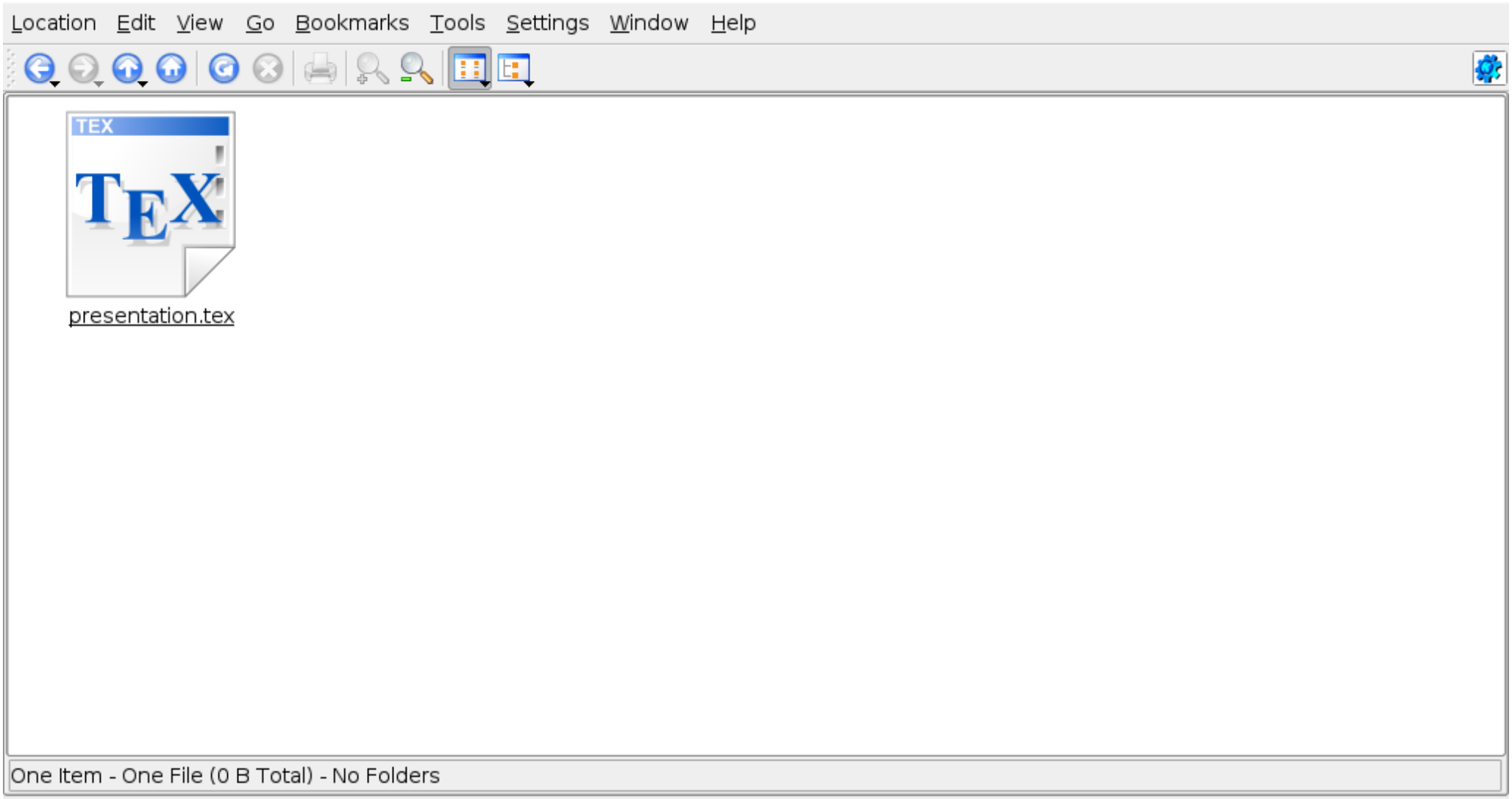
Gestion de version
Subversion

Thomas Petazzoni
1er octobre 2010

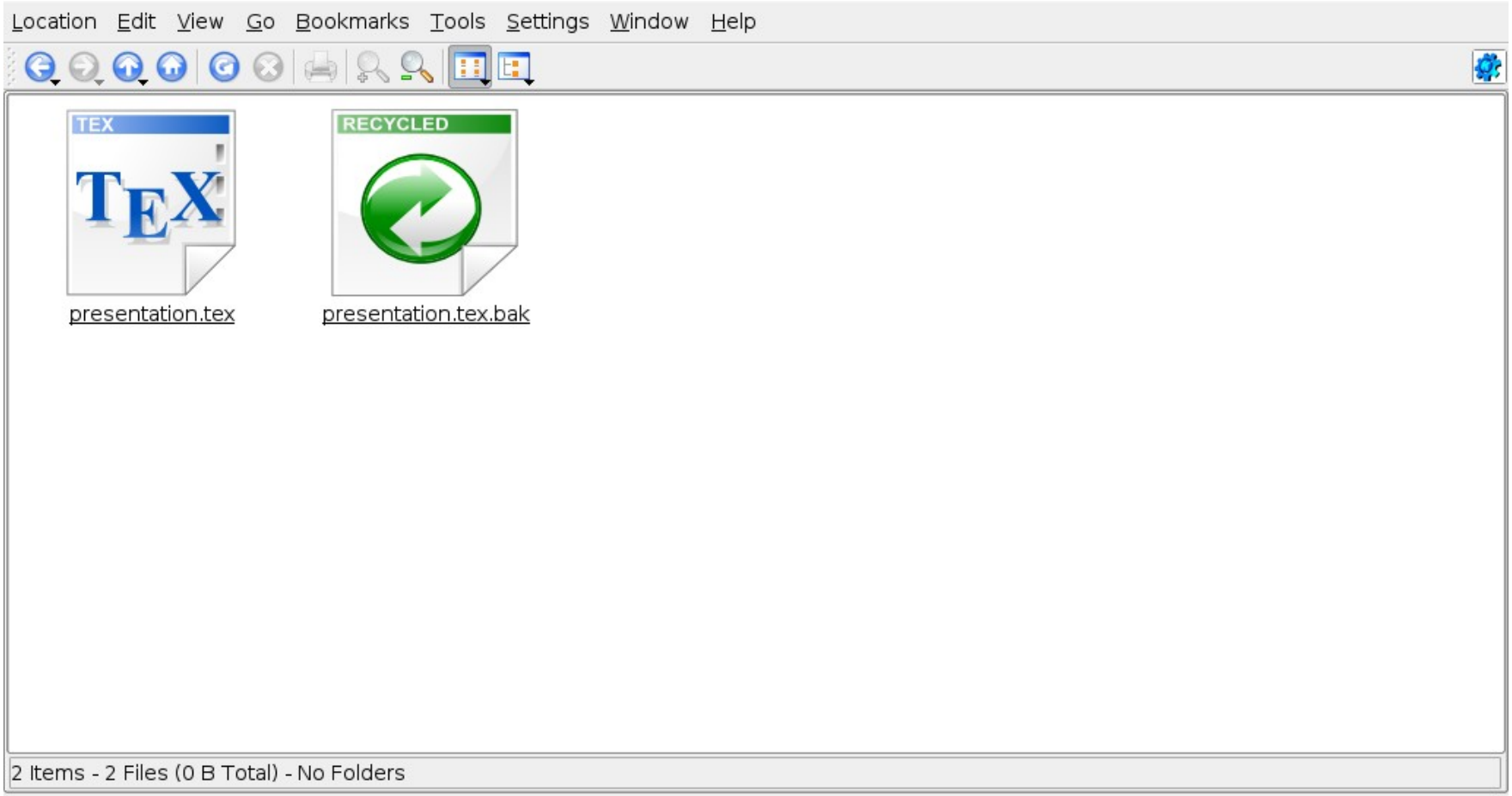
Petite histoire vraie

