquement redirigé vers le client par un tunnel.

Attention, dans ce cas, vous devez être certain que l'utilisateur en question possède les droits appropriés pour accéder au dépôt. En fait, c'est comme si l'utilisateur accédait au dépôt de façon locale (*file :///*).

Le dépôt sera accédé en utilisant l'option *svn+ssh ://*.

Authentification automatique : à chaque fois que vous allez effectuer une requête au serveur, **ssh** va vous demander votre mot de passe. Voici la démarche à suivre pour mémoriser votre mot de passe et donc pour éviter cette lourdeur :

– en local, générer une paire de clé (publique/privée) en utilisant RSA :

```
[mike] $ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/mike/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/mike/.ssh/id_rsa.
Your public key has been saved in /home/mike/.ssh/id_rsa.pub.
The key fingerprint is:
95:eb:01:ca:67:7e:ab:ad:ac:b3:ee:c9:4e:69:ab:c3 mike@mike-pc
```

– copier la clé publique sur le serveur, dans notre répertoire personnel :

```
[mike] $ scp ~/.ssh/id_rsa.pub 192.168.0.1:/home/mike/.ssh
Password:
id_rsa.pub                                    100%  222     0.2KB/s   00:00
```

– copier la clé publique à la fin du fichier *authorized_keys* sur le serveur :

```
[mike] $ ssh mike@192.168.0.1
Password:

mike@suzuki:~$ cd /home/mike/.ssh/
```

```
mike@suzuki:~/.ssh$ cat id_rsa.pub >> authorized_keys
mike@suzuki:~/.ssh$ exit
logout
Connection to 192.168.0.1 closed.
```

– maintenant, on peut tester que **ssh** ne demande plus notre mot de passe mais la phrase saisie plus tôt :

```
[mike] $ ssh mike@192.168.0.1
Enter passphrase for key '/home/mike/.ssh/id_rsa':
mike@suzuki:~$ exit
logout
Connection to 192.168.0.1 closed.
```

– la clé publique marche, on s'occupe maintenant de la clé privée. Pour cela, on fait :

```
[mike] $ ssh-add ~/.ssh/id_rsa
Enter passphrase for /home/mike/.ssh/id_rsa:
Identity added: /home/mike/.ssh/id_rsa (/home/mike/.ssh/id_rsa)
```

– ensuite, on installe **keychain** :

```
[mike] $ sudo apt-get install keychain
```

– enfin, on ajoute ces lignes au fichier *bash_profile* (ou autre en fonction du shell) :

```
`eval ssh-agent`
/usr/bin/keychain /home/mike/.ssh/id_rsa
```

– et puis on recharge le fichier :

```
[mike] $ . ~/.bash_profile
KeyChain 2.5.5; http://www.gentoo.org/proj/en/keychain/
Copyright 2002-2004 Gentoo Foundation; Distributed under the GPL

 * Found existing ssh-agent (7530)
 * Known ssh key: /home/mike/.ssh/id_rsa
```

– et voilà, on peut maintenant tester la connexion **ssh** au serveur et on s'aperçoit que le mot de passe n'est pas demandé.

2.3.4 Serveur par Apache

A l'aide d'un module particulier, le serveur **httpd** peut rendre un dépôt Subversion accessible depuis les clients via le protocole WebDAV/DeltaV. Cette méthode est intéressante car elle utilise un outil standard et robuste (Apache 2). De plus, elle évite à l'administrateur système d'être obligé d'ouvrir un nouveau port sur le réseau.

Cependant, l'installation de Subversion utilisant Apache est plus complexe que par Svnserve.

Installation

Tout d'abord, voici les étapes de l'installation :

– vous devez avoir le serveur **httpd 2.0** qui fonctionne et qui utilise le module *mod_dav*
– vous devez installer le module *mod_dav_svn*, qui permettra d'accéder à la librairie Subversion
– vous devez configurer le fichier *httpd.conf* afin de lui indiquer les dépôts à rendre disponibles

Les deux premières étapes peuvent être réalisées en compilant les sources ou bien en installant les paquetages binaires correspondants. Pour vérifier que vous possédez bien ces fichiers d'installés, vous pouvez regarder si vous possédez bien les fichiers suivants :

– */usr/lib/apache2/modules/mod_dav.so*
– */usr/lib/apache2/modules/mod_dav_svn.so*
– */usr/lib/apache2/modules/mod_authz_svn.so* (sert à définir finement les permissions d'accès à certaines parties du référentiel Subversion.)

Vous devez aussi avoir l'exécutable *apache2* d'installé.

Ensuite, vous devez configurer le fichier */etc/apache2/httpd.conf*, en rajoutant les lignes suivantes si elles n'y sont pas :

```
LoadModule dav_module     modules/mod_dav.so
LoadModule dav_svn_module modules/mod_dav_svn.so
```

Après, vous devez définir l'emplacement de votre dépôt. Pour cela, vous allez rajouter le bloc suivant :

```
<Location /nom>
  DAV svn
  SVNPath /chemin/du/depot
</Location>
```

A ce moment là, vous pouvez relancer Apache 2 et votre dépôt sera accessible, via l'option *http ://*, pour n'importe quel client, et ce, de façon anonyme (sans authentification).

Authentification par Http

La façon la plus facile pour authentifier les clients est d'utiliser le mécanisme de base de Http. Cette méthode demande simplement un nom d'utilisateur et un mot de passe à l'utilisateur, et vérifie si ces données sont bien correctes.

Apache fournit un utilitaire (**htpasswd**) qui permet de gérer la liste des mots de passe acceptables.

Par exemple, attribuons les droits d'accès à *Clara* . Nous devons tout d'abord lui attribuer un mot de passe. Pour cela, on fait :

```
$ htpasswd -cm /etc/svn-auth-file clara
New password: *****
Re-type new password: *****
Adding password for user clara
```

Ensuite, vous devez modifier le bloc "Location" du fichier */etc/apache2/httpd.conf*. On va y rajouter les lignes suivantes :
- *AuthType Basic :* indique qu'il faut utiliser un mode d'authentification normal.
- *AuthName "notre dépôt Subversion" :* nom donné au domaine. Il sera affiché dans la fenêtre de saisie du client.
- *AuthUserFile /etc/svn-auth-file :* indique l'emplacement du fichier contenant les mots de passe.
- *Require valid-user :* indique que l'utilisateur doit avoir réussit l'authentification pour être accepté.

Ainsi, le bloc "Location" devient :

```
<Location /nom>
  DAV svn
  SVNPath /chemin/du/depot
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /etc/svn-auth-file
  Require valid-user
</Location>
```

Sachant que cette méthode transmet en clair les mots de passe, elle n'est pas très sécurisée. Pour obtenir une authentification plus sécurisée, vous devez utiliser le cryptage SSL.

2.4 Conversion d'un dépôt CVS

Le programme **cvs2svn** permet de convertir un dépôt CVS vers un dépôt Subversion. Cette conversion va prendre en compte tous les changements qui ont été effectués sur le dépôt CVS pour les copier vers le nouveau dépot. Ainsi, tout l'historique est aussi converti.

Voici la syntaxe du programme :

cvs2svn *-s <chemin/vers/subversion> </chemin/vers/cvs>*

Cette commande créera un nouveau dépôt Subversion. Pour utiliser un dépôt existant, vous pouvez utiliser l'option *–existing-svnrepos*.

Par défaut, cette conversion va créer à la racine du dépôt SVN trois répertoires :
– trunk
– branches
– tags
Vous pouvez modifier ces noms avec les options *–trunk=<chemin>*, *–branches=<chemin>*, *–tags=<chemin>*.

3 Prise en main de Subversion

Dans cette section, nous allons présenter les commandes de base de Subversion. Nous allons donc apprendre à créer un dépôt, importer des projets, en récupérer, appliquer des modifications et visualiser les modifications. Toutes ces opérations seront effectuées dans un environnement monoposte.

3.1 Création du dépôt

Tout d'abord, nous allons créer le dépôt. Pour cela, nous allons utiliser l'outil *svnadmin* en lui spécifiant le répertoire où sera situé le dépôt.

Voici la commande :

```
$ svnadmin create /home/clara/depot
$ ls -l /home/clara/depot
total 28
drwxr-xr-x  2 clara clara 4096 2005-12-27 15:49 conf
drwxr-xr-x  2 clara clara 4096 2005-12-27 15:49 dav
drwxr-sr-x  5 clara clara 4096 2005-12-27 15:49 db
-r--r--r--  1 clara clara    2 2005-12-27 15:49 format
drwxr-xr-x  2 clara clara 4096 2005-12-27 15:49 hooks
drwxr-xr-x  2 clara clara 4096 2005-12-27 15:49 locks
-rw-r--r--  1 clara clara  379 2005-12-27 15:49 README.txt
```

Cette commande va créer le répertoire */home/clara/depot* qui contiendra le dépôt Subversion. Ce nouveau répertoire contient plusieurs fichiers concernant la base de données Berkeley DB.

Remarque : il est fortement déconseillé d'éditer ces fichiers. Pour rajouter, modifier ou supprimer des fichiers sur le dépôt, vous devez utiliser les commandes appropriées.

3.2 Importation d'un projet

Maintenant que le dépôt est créé, nous allons pouvoir importer un nouveau projet. Pour cela, nous allons imaginer *Clara* qui souhaite importer un projet *p*1 contenant un répertoire *A* contenant lui-même deux fichiers *a* et *b*.

Voici la commande qu'elle va utiliser :

```
$ svn import /home/clara/tmp/p1 file:///home/clara/depot -m "import initial"
Ajout          /home/clara/tmp/p1/A
Ajout          /home/clara/tmp/p1/A/a
Ajout          /home/clara/tmp/p1/A/b

Révision 1 propagée.
```

Ca y est, l'arborescence du projet est située sur le dépôt. Vous remarquerez que le répertoire *p*1 reste inchangé. En fait, Subversion ne tient pas compte de ce répertoire et vous pouvez même le supprimer. Pour pouvoir travailler sur ce projet à l'aide de Subversion, vous allez devoir maintenant le récupérer.

3.3 Récupération d'un projet

Maintenant que le projet est présent dans le dépôt Subversion, vous allez devoir en récupérer une copie en locale pour travailler dessus.

Pour cela, *Clara* exécute la commande ci-dessous :

```
$ svn checkout file:///home/clara/depot/ /home/clara/work/p1
A    /home/clara/work/p1/A
```

```
A    /home/clara/work/p1/A/a
A    /home/clara/work/p1/A/b
Révision 1 extraite.
```

Vous remarquerez que dans chaque répertoire récupéré est présent un répertoire caché *.svn*. Ce répertoire contient des informations pour Subversion et indique que le répertoire est bien pris en considération. Pour visualiser ces informations, vous pouvez utiliser la commande :

```
$ svn info
Chemin : .
URL : file:///home/clara/depot
UUID du dépôt : 0fab0697-e808-0410-8a14-dfa79d20632a
Révision : 1
Type de noeud : répertoire
En attente : normale
Auteur de la dernière modification : clara
Révision de la dernière modification : 1
Date de la dernière modification: 2005-12-27 16:11:47 +0100 (mar, 27 déc 2005)
```

On y retrouve l'adresse du dépôt gérant le projet, le numéro de version, l'auteur de la dernière modification, etc.

3.4 Ajout et modification des fichiers

Clara modifie maintenant le fichier *a* et rajoute un fichier *c*.

Elle peut visualiser les modifications apportées grâce à la commande suivante :

```
$ svn status /home/clara/work/p1/A
?      /home/clara/p1/A/c
M      /home/clara/p1/A/a
```

Dans la première colonne, on peut lire que le fichier *c* possède un statut inconnu signalé par le caractère " ?". Par contre, on peut voir que le fichier *a* a été modifié par le signe "M".

Le fichier *c* possède un statut inconnu car il n'a pas été rajouté dans Subversion. Pour cela, *Clara* va devoir exécuter la commande suivante :

```
$ svn add c
A         c
```

En relançant la commande *svn status*, on remarquera que le symbole " ?" est remplacé par un "A" signifiant que le fichier *c* a été rajouté.

```
$ svn status
M      a
A      c
```

3.5 Validation des modifications

Maintenant que *Clara* est contente de ses modifications, elle va les valider sur le dépôt. Pour cela, elle va exécuter la commande suivante :

```
$ svn commit -m "modification de a et ajout de c"
Envoi          A/a
```

```
Ajout          A/c
Transmission des données ..
Révision 2 propagée.
```

L'option "-m" permet d'indiquer un commentaire concernant les modifications apportées. Ce commentaire est obligatoire et s'il est omis un éditeur de texte s'ouvrira automatiquement.

3.6 Mise à jour des données

Enfin, *Clara* va utiliser la commande *svn update* pour synchroniser les informations situées sur le serveur avec celles de sa copie de travail.

Voici la commande exécutée :

```
$ svn update
À la révision 2.
```

En utilisant la commande *svn info*, on peut voir que la synchronisation a bien eu lieu :

```
$ svn info
Chemin : .
URL : file:///home/clara/depot/A
UUID du dépôt : 0fab0697-e808-0410-8a14-dfa79d20632a
Révision : 2
Type de noeud : répertoire
En attente : normale
Auteur de la dernière modification : clara
Révision de la dernière modification : 2
Date de la dernière modification: 2005-12-27 16:39:35 +0100 (mar, 27 déc 2005)
```

3.7 Création de répertoire et effacement de fichiers

Maintenant, *Clara* souhaite ajouter le répertoire *B* à son projet et effacer le fichier *b*. Il est important de noter que pour Subversion le traitement des répertoires et des fichiers n'est pas différent. De plus, effacer un fichier ou un dossier n'est qu'une vue de l'esprit puisque tout est conservé dans la base Subversion.

Ainsi, imaginons *Clara* qui vient de rajouter le répertoire *B*, contenant le fichier *d*, à la racine de son projet. Pour pouvoir confirmer cet ajout dans le système Subversion, elle va devoir exécuter la commande ci-dessous :

```
$ svn add B
A         B
A         B/d
```

Vous remarquerez que les éléments contenus dans le répertoire *B* ont également été ajoutés.

Maintenant, *Clara* souhaite supprimer le fichier *b* du dépôt. Pour cela, elle va utiliser la commande *svn delete*, comme suit :

```
$ svn delete b
D         b
```

Là, on notera que le fichier *b* a bien été supprimé de l'arborescence.

Maintenant, on peut vérifier les différentes modifications en utilisant la commande *svn status* :

```
$ svn status
D      A/b
A      B
A      B/d
```

De la même façon que précédemment, *Clara* va pouvoir valider ses changements sur le dépôt par la commande :

```
$ svn commit -m "ajout du répertoire B et suppression de c"
Suppression    A/b
Ajout          B
Ajout          B/d
Transmission des données .
Révision 3 propagée.
```

Enfin, elle va pouvoir synchroniser son répertoire avec le dépôt en effectuant un *svn update* :

```
$ svn info
Chemin : .
URL : file:///home/clara/depot
UUID du dépôt : 0fab0697-e808-0410-8a14-dfa79d20632a
Révision : 3
Type de noeud : répertoire
En attente : normale
Auteur de la dernière modification : clara
Révision de la dernière modification : 3
Date de la dernière modification: 2005-12-27 20:01:05 +0100 (mar, 27 déc 2005)
```

3.8 Déplacement de fichiers

Clara vient de remarquer qu'elle a fait une erreur, et elle souhaite déplacer le fichier *a* dans le répertoire *B*. Ce déplacement s'effectue à l'aide de la commande *svn move*, comme suit :

```
$ svn move A/a B/a
A         B/a
D         A/a
```

Vous remarquerez que pour Subversion, déplacer consiste à ajouter un fichier dans *B* et d'effacer l'autre. Ceci apparaît avec un statut en forme de "+" en troisième colonne :

```
svn status
D      A/a
A  +   B/a
```

Lorsque ces modifications sont effectuées, *Clara* peut valider ses modifications sur le dépôt :

```
$ svn commit -m "déplacement du fichier a dans B"
Suppression    A/a
Ajout          B/a
```

3.9 Annulation de modifications

Clara vient de modifier le fichier *c*, mais elle s'aperçoit qu'elle a fait une erreur. Elle a la possibilité d'annuler ses modifications et de revenir à la version précédente de son fichier à l'aide de la commande *svn revert*. Elle exécute donc la commande suivante :

```
$ svn revert c
'c' réinitialisé
```

Ainsi, ses modifications ont été annulées. Cette action peut être validée à l'aide de la commande *svn status* dans laquelle on s'aperçoit que le fichier *c* n'a pas été modifié.

3.10 Résumé d'un cycle de travail

Nous allons ici résumer les différentes commandes de base pour travailler avec Subversion sous forme d'un cycle de commandes de base :

1. Création d'un dépôt : *svnadmin create*
2. Importation d'un projet dans ce dépôt : *svn import*
3. Récupération d'une copie de ce projet : *svn checkout*

   modification du contenu du projet :
   - de façon classique (avec un éditeur de texte)
   - annulation des modifications : *svn revert*
   - rajout de fichiers : *svn add*
   - copie de fichiers : *svn copy*
   - déplacement de fichiers : *svn move*
   - effacement de fichiers : *svn delete*

   consultation des modifications : *svn status*

   validation des modifications : *svn commit*

   synchronisation des révisions : *svn update*

3.11 Les logs

A chaque fois que nous avons soumis des modifications dans la base, nous les avons accompagnées d'un message par le biais de l'option "-m" de la commande *svn commit*. L'ensemble de ces messages peut être consulté à l'aide de l'instruction *svn log*, qui nous affiche le commentaire, la date ainsi que son auteur.