Il était une fois un document...



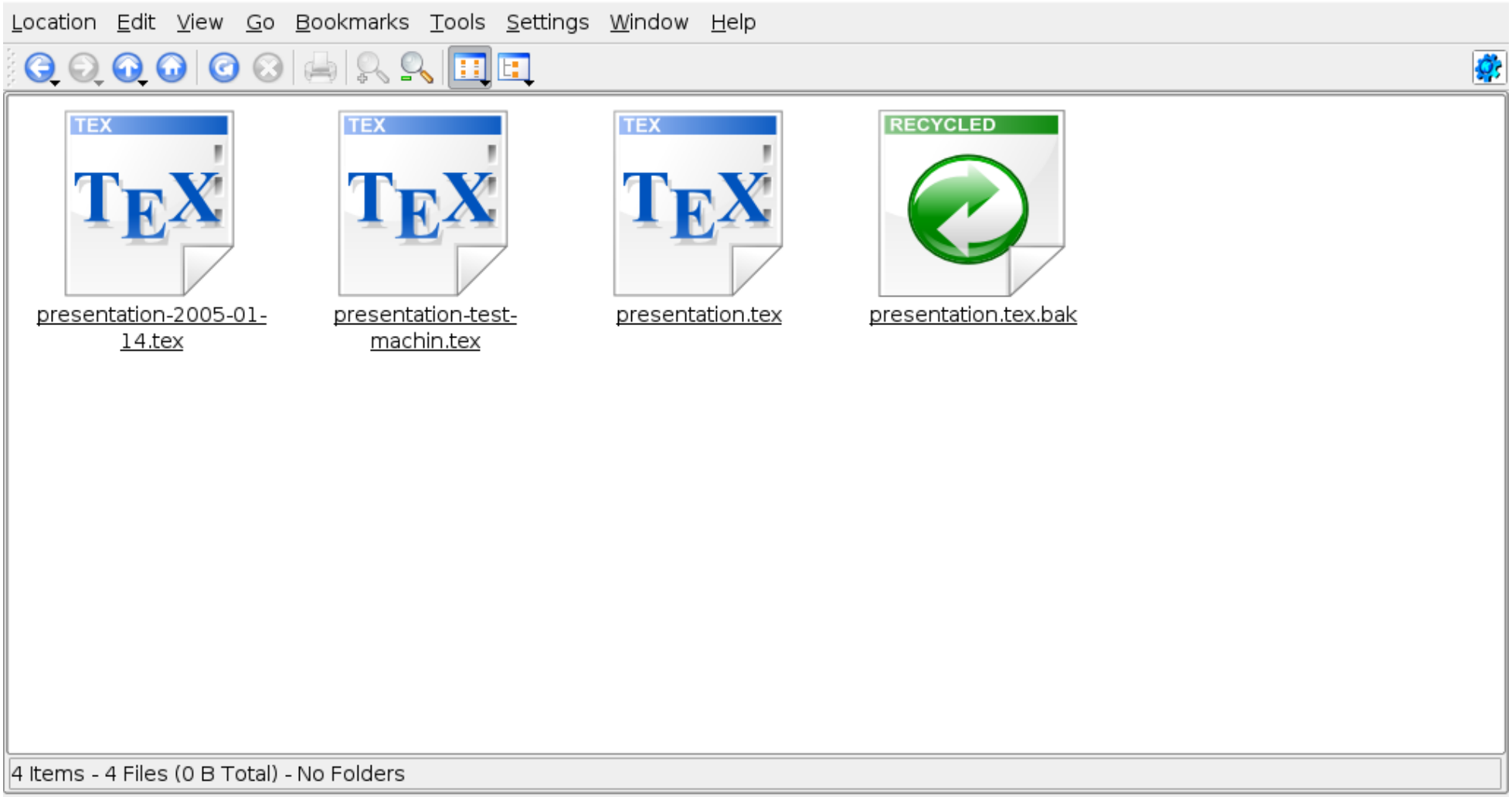
Petite histoire vraie

Tiens, faisons une sauvegarde histoire de ne pas perdre mon travail en cas d'erreur