Voici ce qu'obtient *Clara* :

```
$ svn log
------------------------------------------------------------------------
r4 | clara | 2005-12-27 20:17:32 +0100 (mar, 27 déc 2005) | 1 line

déplacement du fichier a dans B
------------------------------------------------------------------------
r3 | clara | 2005-12-27 20:01:05 +0100 (mar, 27 déc 2005) | 1 line

ajout du répertoire B et suppression de c
------------------------------------------------------------------------
r2 | clara | 2005-12-27 16:39:35 +0100 (mar, 27 déc 2005) | 1 line

modification de a et ajout de c
------------------------------------------------------------------------
r1 | clara | 2005-12-27 16:11:47 +0100 (mar, 27 déc 2005) | 1 line

import initial
------------------------------------------------------------------------
```

4 La gestion des conflits

Dans cette partie, nous allons traiter le cas où plusieurs personnes ont accès au même dépôt. Dans l'exemple précédent, tout allait bien car une seule personne travaillait sur les données. Mais dans le cas d'un travail collaboratif, il est fréquent que deux personnes (ou plus) fassent des modifications sur le même fichier de façon simultanée, ce qui entraîne généralement des conflits.

Nous allons donc étudier dans cette section comment Subversion gère les conflits. Nous verrons tout d'abord comment un conflit est détecté à l'aide de Subversion. Ensuite, nous traiterons le cas d'un conflit réglé automatiquement par Subversion. Enfin, nous aborderons le cas des conflits à gérer manuellement.

4.1 Détection d'un conflit

Imaginons *Clara* et *Morgane* qui viennent de récupèrer le même projet et travaillent sur le même fichier *a*. *Clara* vient de terminer ses modifications et valide ses changements sur le dépôt. Ensuite, *Morgane* termine à son tour et essaie de valider ses changements. Subversion détecte à ce moment là que le fichier local n'est pas à jour et annule la validation. *Morgane* doit donc maintenant effectuer une mise à jour de sa copie en local pour récupérer la nouvelle version du serveur. Lors de cette mise à jour, Subversion essaie de fusionner automatiquement les modifications. Ceci est possible si les modifications ne se chevauchent pas. Sinon, la fusion est impossible et ce sera à *Morgane* de l'effectuer. Une fois que le conflit est géré, *Morgane* va pouvoir valider les modifications sur le serveur.

Cette procédure est représentée par la figure 1.

Pour illustrer ce problème, nous allons travailler sur un projet contenant 2 fichiers, *conflit_automatique.txt* et *conflit_manuel.txt*, dont voici respectivement le contenu :

```
1 ligne commune avant
2 ligne à modifier par Clara :
3 ligne à modifier par Morgane :
4 ligne commune après
```

et

```
1 ligne commune avant
2 cette ligne est modifiée par <nom>
3 je suis Clara (oui/non) :
4 je suis Morgane (oui/non) :
5 ligne commune après
```

4.2 Gestion automatique d'un conflit

En fait, si les modifications faites par les personnes ne concernent pas les mêmes lignes, Subversion gère lui-même la fusion des différences. Voici le fichier *conflit_automatique.txt* modifié par *Clara* et par *Morgane* :

```
1 ligne commune avant
2 ligne à modifier par Clara : salut Morgane
3 ligne à modifier par Morgane :
4 ligne commune après
```

et

```
1 ligne commune avant
2 ligne à modifier par Clara :
3 ligne à modifier par Morgane : bonjour Clara
4 ligne commune après
```

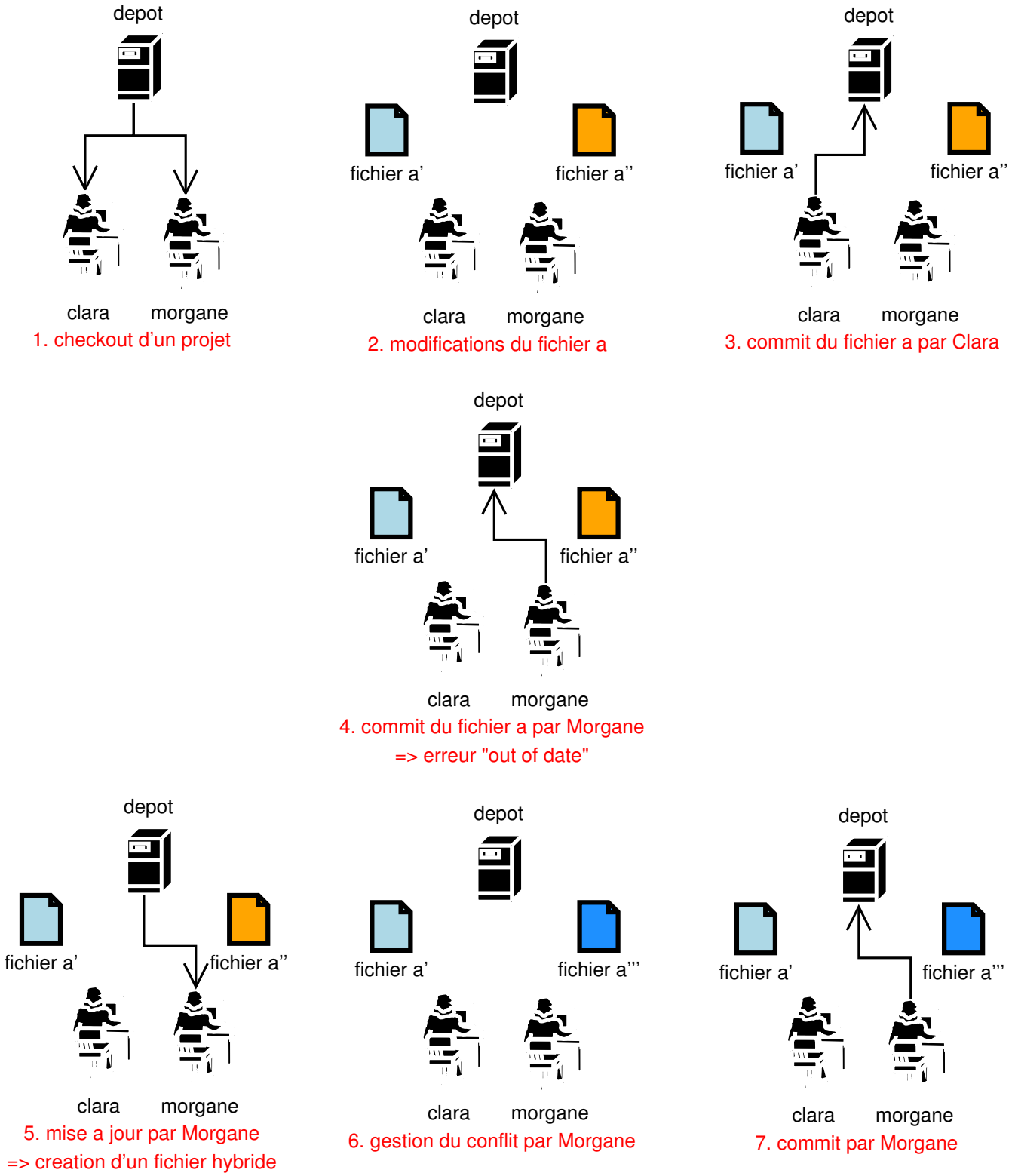

FIG. 1 – Gestion d'un conflit

On peut voir que dans ce cas, aucune modification n'empiète sur celle de l'autre. Ainsi, imaginons maintenant que *Morgane* valide ses modifications :

```
[morgane]$ svn status
M       conflit_automatique.txt
[morgane]$ svn commit -m "modification du fichier conflit_automatique.txt"
```

```
Envoi         conflit_automatique.txt
Transmission des données .
Révision 2 propagée.
```

C'est maintenant au tour de Clara :

```
[clara]$ svn status
M      conflit_automatique.txt
[clara]$ svn commit -m "modification du fichier conflit_automatique.txt"
Envoi         conflit_automatique.txt
svn: Échec de la propagation (commit), détails :
svn: Out of date: 'conflit_automatique.txt' in transaction '3'
```

Le message *svn : Out of date : 'conflit_automatique.txt' in transaction '3'* indique à *Clara* qu'elle doit d'abord mettre à jour sa copie en local. Pour cela, elle va utiliser la commande *svn update* comme suit :

```
[clara]$ svn update
G    conflit_automatique.txt
Actualisé à la révision 2.
```

La lettre "G" dans la première colonne indique que les modifications contenues dans la base n'impiètent pas sur celles réalisées en local. En effet, voici le contenu du fichier *conflit_automatique.txt* de *Clara* :

```
ligne commune avant
ligne à modifier par Clara : salut Morgane
ligne à modifier par Morgane : bonjour Clara
ligne commune après
```

Ainsi, *Clara* va pouvoir maintenant propager ses modifications avec la commande *svn commit* :

```
[clara]$ svn commit -m "modification du fichier conflit_automatique.txt"
Envoi         conflit_automatique.txt
Transmission des données .
Révision 3 propagée.
```

Morgane pourra mettre à jour sa version du fichier en exécutant la commande *svn update* :

```
[morgane]$ svn update
U    conflit_automatique.txt
Actualisé à la révision 3.
```

Elle trouvera que le fichier *conflit_automatique.txt* a été modifié et possède le contenu suivant :

```
ligne commune avant
ligne à modifier par Clara : salut Morgane
ligne à modifier par Morgane : bonjour Clara
ligne commune après
```

4.3 Gestion manuelle d'un conflit

Maintenant, prenons le cas d'un réel conflit et de sa gestion manuelle. Pour cela, *Clara* et *Morgane* vont modifier le fichier *conflit_manuel.txt*. Voici leur contenu respectif après modifications :

```
ligne commune avant
cette ligne est modifiée par Clara
je suis Clara (oui/non) : oui
je suis Morgane (oui/non) : non
ligne commune après
```

et

```
ligne commune avant
cette ligne est modifiée par Morgane
je suis Clara (oui/non) : non
je suis Morgane (oui/non) : je ne sais pas
ligne commune après
```

Toutes les modifications sont cette fois-ci incompatibles. *Morgane* soumet ses modifications :

```
[morgane]$ svn commit -m "modifications du fichier conflit_manuel.txt"
Envoi          conflit_manuel.txt
Transmission des données .
Révision 4 propagée.
```

Puis c'est au tour de *Clara* :

```
[clara]$ svn commit -m "modifications du fichier conflit_manuel.txt"
Envoi          conflit_manuel.txt
svn: Échec de la propagation (commit), détails :
svn: Out of date: 'conflit_manuel.txt' in transaction '5'
```

Comme précédemment, *Clara* utilise la commande *svn update* pour mettre à jour sa copie locale :

```
[clara]$ svn update
C    conflit_manuel.txt
Actualisé à la révision 4.
```

La lettre C dans la première colonne indique cette fois-ci un conflit à gérer manuellement. Lorsque ceci arrive sur un fichier *file*, Subversion modifie le répertoire local de la personne en ajoutant différents fichiers :

- *file* : contient maintenant l'ensemble des modifications,
- *file.mine* : contient le fichier *file* modifié par *Clara* , avant que celle-ci ne mette à jour son répertoire,
- *file.r#old* (où #old est le numéro de l'ancienne révision) : c'est le fichier de la base avant que l'utilisateur ne fasse ses propres modifications,
- *file.r#new* (où #new est le numéro de la nouvelle révision) : c'est le fichier de la base le plus récent

Dans notre cas, *Clara* possède :

```
[clara]$ ls -l
total 28
-rw-r--r--  1 clara clara 118 2005-12-29 16:18 conflit_automatique.txt
-rw-r--r--  1 clara clara 280 2005-12-29 16:27 conflit_manuel.txt
-rw-r--r--  1 clara clara 130 2005-12-29 16:27 conflit_manuel.txt.mine
-rw-r--r--  1 clara clara 122 2005-12-29 16:27 conflit_manuel.txt.r3
-rw-r--r--  1 clara clara 144 2005-12-29 16:27 conflit_manuel.txt.r4
```

Et voici le contenu du fichier *conflit_manuel.txt* :

```
ligne commune avant
<<<<<<< .mine
cette est ligne est modifiée par Clara
je suis Clara (oui/non) : oui
je suis Morgane (oui/non) : non
=======
cette est ligne est modifiée par Morgane
je suis Clara (oui/non) : non
je suis Morgane (oui/non) : je ne sais pas
>>>>>>> .r9
ligne commune après
```

Voici le fichier *conflit_manuel.txt.mine* :

```
ligne commune avant
cette est ligne est modifiée par Clara
je suis Clara (oui/non) : oui
je suis Morgane (oui/non) : non
ligne commune après
```

Voici le fichier *conflit_manuel.txt.r3* :

```
ligne commune avant
cette est ligne est modifiée par <nom>
je suis Clara (oui/non) :
je suis Morgane (oui/non) :
ligne commune après
```

Voici le fichier *conflit_manuel.txt.r4* :

```
ligne commune avant
cette est ligne est modifiée par Morgane
je suis Clara (oui/non) : non
je suis Morgane (oui/non) : je ne sais pas
ligne commune après
```

Clara a alors trois possibilités :
- gérer le conflit manuellement,
- écraser le fichier du répertoire par l'une de ces variantes temporaires,
- taper la commande *svn revert* suivie du nom du fichier pour éliminer les modifications locales.

Nous allons gérer ici le conflit manuellement. *Clara* change le contenu du fichier en :

```
ligne commune avant
cette est ligne est modifiée par Morgane puis Clara
je suis Clara (oui/non) : oui
je suis Morgane (oui/non) : non
ligne commune après
```

Clara va ensuite indiquer à Subversion que le conflit a été résolu en utilisant la commande *svn resolved* suivie du nom du fichier concerné. Cette commande efface également les fichiers temporaires. Ensuite, *Clara* va propager comme d'habitude les modifications :

```
[clara]$ svn resolved conflit_manuel.txt
Conflit sur 'conflit_manuel.txt' résolu
```

```
[clara]$ svn commit -m "modifications du fichier conflit_manuel.txt"
Envoi          conflit_manuel.txt
Transmission des données .
Révision 5 propagée.
```

5 Gestion de l'historique

Subversion permet donc de mémoriser tous les changements effectués sur les fichiers. Il est donc possible de suivre ces modifications, de revoir les versions plus anciennes, de reprendre des fichiers de versions différentes et d'en faire une nouvelle version de projet, de comparer des versions de fichiers entre elles, etc. Nous allons donc voir ces différents aspects dans cette partie.

5.1 svn log

La commande *svn log* permet de suivre les modifications faites à chaque nouvelle version. Par exemple, si *Clara* effectue cette commande, elle obtient le résultat suivant pour l'exemple précédent :

```
[clara] $ svn log
------------------------------------------------------------------------
r10 | clara | 2005-12-29 16:39:04 +0100 (jeu, 29 déc 2005) | 1 line

modifications du fichier conflit_manuel.txt
------------------------------------------------------------------------
r9 | morgane | 2005-12-29 16:25:50 +0100 (jeu, 29 déc 2005) | 1 line

modifications du fichier conflit_manuel.txt
------------------------------------------------------------------------
r8 | clara | 2005-12-29 16:20:08 +0100 (jeu, 29 déc 2005) | 1 line

modification du fichier conflit_automatique.txt
------------------------------------------------------------------------
r7 | morgane | 2005-12-29 16:14:52 +0100 (jeu, 29 déc 2005) | 1 line

modification du fichier conflit_automatique.txt
------------------------------------------------------------------------
r6 | clara | 2005-12-29 16:09:15 +0100 (jeu, 29 déc 2005) | 1 line

import initial
------------------------------------------------------------------------
```

L'option *−v* détaille plus précisément les changements effectués sur chaque version. En effet, elle nous informe sur les fichiers qui ont été modifiés. Par exemple, si *Clara* effectue cette commande :

```
[clara] $ svn log -v
------------------------------------------------------------------------
r5 | clara | 2005-12-29 16:39:04 +0100 (jeu, 29 déc 2005) | 1 line
Chemins modifiés :
   M /conflit_manuel.txt

modifications du fichier conflit_manuel.txt
------------------------------------------------------------------------
r4 | morgane | 2005-12-29 16:25:50 +0100 (jeu, 29 déc 2005) | 1 line
Chemins modifiés :
   M /conflit_manuel.txt

modifications du fichier conflit_manuel.txt
------------------------------------------------------------------------
r3 | clara | 2005-12-29 16:20:08 +0100 (jeu, 29 déc 2005) | 1 line
Chemins modifiés :
   M /conflit_automatique.txt

modification du fichier conflit_automatique.txt
```

```
------------------------------------------------------------------------
r2 | morgane | 2005-12-29 16:14:52 +0100 (jeu, 29 déc 2005) | 1 line
Chemins modifiés :
   M /conflit_automatique.txt

modification du fichier conflit_automatique.txt
------------------------------------------------------------------------
r1 | clara | 2005-12-29 16:09:15 +0100 (jeu, 29 déc 2005) | 1 line
Chemins modifiés :
   A /conflit_automatique.txt
   A /conflit_manuel.txt

import initial
------------------------------------------------------------------------
```

Enfin, il est possible de préciser avec l'option *−r* un seul numéro de révision, ou même une plage de révisions si l'on donne deux numéros séparés par le caractère " :". Par exemple, pour suivre les modifications entre la révision 3 et la révision 5, *Clara* va exécuter la commande :

```
[clara] $ svn log -r 3:5 -v
------------------------------------------------------------------------
r3 | clara | 2005-12-29 16:20:08 +0100 (jeu, 29 déc 2005) | 1 line
Chemins modifiés :
   M /conflit_automatique.txt

modification du fichier conflit_automatique.txt
------------------------------------------------------------------------
r4 | clara | 2005-12-29 16:25:50 +0100 (jeu, 29 déc 2005) | 1 line
Chemins modifiés :
   M /conflit_manuel.txt

modifications du fichier conflit_manuel.txt
------------------------------------------------------------------------
r5 | morgane | 2005-12-29 16:39:04 +0100 (jeu, 29 déc 2005) | 1 line
Chemins modifiés :
   M /conflit_manuel.txt

modifications du fichier conflit_manuel.txt
------------------------------------------------------------------------
```

5.2 svn diff

La commande *svn diff* permet de regarder les différences entre deux révisions données.