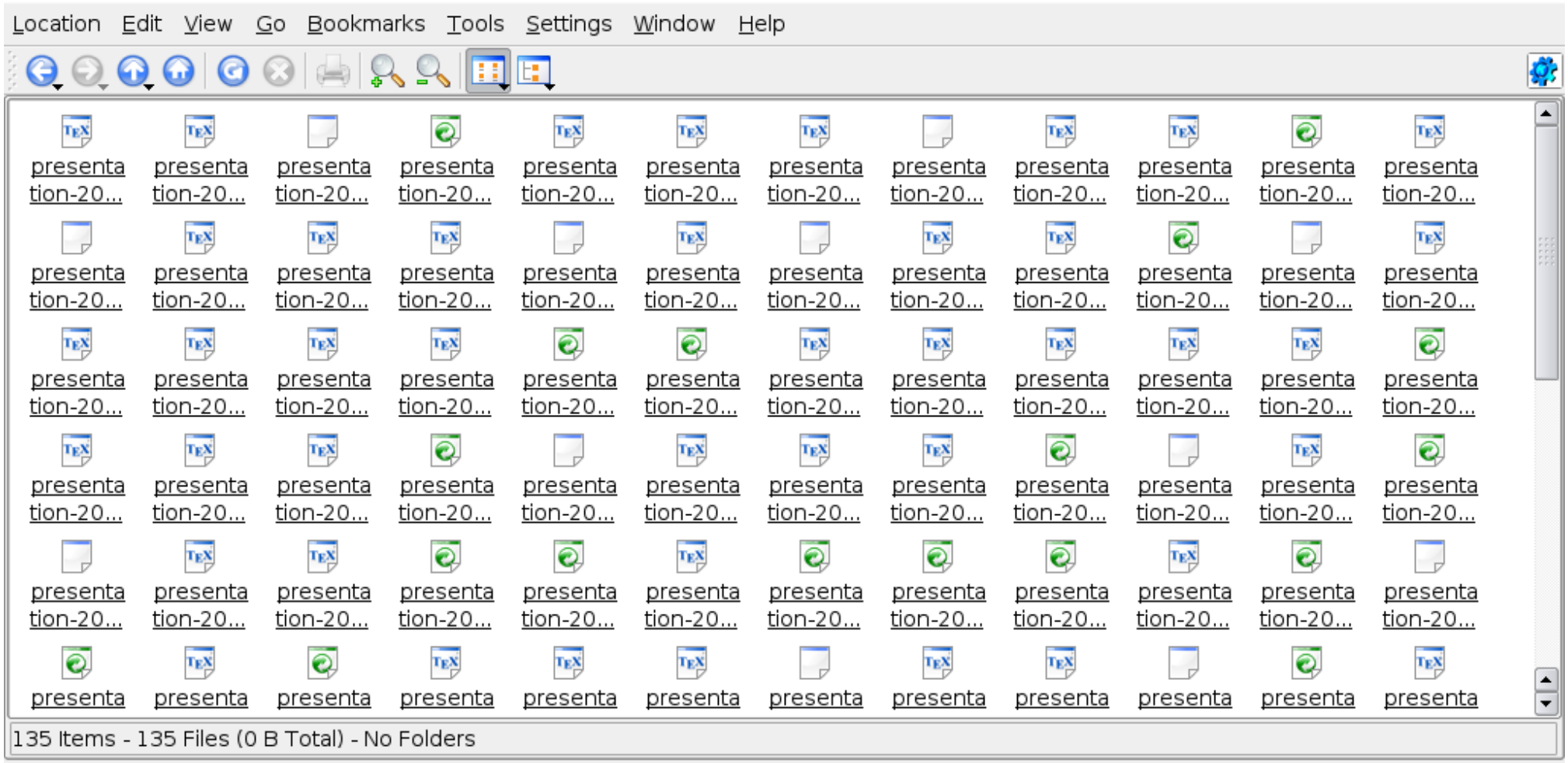
Petite histoire vraie

Maintenant, je veux tester un truc,
faisons une autre sauvegarde



Petite histoire vraie

Euh, j'en étais à quelle version déjà ?



Ça ne peut pas fonctionner...

- Explosion du nombre de fichiers à gérer
- Impossible de savoir quelle est la dernière version
- Impossible de maintenir un historique cohérent et utilisable
- Ingérable pour un développeur seul, totalement inenvisageable dès que le nombre de développeurs du projet dépasse un

- La gestion de version a pour objectif de résoudre ces problèmes
  - Fournir un outil pour la gestion du code source, de la documentation et de tout élément de type fichier lié à un projet
  - Est un sous-ensemble de la « gestion de configuration » qui consiste à s'assurer de la reproductibilité de la construction d'un logiciel

Gestion de version

- Activité consistant à maintenir l'intégralité de l'historique des fichiers d'un projet
- Toutes les versions de chaque fichier sont enregistrées
- Permet notamment de :
  - Revenir en arrière cas d'erreur
  - Relire les modifications introduites entre deux versions différentes
  - Travailler à plusieurs sur un même projet
  - Gérer des branches de développement parallèles
  - Estampiller des versions précises du projet

Gestion de version

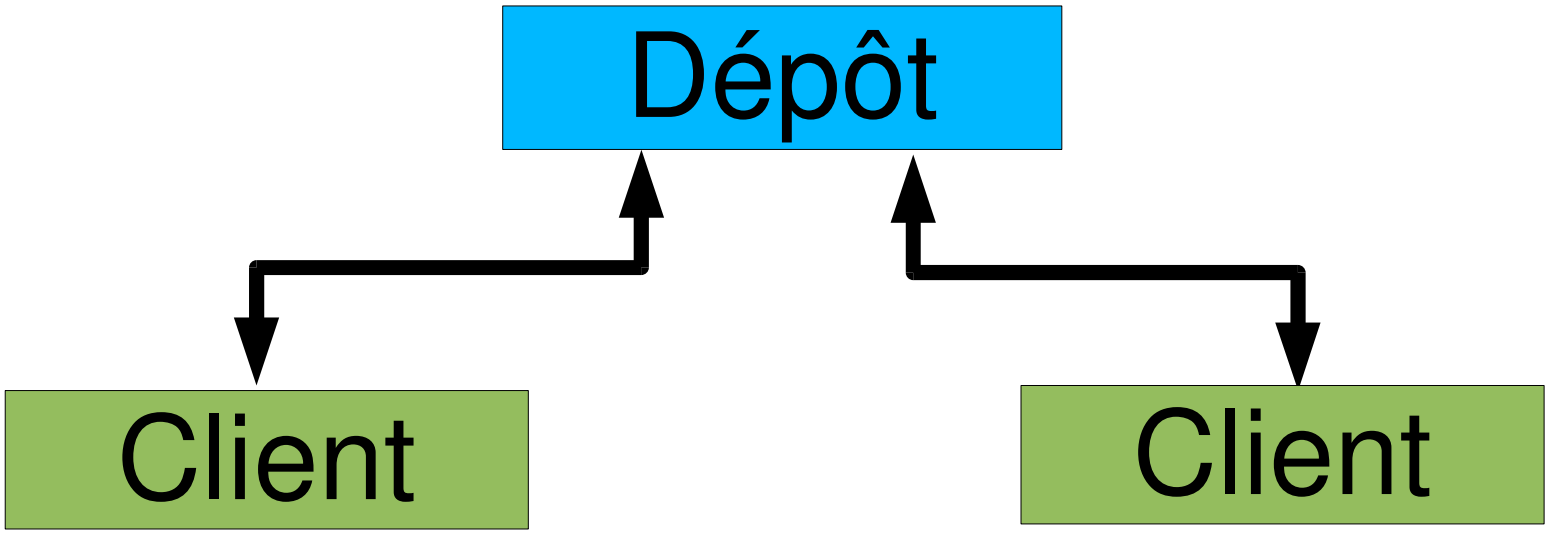
- Utile pour versionner :
  - Code source, évidemment
  - Documentation, chaînes de traduction, scripts de compilation
  - Les données binaires peuvent être introduites, mais les fonctionnalités seront réduites (pas de diff). Les fichiers en texte simple sont à privilégier
- Les fichiers générés ne doivent pas être versionnés !
- Indispensable dès que l'on travaille à plusieurs
  - Un des outils au coeur du modèle de développement des Logiciels Libres
- Très pratique même lorsque l'on travaille seul sur un projet (historique, développement en parallèle sur plusieurs aspects, etc.)