Tout d'abord, la commande *svn diff* permet de comparer les fichiers modifiés localement avec ceux de la base. Par exemple, si *Clara* vient de modifier le fichier *conflit_manuel.txt*, elle obtient :

```
[clara] $ svn status
M      conflit_manuel.txt
[clara] $ svn diff
Index: conflit_manuel.txt
===================================================================
--- conflit_manuel.txt  (révision 5)
+++ conflit_manuel.txt  (copie de travail)
@@ -2,4 +2,5 @@
 ligne commune avant
 cette est ligne est modifiée par Morgane puis Clara
```

```
 je suis Clara (oui/non) : oui
 je suis Morgane (oui/non) : non
 ligne commune après
+ajout pour svn diff
```

Bien entendu, cette commande ne fonctionne que si les changements n'ont pas été propagés sur le dépôt.

Il est également possible de comparer les fichiers locaux à ceux d'une version donnée en utilisant l'option $-r$. Par exemple, si *Clara* souhaite comparer son fichier modifié localement avec la révision 1, elle va exécuter :

```
[clara] $ svn diff -r 1
Index: conflit_automatique.txt
===================================================================
--- conflit_automatique.txt     (révision 1)
+++ conflit_automatique.txt     (copie de travail)
@@ -1,4 +1,4 @@
 ligne commune avant
-ligne à modifier par Clara :
-ligne à modifier par Morgane :
+ligne à modifier par Clara : salut Morgane
+ligne à modifier par Morgane : bonjour Clara
 ligne commune après
Index: conflit_manuel.txt
===================================================================
--- conflit_manuel.txt  (révision 1)
+++ conflit_manuel.txt  (copie de travail)
@@ -1,5 +1,6 @@
 ligne commune avant
-cette est ligne est modifiée par <nom>
-je suis Clara (oui/non) :
-je suis Morgane (oui/non) :
+cette est ligne est modifiée par Morgane puis Clara
+je suis Clara (oui/non) : oui
+je suis Morgane (oui/non) : non
+ajout pour svn diff
```

Pour pouvoir préciser un fichier en particulier, il faut écrire son nom comme dernier paramètre. Pour le fichier *conflit_manuel.txt*, *Clara* va donc exécuter la commande suivante :

```
[clara] $ svn diff -r 1 conflit_manuel.txt
Index: conflit_manuel.txt
===================================================================
--- conflit_manuel.txt  (révision 1)
+++ conflit_manuel.txt  (copie de travail)
@@ -1,5 +1,6 @@
 ligne commune avant
-cette est ligne est modifiée par <nom>
-je suis Clara (oui/non) :
-je suis Morgane (oui/non) :
-ligne commune après
+cette est ligne est modifiée par Morgane puis Clara
+je suis Clara (oui/non) : oui
+je suis Morgane (oui/non) : non
+ligne commune après
+ajout pour svn diff
\ No newline at end of file
```

Enfin, il est possible de comparer deux versions en séparant leur numéro par le caractère " :". Par exemple, pour comparer le fichier *conflit_automatique.txt* entre les révisions 1 et 3, *Clara* devra exécuter :

```
[clara] $ svn diff -r 1:3 conflit_automatique.txt
Index: conflit_automatique.txt
===================================================================
--- conflit_automatique.txt     (révision 1)
+++ conflit_automatique.txt     (révision 3)
@@ -1,4 +1,4 @@
 ligne commune avant
-ligne à modifier par Clara :
-ligne à modifier par Morgane :
+ligne à modifier par Clara : salut Morgane
+ligne à modifier par Morgane : bonjour Clara
 ligne commune après
```

5.3 svn update

La commande *svn update* sans paramètre permet de récupérer la dernière version du projet présente sur le serveur. Cependant, elle permet aussi de récupérer une version donnée en spécifiant un numéro de révision par l'option $-r$.

5.4 svn cat

La commande *svn cat* permet d'afficher le contenu d'un fichier à révision donnée par l'option "-r". Par exemple, si *Clara* souhaite visualiser le fichier *conflit_manuel.txt* en révision 1, elle va exécuter :

```
[clara] $ svn cat -r 1 conflit_manuel.txt
ligne commune avant
cette est ligne est modifiée par <nom>
je suis Clara (oui/non) :
je suis Morgane (oui/non) :
ligne commune après
```

Cette commande permet par exemple de récupérer un fichier à une révision donnée dans un fichier local afin d'utiliser un outil de comparaison graphique plus pratique (*kdiff3* par exemple) que la commande *diff*.

5.5 svn revert

La commande *svn revert* permet d'annuler les modifications apportées localement à un fichier, et de récupérer la version la plus récente en base. Ainsi, pour annuler ses modifications sur le fichier *conflit_manuel.txt*, *Clara* va effectuer :

```
[clara] $ svn status
M      conflit_manuel.txt
[clara] $ svn revert conflit_manuel.txt
'conflit_manuel.txt' réinitialisé
```

5.6 svn list

La commande *svn list* permet de lister les fichiers d'un dépôt. En utilisant l'option $-v$, on obtient un résultat plus détaillé. Par exemple, si *Clara* effectue cette commande, elle obtient :

```
[clara] $ svn list -v
      3 clara            118 déc 29 16:20 conflit_automatique.txt
      5 clara            153 déc 29 16:39 conflit_manuel.txt
```

6 Tags et branches

6.1 Méthologie

Les tags et les branches sont deux notions particulièrement importantes dans la gestion de versions.

Les tags permettent d'attribuer une étiquette à une révision du projet, permettant de la retrouver plus facilement. En d'autres termes, un tag peut être considéré comme une sauvegarde du projet à un état particulier.

Les branches permettent de fabriquer une sorte de nouveau projet à partir d'une révision particulière d'un projet. Ainsi, une personne pourra travailler sur cette nouvelle branche pour par exemple tester des algorithmes. Si ces essais sont validés, ils pourront être réinjectés par la suite dans la branche d'origine.

Dans le cadre du développement de logiciels, ces notions sont très pratiques et doivent être reliées directement à la gestion de projet.

Prenons un exemple concret : *Clara* et *Morgane* doivent réaliser un logiciel pour un client. Après avoir rédigé un cahier des charges détaillé, le logiciel a pu être décomposé en 5 fonctionnalités qui ont été classées par ordre de priorité. Le client ayant besoin des 2 premières fonctionnalités au plus tôt, il a été décidé qu'une première version de ce logiciel, ne contenant que les fonctionnalités 1 et 2, devra être livrée. Ensuite, une deuxième version contenant les trois fonctionnalités restantes sera livrée plus tard.

En tant que chef de projet, *Clara* a dressé le diagramme représenté par l'image 2.

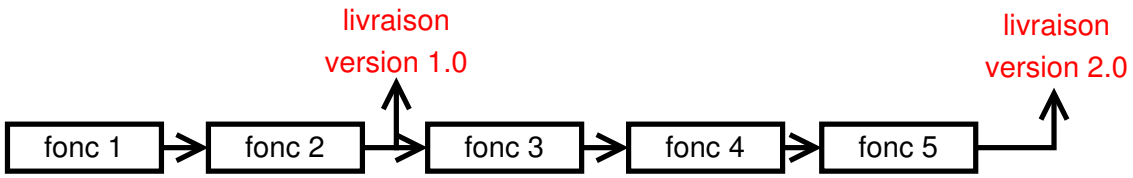

FIG. 2 – Diagramme de livraison des versions

Avant de commencer le développement, *Clara* décide d'organiser son dépôt Subversion en créant trois répertoires :
- *trunk :* ce répertoire va contenir le projet en développement.
- *tags :* ce répertoire va contenir la version 1.0 et la version 2.0. Il va aussi contenir les versions intermédiaires. Personne ne devra travailler sur les fichiers présents dans ce répertoire, il sert en quelque sorte de répertoire de sauvegarde.
- *branches :* lorsque la version 1.0 sera livrée au client, celui-ci va sans doute détecter des bugs. Cependant, il ne voudra certainement pas attendre la livraison de la version 2.0 pour qu'ils soient corrigés. Ainsi, les bugs de la version 1.0 seront corrigés en créant une nouvelle branche dans ce répertoire. Lorsque les corrections seront validées, des versions intermédiaires (1.1, 1.2...) seront livrées au client et les modifications seront fusionnées dans la branche principale (*trunk*).

Pour être plus claire, *Clara* a dessiné le processus représenté par le diagramme 3.

Ainsi, elles vont tout d'abord travailler sur la première version du logiciel. Ce travail va s'effectuer dans le répertoire */trunk/*. Lorsque les deux fonctionnalités seront terminées, elles vont tagger la version. Pour cela, elle vont en fait copier la version actuelle de */trunk/* dans le répertoire */tags/1.0/*. C'est cette version qui sera livrée au client.