Outils et approches

- De nombreux outils de gestion de version
  - Propriétaires: Perforce, Synergy, StarTeam, BitKeeper, ClearCase
  - Libres: CVS, Subversion, Git, Mercurial, Arch, Monotone
- Parmi les solutions libres, deux approches distinctes
  - L'approche centralisée, implémentée par CVS et Subversion
  - L'approche décentralisée, implémentée dans Git, Mercurial, Arch ou Monotone

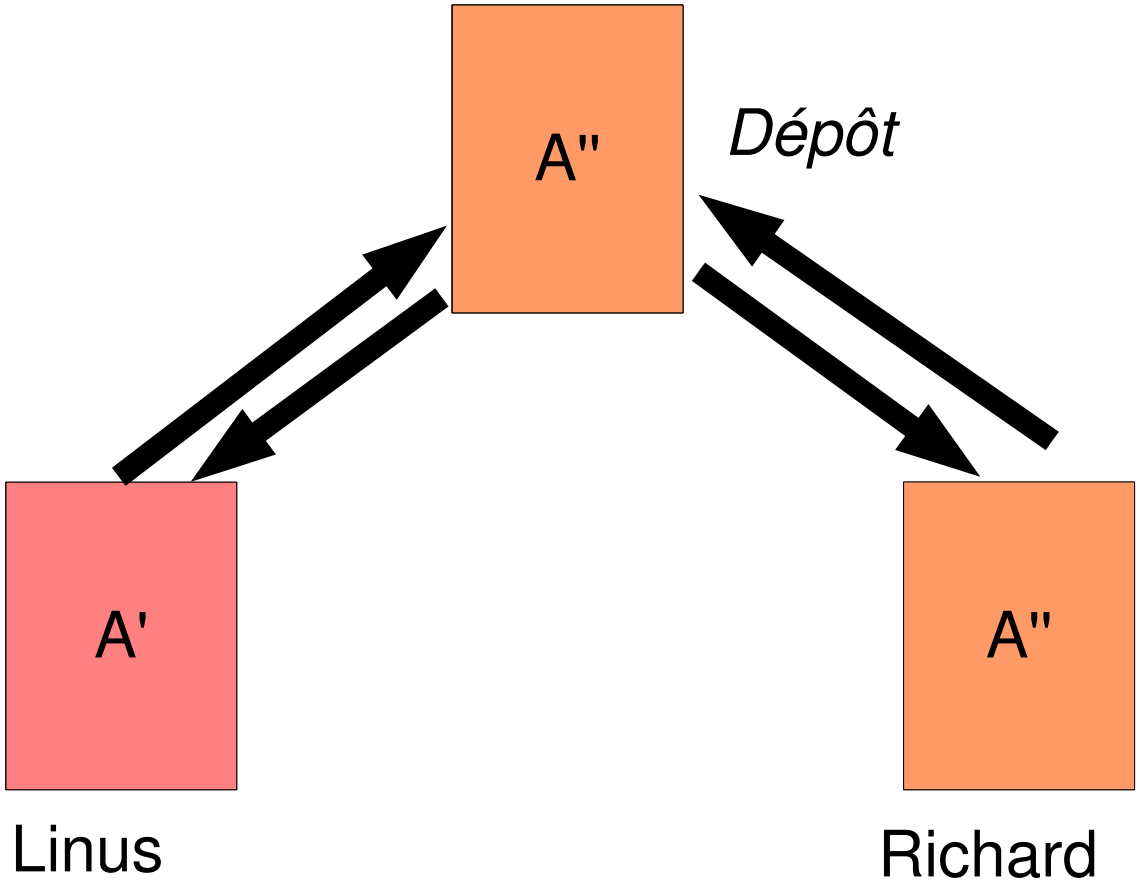
Approche centralisée

- L'approche centralisée est basée sur la présence d'un serveur central appelé **dépôt** ou **repository**
- Il contient l'ensemble de l'historique et fait référence pour tous les participants du projet



Gestion des conflits

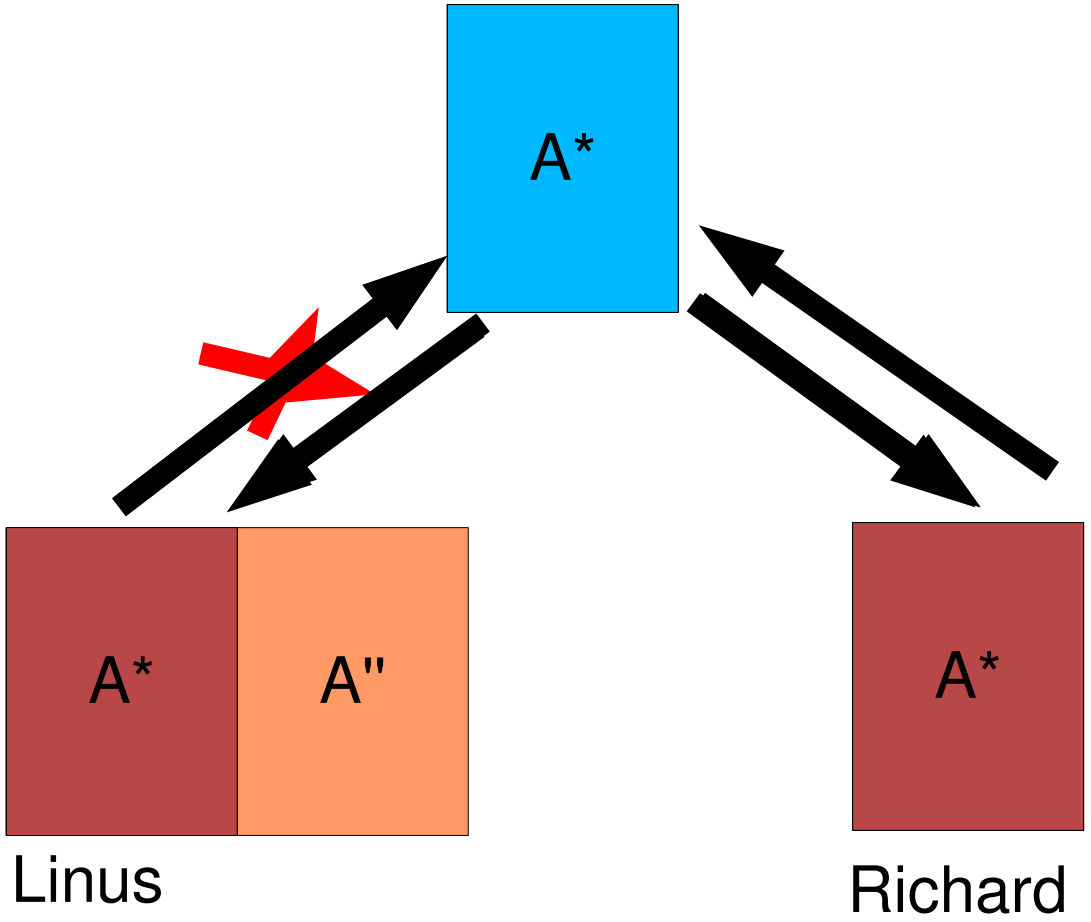
- Deux personnes travaillent en même temps sur un même fichier
- Lors de l'envoi des changements vers le serveur, un conflit peut se produire si les modifications réalisées ne sont pas « compatibles »



Gestion des conflits

- Deux solutions pour résoudre ce problème
  - L'approche par **verrouillage** : on empêche deux développeurs de travailler sur le même fichier
  - L'approche par **fusion** : avant d'envoyer ses modifications vers le serveur, le deuxième développeur a la responsabilité de les fusionner avec les modifications du premier
- Le modèle traditionnel dans les outils libres est celui de la fusion, l'approche par verrouillage étant ingérable
- Mais Subversion implémente également l'approche par verrouillage

Gestion des conflits par fusion



Subversion

- Logiciel Libre
- Système de gestion de version centralisée libre le plus utilisé aujourd'hui
- Utilisé par de gros projets (KDE, Apache, Eclipse, etc.)
- Très similaire à CVS, aussi bien au niveau des principes que des commandes
  - *« CVS, but without the bugs and misfeatures »*
- Site officiel: http://subversion.tigris.org/
- Excellente documentation « Subversion book », disponible en ligne : http://svnbook.red-bean.com/

Utilisation de Subversion

- En ligne de commande
  - Client Subversion sous la forme de la commande `svn`
  - `svn help` donne la liste des commandes
    - `add, blame, cat, changelist, checkout, cleanup, commit, copy, delete, diff, export, help, import, info, list, lock, log, merge, mergeinfo, mkdir, move, propdel, propedit, propget, proplist, propset, resolve, resolved, revert, status, switch, unlock, update`
  - `svn help command` donne l'aide sur une commande
- Interface graphique
  - Clients lourds: TortoiseSVN (intégration Windows), Subclipse (intégration Eclipse), esvn (client Qt), kdesvn (intégration à KDE), rapidsvn, etc.
  - Web, visualisation seulement: viewvc, websvn, Trac, etc.

Création de la copie de travail

- La première étape est de créer une **copie de travail** (*working copy*)
  - Copie sur la machine du développeur d'un arbre complet des sources du projet
  - C'est sur cette copie que le développeur va travailler, de façon naturelle
- `svn checkout adresse-depot`
  - Créé une copie de travail depuis le dépôt adresse-depot, dans un répertoire dont le nom est le dernier élément de adresse-depot
- `svn checkout adresse-depot repertoire-local`
  - Idem, sauf que le nom du répertoire où sera créée la copie de travail est spécifié
- `svn co adresse-depot`

Différents types de dépôt

- Dépôts locaux
  - `svn co file:///var/svn/project1/trunk/`
- Dépôts stockés sur un serveur Web, accédés au travers de WebDAV (HTTP lecture/écriture)
  - `svn co http://www.serveur.com/svn/project1/trunk/`
  - `svn co https://www.serveur.com/svn/project1/trunk/`
- Dépôts stockés sur un serveur, accédé au travers du shell sécurisé ssh
  - `svn co svn+ssh://user@serveur.com/svn/project1/trunk/`
- Dépôts stockés à l'aide d'un serveur dédié, *svnserve*
  - `svn co svn://user@serveur.com/svn/project1/trunk/`

Utilisation simple

- Création d'une copie de travail
  - À faire une fois pour toutes
- Modification des fichiers de la copie de travail
  - On peut modifier directement les fichiers: Subversion connaît le contenu de la version d'origine et est capable de calculer les modifications apportées
- Envoi des améliorations vers le dépôt central : commit
  - `svn commit`
  - Lance un éditeur pour rédiger le message qui accompagnera le commit dans l'historique
- Récupérer les améliorations apportées par les autres développeurs : update
  - `svn update`

Exemple

```
$ svn co file:///tmp/repo/trunk project
A    project/test.c
A    project/Makefile
A    project/README
Checked out revision 2.
$ cd project/
$ vi Makefile
$ svn commit
Sending        Makefile
Transmitting file data .
Committed revision 3.
$ svn up
U    README
Updated to revision 4.
```

Gestion des fichiers

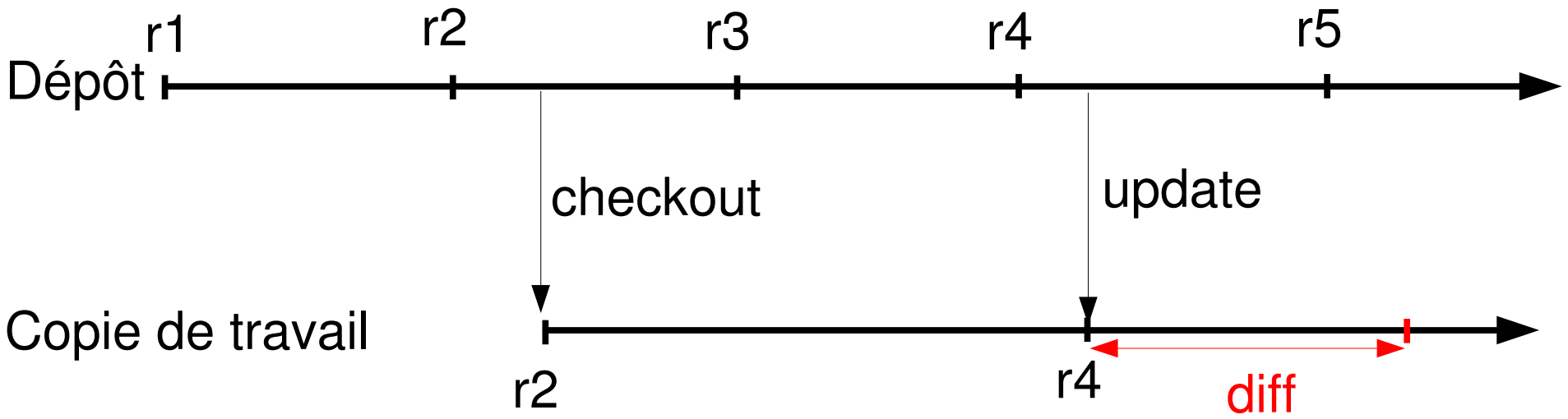
- Ajouter un fichier ou répertoire au projet : **add**
  - Créer le fichier ou répertoire, avec ou sans contenu
  - `svn add nom-du-fichier`
  - Pour un répertoire, `svn mkdir` va créer le répertoire et l'ajouter en une seule commande
- Supprimer un fichier ou répertoire du projet : **remove**
  - `svn remove nom-du-fichier`
- Renommer un fichier ou répertoire : **move**
  - `svn move ancien-nom nouveau-nom`
- Ces commandes n'opèrent que sur la copie de travail. Il faut *committer* pour propager les modifications au dépôt central.