Ensuite, elles vont travailler sur la deuxième version. Ce travail va toujours s'effectuer dans le répertoire */trunk/*. Si le client détecte des bugs dans la version qu'il utilise (la version 1.0) avant que la deuxième version ne soit validée, *Clara* va créer une nouvelle branche. Pour cela, elle va en fait copier la version taggée */tags/1.0/* dans un répertoire */branches/1.1*. A partir de là, elle va corriger le bug dans ce répertoire. Lorsque l'erreur sera corrigée, elle va tagger cette nouvelle version en créant une copie à l'emplacement */tags/1.1/*. Cette version sera livrée au client. Elle fusionnera aussi les modifications apportées sur cette branche avec le projet principal développé dans le répertoire */trunk/*.

Enfin, elle va retourner travailler sur le développement de la version 2.0 dans le répertoire */trunk/*. La prochaine fois que le client détectera un bug, une nouvelle branche */branches/1.2/* sera créée à partir de la version taggée */tags/1.1*, et ainsi de suite.

Cette organisation n'est en fait qu'une convention de nommage proposée par les créateurs de Subversion. Vous pouvez modifier le mode de gestion de votre référentiel mais ce serait un mauvais choix en raison des différentes documentations de Subversion qui l'utiliseront : les nouveaux développeurs du projet devront s'habituer à votre mode de fonctionnement ce qui

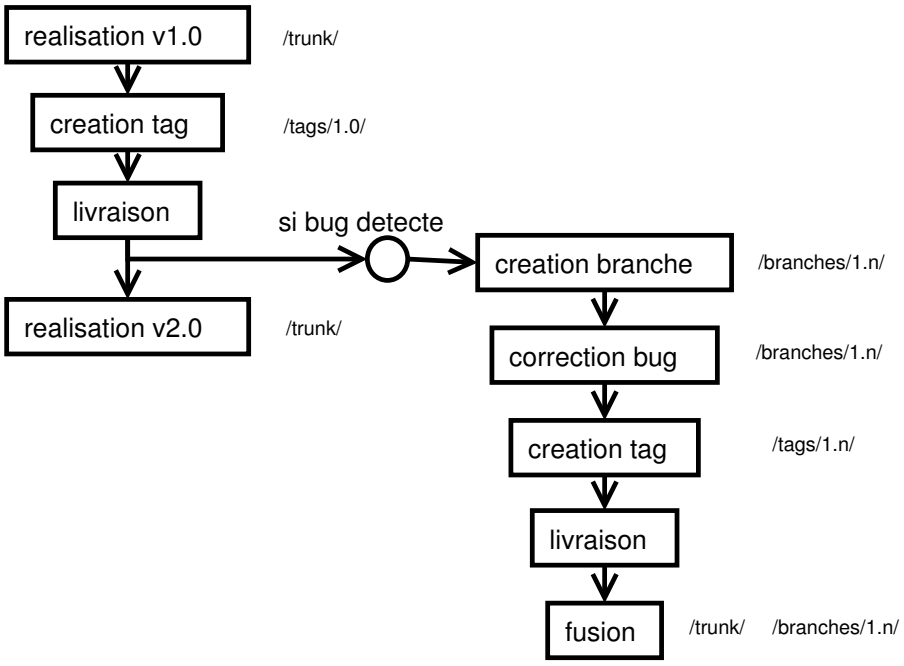

FIG. 3 – Processus de réalisation du logiciel

n'est pas forcément une bonne chose.

6.2 Exemple pratique

Nous allons maintenant mettre en oeuvre l'exemple ci-dessus. Pour cela, nous allons considérer que le projet n'est composé que d'un seul fichier *projet.txt*. Ce fichier contient 5 lignes, une pour chaque fonctionnalité à développer. Voici son contenu :

```
fonc 1 : vide
fonc 2 : vide
fonc 3 : vide
fonc 4 : vide
fonc 5 : vide
```

Lorsqu'une fonctionnalité sera réalisée, le texte "vide" sera remplacé par "ok". Si un bug est détecté sur une fonctionnalité et qu'il est corrigé, le texte sera "ok, correction 1".

6.2.1 Initialisation du dépôt

Nous allons tout d'abord créer les trois répertoires :

```
[clara] $ svn mkdir file:///home/clara/depot/trunk file:///home/clara/depot/tags
file:///home/clara/depot/branches -m "création des répertoires"

Révision 1 propagée.
```

Ensuite, *Clara* va récupérer l'arborescence créée à l'aide de la commande suivante :

```
[clara] $ svn checkout file:///home/clara/depot/
A    depot/trunk
A    depot/tags
```

```
A    depot/branches
Révision 1 extraite.
```

Maintenant, *Clara* va créer dans le répertoire */trunk/* le fichier *projet.txt*. Après, elle va l'ajouter au dépôt à l'aide de la commande suivante :

```
[clara] $ svn add projet.txt
A         projet.txt
[clara] $ svn status
A      projet.txt
[clara] $ svn commit -m "ajout du fichier projet"
Ajout          trunk/projet.txt
Transmission des données .
Révision 2 propagée.
```

6.2.2 Travail sur la version 1.0

Maintenant que le dépôt est initialisé, *Clara* et *Morgane* travaillent sur le projet, *Clara* sur la première fonctionnalité et *Morgane* sur la seconde. Chacune travaille sur une copie en local sur leur poste. Au bout d'un certain temps, la première fonctionnalité est terminée. Le fichier *projet.txt* devient donc :

```
fonc 1 : ok
fonc 2 : vide
fonc 3 : vide
fonc 4 : vide
fonc 5 : vide
```

Clara propage ses modifications en exécutant la commande suivante :

```
[clara] $ svn status
M      projet.txt
[clara] $ svn commit -m "réalisation de la fonc 1"
Envoi          trunk/projet.txt
Transmission des données .
Révision 3 propagée.
```

Morgane vient maintenant de terminer et comme elle n'a pas mis à jour sa copie en local, le fichier *projet.txt* possède le contenu suivant :

```
fonc 1 : vide
fonc 2 : ok
fonc 3 : vide
fonc 4 : vide
fonc 5 : vide
```

Elle effectue maintenant sa sauvegarde.

```
[morgane] $ svn status
M      projet.txt
[morgane] $ svn commit -m "realisation func 2"
Envoi          trunk/projet.txt
svn: Échec de la propagation (commit), détails :
svn: Out of date: '/trunk/projet.txt' in transaction '3-1'
[morgane] $ svn update
```

```
G    projet.txt
Actualisé à la révision 3.
[morgane] $ svn commit -m "realisation func 2"
Envoi          trunk/projet.txt
Transmission des données .
Révision 4 propagée.
```

Vous remarquerez qu'un conflit a eu lieu sur le fichier *projet.txt*. En effet, *Morgane* n'a pas effectué de mise à jour et à modifier le même fichier que *Clara* . Cependant, comme elles n'ont pas travaillé sur les mêmes lignes, la fusion a été faite automatiquement par Subversion.

6.2.3 Tag de la version 1.0

Ca y est, les deux premières fonctionnalités sont réalisées, il faut maintenant tagger la version. Pour cela, *Clara* va tout d'abord effectuer une mise à jour de son dépôt.

```
[clara] $ svn update
U    projet.txt
Actualisé à la révision 4.
```

Ensuite, elle va tagger la version 1.0. En fait, dans Subversion, "tagger" consiste tout simplement à copier. Ainsi, *Clara* va exécuter la commande suivante :

```
[clara] $ svn copy file:///home/clara/depot/trunk/ file:///home/clara/depot/tags/1.0
-m "création du tag 1.0"

Révision 5 propagée.
```

Enfin, elle va mettre à jour sa copie en local et elle va s'apercevoir que le répertoire 1.0 a été créé.

```
[clara] $ svn update
A    tags/1.0
A    tags/1.0/projet.txt
Actualisé à la révision 5.
[clara] $ ls tag
1.0
```

Cette version du projet est donc sauvegardée et elle va pouvoir être livrée au client.

6.2.4 Réalisation de la version 2.0 et correction des bugs

Deux tâches vont maintenant être réalisées en parallèle :
- la réalisation des fonctionnalités 3, 4 et 5 pour la version 2.0. Les fonctionnalités 3 et 4 vont être réalisées par *Clara* et la 5 par *Morgane* .
- la correction des bugs remontés par le client. *Morgane* est responsable de la correction des bugs.

Après quelques temps de travail, *Clara* termine la fonctionnalité 4 et propage ses modifications. Le contenu du fichier est désormais :

```
fonc 1 : ok
fonc 2 : ok
fonc 3 : ok
fonc 4 : vide
fonc 5 : vide
```

Et voici la commande exécutée :

```
[clara] $ svn status
M      projet.txt
[clara] $ svn commit -m "realisation func 3"
Envoi          trunk/projet.txt
Transmission des données .
Révision 6 propagée.
```

Pendant ce moment là, le client a utilisé la version livrée et vient de détecter un bug sur la fonctionnalité 2. Il le remonte donc à *Morgane* . Pour satisfaire le client, *Morgane* va stopper son travail sur la fonctionnalité 5 et va corriger le bug. Seulement, le bug ne va pas être corrigé directement sur la version située dans */trunk/*. En effet, cette version est en cours de développement, certaines fonctionnalités ont été rajoutées mais tout n'est pas encore parfaitement stable. Ainsi, pour ne pas déranger le développement de cette version, une branche va être créée à partir de la version que possède le client. Sachant que cette version est taggée à l'emplacement */tags/1.0/*, *Morgane* va tout simplement copier ce répertoire à l'emplacement */branches/1.1/*. En fait, comme les tags, une branche est tout simplement une copie d'un répertoire.