Exemple

```
$ echo "This is the doc" > HOWTO
$ svn add HOWTO
A          HOWTO
$ svn mv README README.old
A          README.old
D          README
$ svn rm Makefile
D          Makefile
$ svn commit
Adding           HOWTO
Deleting         Makefile
Deleting         README
Adding           README.old
Transmitting file data .
Committed revision 5.
```

Observer l'état de la copie de travail

- Commande `status`
  - Donne une information d'état pour chaque fichier et répertoire
  - A : ajouté, D : supprimé, M : modifié, R : remplacé, C : en conflit, ? : pas en gestion de version, ! : fichier manquant, etc.
- Commande `diff`
  - Affiche les différences entre la version courante d'un fichier de la copie de travail et la version d'origine sur laquelle sont basées les modifications



Exemple

```
$ svn status
$ echo "Test" >> HOWTO
$ svn status
M       HOWTO
$ rm README.old
$ svn status
M       HOWTO
!       README.old
$ touch toto.c
$ svn status
?       toto.c
M       HOWTO
!       README.old
$ svn add toto.c
A         toto.c
$ svn status
M       HOWTO
A       toto.c
!       README.old
```

Exemple

```
$ svn diff
Index: HOWTO
===================================================================
--- HOWTO  (revision 5)
+++ HOWTO  (working copy)
@@ -1 +1,2 @@
 This is the documentation
+Test
Index: toto.c
===================================================================
```

Utiliser l'historique

- Consulter les messages de commit
  - `svn log`
  - `svn log fichier.c`
- Voir les changements entre deux révisions
  - `svn diff -r 12:13 fichier.c`
  - Avec Subversion, le numéro de révision est un identifiant global et unique pour l'état du dépôt. Chaque changement sur un fichier incrémente ce numéro global. Il n'y a pas de version par fichier.
- Voir qui a écrit quelle ligne
  - `svn blame fichier`
- Une interface graphique (client lourd ou Web) sera d'une grande utilité pour la navigation dans l'historique

Gestion des conflits

- Après modification d'un fichier, commit :

```
$ svn commit -m "Improve test.c"
Sending        test.c
svn: Commit failed (details follow):
svn: File '/trunk/test.c' is out of date
```

- Le fichier `test.c` n'est pas à jour avec le dépôt, mettons-le à jour :

```
$ svn update
Conflict discovered in 'test.c'.
Select: (p) postpone, (df) diff-full, (e) edit,
        (h) help for more options: p
C    test.c
Updated to revision 7.
```

Gestion des conflits

- On a maintenant 4 fichiers pour test.c
  - `test.c` lui même, notre fichier, avec des marqueurs entourant le conflit
  - `test.c.mine`, notre version du fichier, telle qu'on l'avait créée
  - `test.c.rX`, le fichier tel qu'il était avant que nous n'y apportions des modifications
  - `test.c.rY`, le fichier, tel que committé par l'autre développeur
  - Avec `rX < rY`

Gestion des conflits

```
int main(void) {
<<<<<<< .mine
    printf("Bonjour monde\n");
=======
    printf("Hello World\n");
>>>>>>> .r7
    return 0;
}
```

test.c

```
int main(void) {
    printf("Bonjour monde\n");
    return 0;
}
```

test.c.mine

```
int main(void) {
    return 0;
}
```

test.c.rX

```
int main(void) {
    printf("Hello World\n");
    return 0;
}
```

test.c.rY

Résolution du conflit

- Tant que le conflit n'est pas résolu, Subversion ne nous autorise pas à committer

```
$ svn commit
svn: Commit failed (details follow):
svn: Aborting commit: '/tmp/project/test.c' remains in
conflict
$ svn status
?       test.c.r5
?       test.c.r7
?       test.c.mine
C       test.c
```

- Mise à contribution du cerveau du développeur, qui doit savoir comment résoudre le conflit
  - Faut-il garder « Hello World » ? Faut-il garder « Bonjour monde » ? Faut-il marquer « Hello World / Bonjour monde » ? Autre chose ?
  - La résolution se fait directement dans test.c, en s'aidant éventuellement des autres fichiers

Résolution du conflit

- Une fois le conflit résolu par le développeur, il faut l'indiquer à Subversion. Il va supprimer le fichier .mine, .rX et .rY.

```
$ svn resolved test.c
Resolved conflicted state of 'test.c'
$ svn status
M       test.c
$ svn diff
Index: test.c
===================================================================
--- test.c  (revision 7)
+++ test.c  (working copy)
@@ -1,4 +1,4 @@
 int main(void) {
-    printf("Hello World\n");
+    printf("Bonjour monde\n");
     return 0;
 }
```

- On peut maintenant committer la nouvelle version

Initialisation d'un dépôt

- Pour créer un dépôt Subversion, utilisation de la commande svnadmin
  - `svnadmin create /chemin/vers/depot/`
- Une fois le dépôt créé, il est accédé au travers d'une des méthodes décrites précédemment (fichier, http, svnserve, ssh)
- Si un projet existant doit être importé, utilisation de la commande svn import
  - `svn import . http://serveur/svn/project/`
  - L'arbre du projet doit être propre: pas de fichiers générés, de fichiers temporaires, etc.