Morgane va donc effectuer la commande suivante :

```
[morgane] $ svn copy file:///home/clara/depot/tags/1.0/ file:///home/clara/depot/branches/1.1/
-m "creation de la branche 1.1"

Révision 7 propagée.
```

Ensuite, elle va mettre à jour sa copie en local avec la commande suivante :

```
[morgane] $ svn update
A    branches/1.1
A    branches/1.1/projet.txt
Actualisé à la révision 7.
```

Voilà, la nouvelle branche est créée, *Morgane* va maintenant pouvoir corriger le bug sur la fonctionnalité 2. Elle modifie donc le fichier *projet.txt* situé dans le répertoire */branches/*1.1/. Ce fichier possède maintenant le contenu suivant :

```
fonc 1 : ok
fonc 2 : ok, correction 1
fonc 3 : vide
fonc 4 : vide
fonc 5 : vide
```

La correction est terminée, *Morgane* propage ses modifications sur la base :

```
[morgane] $ svn status
M      projet.txt
[morgane] $ svn commit -m "correction de la fonc 2"
Envoi          1.1/projet.txt
Transmission des données .
Révision 8 propagée.
```

Cette version est bonne et va être livrée au client. Un nouveau tag va donc être créé pour cette version. Ainsi, *Morgane* crée le tag 1.1 à l'aide de la commande suivante :

```
[morgane] $ svn copy file:///home/clara/depot/branches/1.1 file:///home/clara/depot/tags/1.1
-m "création du tag 1.1"

Révision 9 propagée.
```

Morgane termine par une mise à jour de sa copie en local :

```
[morgane] $ svn update
A    tags/1.1
A    tags/1.1/projet.txt
Actualisé à la révision 9.
```

Morgane va pouvoir maintenant retourner au développement de la fonctionnalité 5. Si un nouveau bug est détecté par le client, la même procédure sera effectuée, mais cette fois-ci en récupérant la version taggée 1.1.

6.3 Fusion de branches

Lorsque le bug sur la fonctionnalité 2 a été corrigé, *Clara* aurait pu fusionner cette correction avec son développement courant. Cette fusion est effectuée grâce à la commande *svn merge*. En fait, la fusion va consister à prendre les changements qui ont été apportés sur la branche 1.1 et à les appliquer sur la branche principale.

On peut voir ces changements à l'aide de la commande *svn diff*. *Clara* obtient par exemple :

```
[clara] $ svn diff -r 7:8 file:///home/clara/depot/branches/1.1/Index: projet.txt
===================================================================
--- projet.txt  (révision 7)
+++ projet.txt  (révision 8)
@@ -1,5 +1,5 @@
 fonc 1 : ok
-fonc 2 : ok
+fonc 2 : ok, correction 1
 fonc 3 : vide
 fonc 4 : vide
 fonc 5 : vide
```

A noter que les numéros de révisions indiqués dans l'option $-r$ ont été obtenus à l'aide de la commande *svn log*.

On va donc prendre ces modifications pour les apporter sur la branche principale. Pour cela, *Clara* va se déplacer dans le répertoire */trunk/*, puis elle va excécuter la commande suivante :

```
[clara] $ svn merge -r 7:8 file:///home/clara/depot/branches/1.1/
U    projet.txt
```

Comme vous pouvez le voir, le fichier *projet.txt* a été mis à jour, et voici son nouveau contenu :

```
fonc 1 : ok
fonc 2 : ok, correction 1
fonc 3 : ok
fonc 4 : vide
fonc 5 : vide
```

La fusion a été réalisée automatiquement, mais il se peut que des conflits apparaissent. Dans ce cas, il faut suivre la même procédure qu'énoncée précédemment.

La dernière étape de *Clara* va d'être de sauvegarder les changements sur la base. Pour cela, elle exécute la commande suivante :

```
[clara] $ svn commit -m "fusion avec la branche 1.1"
Envoi          trunk/projet.txt
Transmission des données .
Révision 10 propagée.
```

Enfin, les filles vont devoir travailler pour corriger toutes les erreurs qu'elles ont faites et pour terminer la deuxième version du logiciel.

7 Références

Voici la liste des sites internet officiels des logiciels de gestion des versions les plus répandus :

- *CVS :* `http://ximbiot.com/cvs/`
- *Subversion :* `http://subversion.tigris.org/`
- *Clearcase :* `http://www-306.ibm.com/software/awdtools/clearcase/`
- *Visual SourceSafe :* `http://msdn.microsoft.com/vstudio/previous/ssafe/productinfo/overview/`
- *Darcs :* `http://www.darcs.net/`
- *Perforce :* `http://www.perforce.com/`
- *BitKeeper :* `http://www.bitkeeper.com/`
- *GNU Arch :* `http://gnuarch.org/`

Vous trouverez dans la liste ci-dessous les sites internet des outils associés à Subversion :

- *eSVN :* `http://esvn.umputun.com/`
- *Subclipse :* `http://subclipse.tigris.org/`
- *TortoiseSVN :* `http://tortoisesvn.tigris.org/`
- *RapidSVN :* `http://rapidsvn.tigris.org/`

Enfin, voici d'autres sites traitant du sujet :

- *Wikipedia français :* `http://fr.wikipedia.org/wiki/Gestion_de_version`
- *Wikipedia anglais :* `http://en.wikipedia.org/wiki/Version_control_system`

A Outils liés à Subversion

L'architecture modulaire de Subversion permet de développer aisément des applications tierces, dont voici une brève présentation.

A.1 Clients et plugins

- **AnkhSVN :** add-in pour l'IDE Microsoft Visual .NET. (`http://ankhsvn.tigris.org/`)
- **JSVN :** client pour Subversion en Java, incluant aussi un plugin pour IDEA. (`http://jsvn.alternatecomputing.com/`)
- **psvn.el :** interface Subversion pour l'éditeur Emacs. (`http://xsteve.nit.at/prg/vc_svn/`)
- **RapidSVN :** interface graphique multi-plateforme pour Subversion, basée sur les librairies WxPython. (`http://rapidsvn.tigris.org/`)
- **Subclipse :** plugin Subversion pour l'environnement Eclipse. (`http://subclipse.tigris.org/`)
- **Subway :** plugin pour IDE compatibles avec l'api SCC de Microsoft (Visual Studio .Net). (`http://nidaros.homedns.org/subway/`)
- **sourcecross.org :** plugin pour IDE compatibles avec l'api SCC de Microsoft (Visual Studio .Net). (`http://www.sourcecross.org/`)
- **Supervision :** client pour Subversion en Java/Swing. (`http://supervision.tigris.org/`)
- **Sven :** interface graphique native pour Subversion utilisant le framework Cocoa de Mac OS X. (`http://www.nikwest.de/Software/#SvenOverview`)
- **Svn4Eclipse :** plugin Subversion pour l'environnement Eclipse. (`http://svn4eclipse.tigris.org/`)
- **Svn-Up :** interface graphique pour Subversion en Java et plugin pour l'IDE IDEA. (`http://svnup.tigris.org/`)
- **TortoiseSVN :** interface graphique pour SVN intégrée dans l'explorateur Windows. (`http://tortoisesvn.tigris.org/`)
- **WorkBench :** interface de développement multi-plateforme réalisée en Python et basée sur Subversion. (`http://pysvn.tigris.org/`)

A.2 Librairies de développement

- **PySVN :** interface Python orientée objet pour l'API client de Subversion. (`http://pysvn.tigris.org/`)
- **Subversion :** abstraction Python, Perl et Java de l'API Subversion. (`http://subversion.tigris.org/`)
- **SVNCPP :** abstraction C++ orientée objet pour l'API client Subversion. (`http://rapidsvn.tigris.org/`)

A.3 Convertisseur de dépôt

- **cvs2svn :** convertisseur CVS vers Subversion. (`http://cvs2svn.tigris.org/`)
- **vss2svn :** convertisseur Microsoft SourceSafe vers Subversion. (`http://vss2svn.tigris.org/`)
- **Subversion VCP Plugin :** plugin VCP pour la conversion CVS vers Subversion. (`http://svn.clkao.org/revml/branches/svn-perl/`)

A.4 Outils haut niveau

- **Trac :** logiciel de gestion de projet et de suivi de bugs basé sur une interface web. Intègre un wiki et une interface de gestion des versions. (`http://projects.edgewall.com/trac`)
- **Scmbug :** logiciel de gestion des configurations avec suivi des bugs, intégrant Subversion. (`http://freshmeat.net/projects/scmbug/`)
- **Subissue :** suivi des bugs intégré dans le dépôt Subversion. (`http://subissue.tigris.org/`)
- **Subwiki :** Wiki qui utilise Subversion pour son dépôt de données. (`http://subwiki.tigris.org/`)
- **svk :** système de contrôle des versions décentralisé basé sur Subversion. (`http://svk.elixus.org/`)
- **submaster :** système pour le développement de logiciel distribué basé sur Subversion. (`http://www.rocklinux.org/submaster.html`)

A.5 Outils de visualisation de dépôt

- **SVN : :Web :** interface Web basée sur Perl pour visualiser un dépôt Subversion. (`http://svn.elixus.org/repos/member/clkao/`)
- **ViewCVS :** script CGI en Python pour visualiser les dépôts CVS et Subversion. (`http://viewcvs.sourceforge.net/`)
- **WebSVN :** interface PHP pour visualiser un dépôt Subversion. (`http://websvn.tigris.org/`)

B Glossaire

- **commit :** opération consistant à confirmer dans le référentiel les modifications apportées sur la copie de travail.
- **copie de travail ou copie locale :** partie ou ensemble d'un référentiel copié sur le poste de la personne chargée d'y faire des modifications.
- **référentiel :** équivalent français pour "repository". Certains préfèrent la traduction littérale "dépôt". Base de données conservant les différentes versions d'une ou plusieurs ressources.
- **ressource :** tout document (fichier, répertoire, lien, ...) enregistré dans le référentiel.
- **révision :** toute modification effectuée sur une des ressources du référentiel produit une révision. Un numéro unique est donné à chacune de ces modifications : r0, r1, r2, r3, ... Un numéro de révision est aussi affecté au référentiel lui même.
- **tag/étiquette :** marquage fait sur une révision afin d'y revenir plus tard.

C Subversion : mémo

C.1 Types de connexion

Les dépôts peuvent être accédés de plusieurs façons (local, http, ssh...). Dans chaque commande, le mode d'accès est indiqué au début du chemin d'accès. Voici le code à mettre :

- *file :/// :* accès direct à un dépôt sur un disque en local
- *http :// :* accès via le protocole WebDAV à un serveur Apache utilisant le plugin Subversion
- *https :// :* comme pour *http ://* mais avec le cryptage SSL
- *svn :// :* accès via le protocole spécifique à un serveur Subversion de type *svnserve*
- *svn+ssh :// :* comme pour *svn ://* mais à travers un tunnel SSH.

Pour lancer simplement un serveur **svnserve**, utilisez la commande suivante :

svnserve -d .

Un démon sera lancé et les utilisateurs pourront se connecter par l'option *svn ://*.

C.2 Commandes courantes

- création d'un dépôt :
  # **svnadmin create** <*cheminlocal*>
- importation d'un projet :
  # **svn import** <*cheminlocal*> <*depot*> *-m "commentaire"*
- récupération d'un projet :
  # **svn checkout** <*depot*> <*cheminlocal*>
- obtenir des infos sur la copie en local :
  # **svn info**
- visualiser les états des fichiers :
  # **svn status** <*cheminlocal*>
- ajout d'un fichier :
  # **svn add** <*nomfichier*>
- suppression d'un fichier :
  # **svn delete** <*nomfichier*>
- déplacement d'un fichier :
  # **svn move** <*source*> <*destination*>
- copie d'un fichier :
  # **svn copy** <*source*> <*destination*>
- validation des modifications :
  # **svn commit** *-m "commentaire"*
- résolution d'un conflit :
  # **svn resolved** <*nomfichier*>
- mise à jour à la dernière version :
  # **svn update**

C.3 Gestion des modifications

- suivre les modifications sur le projet :
  # **svn log**
- suivre les modifications sur le projet (affichage détaillé) :
  # **svn log** *-v*
- suivre les modifications sur un intervalle de révisions :
  # **svn log** *-r <numéro> :<numéro>*
- visualiser les différences entre les fichiers modifiés localement et ceux de la dernière révision de la base :
  # **svn diff**
- visualiser les différences entre les fichiers modifiés localement et ceux d'une révision donnée de la base :
  # **svn diff** *-r <numéro>*
- visualiser les différences entre un fichier particulier modifié localement et celui d'une révision donnée de la base :
  # **svn diff** *-r <numéro> <nomfichier>*

– visualiser les différences entre deux révisions d'un fichier :
  # **svn diff** *-r <numéro> :<numéro> <nomfichier>*
– mise à jour à une révision précise :
  # **svn update** *-r <numéro>*
– affichage du contenu d'un fichier pour un révision donnée :
  # **svn cat** *-r <numéro> <nomfichier>*
– annulation des modifications effectuées localement sur un fichier :
  # **svn revert** *<nomfichier>*
– lister les fichiers d'un dépôt (affichage détaillé) :
  # **svn list** *-v*
– appliquer les modifications apportées sur un répertoire sur un autre répertoire :
  # **svn merge** *-r <numéro> :<numéro> <source> <destination>*

D TP : Le Juste Prix

Ce TP a pour objectif de mettre en application l'utilisation de Subversion pour le développement en équipe d'un projet informatique.

Le projet consiste à réaliser le jeu du juste prix : un nombre est choisi aléatoirement entre deux bornes et le joueur doit le deviner en un minimum d'essais. A chaque essai, si le nombre n'est pas le bon, on indique au joueur si celui recherché est supérieur ou inférieur à celui énoncé.

A terme, ce jeu devra être utilisé par plusieurs personnes jouant sur des ordinateurs différents. Ainsi, pour pouvoir les mettre en compétition, un fichier contenant les meilleurs scores devra être partagé et maintenu automatiquement.

Concrètement, voici les fonctionnalités qui devront être disponibles au joueur :
- saisir le nom du joueur
- jouer au jeu
- afficher le nombre de coups utilisés pour arriver à trouver le juste prix
- afficher l'historique des parties
- afficher un classement provenant d'un fichier
- afficher la position du résultat du joueur dans le classement
- sauvegarder le résultat du joueur dans le fichier
- synchroniser automatiquement les modifications apportées sur le fichier

Ce projet devra être réalisé sous forme de quatre versions contenant les fonctionnalités suivantes :
- **Version 1.0 :**
  saisir le nom du joueur
  jouer au jeu
  afficher le nombre de coups utilisés pour arriver à trouver le juste prix
  afficher l'historique des parties

- **Version 2.0 :**
  afficher un classement provenant d'un fichier
  afficher la position du résultat du joueur dans le classement

- **Version 3.0 :**
  sauvegarder le résultat du joueur dans le fichier

- **Version 4.0 :**
  synchroniser automatiquement les modifications apportées sur le fichier

Chaque version sera mise en production après passage d'une multitude de tests. La version suivante de ne sera pas auditée tant que la version précédente ne sera pas complètement validée.

Exercice 1 : Organisation et planification

Après avoir formé des équipes de 4 personnes, le premier exercice va consister à désigner un chef de projet. Cette personne sera responsable de la gestion de projet et servira d'intermédiaire avec le client.

Une fois le chef de projet désigné, vous devez réfléchir au projet, aux fonctionnalités et aux choix techniques (structure de données, algorithmes...). Vous devez aussi définir une organisation de travail (dépôt Subversion, gestion des bugs...). Enfin, vous devez planifier les tâches et les attribuer aux membres de l'équipe de projet.

Toutes ces informations devront être écrites sur papier et devront être disponibles à chaque membre du projet.

A la fin du temps attribué à cet exercice, le chef de projet présentera brièvement son plan d'action et des questions pourront être posées au client concernant les choix techniques, le besoin, etc.

Exercice 2 : Réalisation du jeu

L'exercice suivant va consister à réaliser le projet. N'oubliez pas de préparer votre dépôt Subversion et de mettre en place vos procédures de compilation (Makefile...).

Lorsqu'une version est terminée, vous pouvez contacter le client qui jouera au juste prix. Si des bugs sont trouvés lors de l'utilisation, ils devront être corrigés dans les plus brefs délais. Si plus aucun bug n'est trouvé, la version sera acceptée définitivement et la version suivante pourra être testée.

Ainsi, une version possède trois états :
- *en développement :* la version est en cours de développement.
- *livrée :* la version est livrée chez le client qui l'utilise.
- *acceptée :* le client a utilisé la version et n'a plus détecté de problèmes d'utilisation. Lorsqu'une version est acceptée, la version suivante peut être livrée si elle est terminée.

Mémo C++ :
- *Génération aléatoire d'un nombre :*
  exécuter l'instruction *srand(time(NULL)) ;* au début du programme une seule fois.
  pour générer un nombre entre *a* et *b*, voici la formule : *nombre = rand()%(b-a) + a ;*

- *Lecture d'une chaîne de caractère :*
  instruction *getline(<flux d'entrée>,<string>)*. Par exemple, pour lire au clavier, on fera : *getline(cin, machaine)*.
- Sites internet utiles :
  `http://www.cppreference.com`
  `http://c.developpez.com/faq/cpp/`