Organisation du dépôt

- Subversion est en fait un système de fichiers versionné
  - Pas d'organisation prédéfinie pour les projets, les versions, etc.
  - On s'organise comme on veut, au travers de conventions
- Une organisation très utilisée :
  - À la racine du dépôt, un répertoire par projet
  - Dans chaque répertoire de projet
    - Un répertoire trunk/ qui contient la version de développement principale du projet
    - Un répertoire tags/ qui contient les tags du projet
    - Un répertoire branches/ qui contient les branches du projet
- Une autre organisation consiste à avoir les répertoires trunk, tags et branches à la racine, puis dans chacun de ces répertoires, un répertoire par projet
- Autre solution: un dépôt par projet

Organisation du dépôt

- /
  - project1
    - trunk
    - tags
    - branches
  - project2
    - trunk
    - tags
    - branches

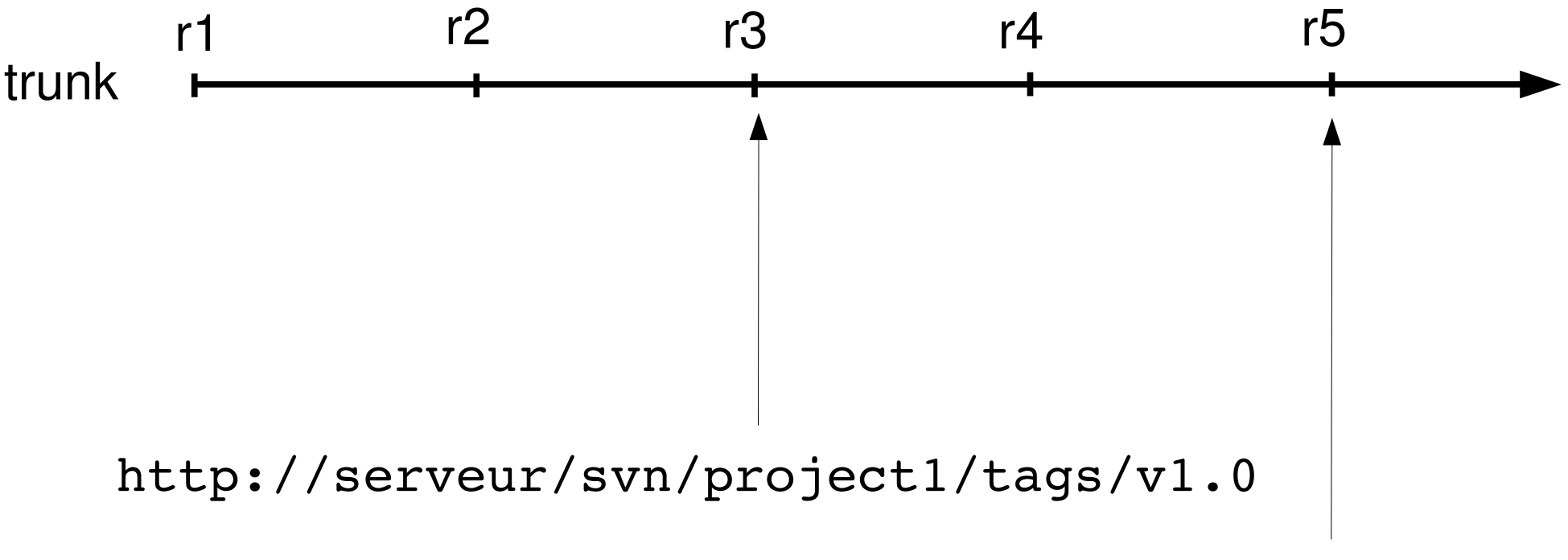
- /
  - trunk
    - project1
    - project2
  - tags
    - project1
    - project2
  - branches
    - project1
    - project2

Tags

- Un tag est un identifiant unique pour une version donnée d'un projet
- Avec Subversion, chaque révision est un tag puisqu'elle identifie de manière unique l'état du dépôt
- En pratique, on souhaite en général associer un nom à certaines versions particulières d'un projet (1.0, 1.1, 1.1-beta42, etc.)
- Cela permet de revenir facilement à une version identifiée
- Pour créer un tag, il suffit de copier l'état actuel d'une version de développement :
  - `svn copy http://serveur/svn/project/trunk http://serveur/svn/project/tags/v1.0/`
- La copie est optimisée par Subversion: les fichiers ne sont pas dupliqués. Un tag ne coûte rien !

Tags

```
svn copy
http://serveur/svn/project1/trunk
http://serveur/svn/project1/tags/v1.0
```

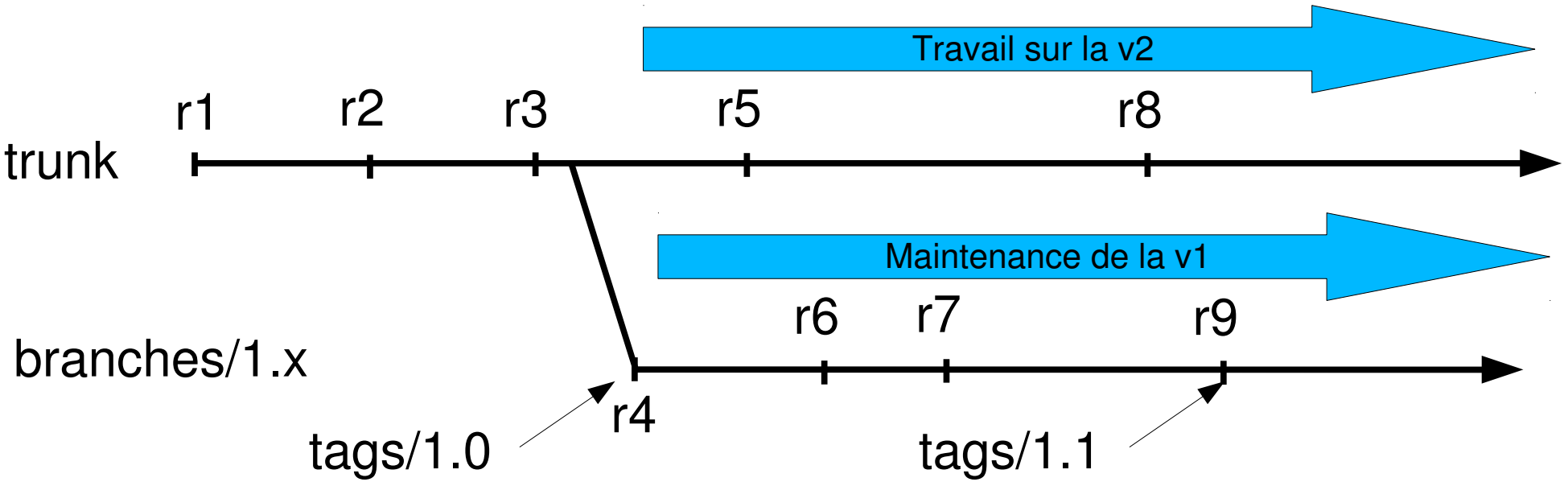


Branches

- Les branches permettent de créer des espaces de travail distincts de la branche de développement principale (*trunk*)
- Développement possible en parallèle
- Créer une branche
  - `svn copy http://serveur/svn/project1/trunk http://serveur/svn/project1/branches/dev-featureX`
- La branche est créée avec l'état courant du trunk. Pour utiliser la branche, il faut en créer une copie de travail
  - `svn co http://serveur/svn/project1/branches/dev-featureX`

Branche « de release »

- Pour réaliser la maintenance d'une version tout en poursuivant le développement de la prochaine version



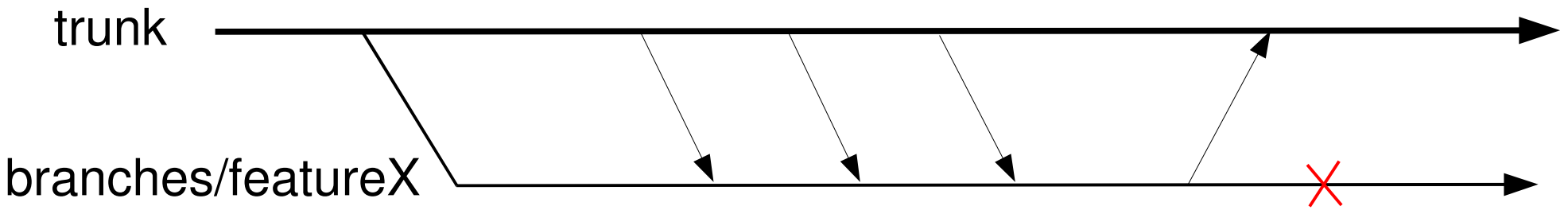

```
svn copy http://serveur/svn/project1/trunk
http://serveur/svn/project1/branches/1.x
http://serveur/svn/project1/tags/1.0
```

Échanges entre branches

- Un correctif arrive dans trunk et est intéressant pour la version 1.x, ou à l'inverse, un correctif dans 1.x doit être reporté dans trunk. Comment procéder ?
- Solution 1 – pas terrible
  - Dans une copie de travail du trunk: `svn diff -r 12:13 > monpatch`
  - Dans une copie de travail de la branche: `cat monpatch | patch -p0`
  - Ne gère pas les suppressions/renommage de fichiers/répertoires
- Solution 2 – bien
  - Dans une copie de travail de la branche:
    `svn merge -r 12:13 http://serveur/svn/project/trunk/`
- Processus de « *cherry-picking* »

Branche de développement

- Pour travailler sur une nouvelle fonctionnalité majeure, ou un refactoring important, sans gêner le développement sur la branche principale
- Procédure
  - Création de la branche de développement
  - *Merge* répété depuis la branche principale vers la branche de développement, pour que celle-ci reste à jour avec les autres développements en cours. Subversion >= 1.5 significativement amélioré à ce niveau.
  - À la fin du développement de la fonctionnalité, merge de la branche de développement vers la branche principale



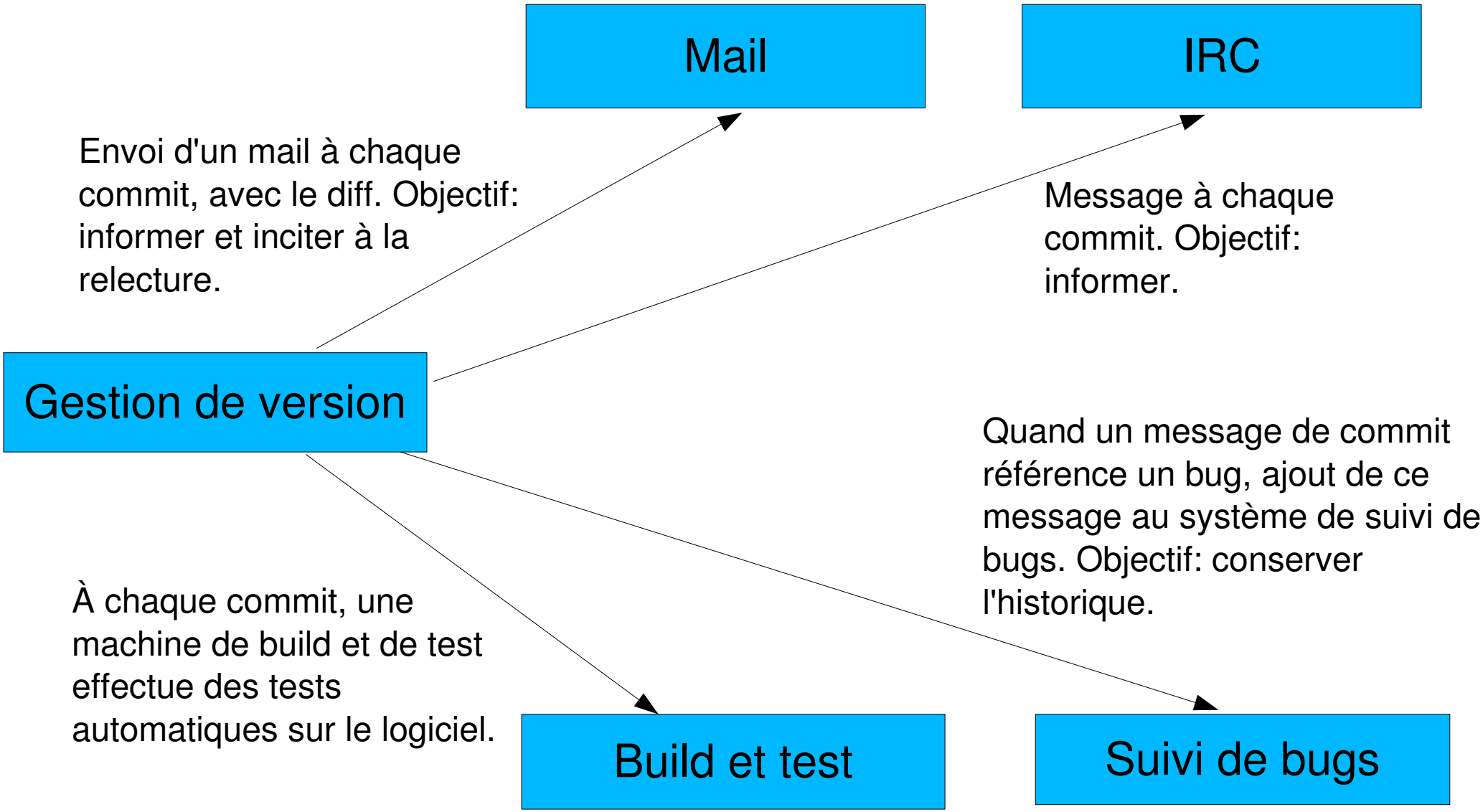
Propriétés

- Fonctionnalité de Subversion permettant d'ajouter des méta-données aux fichiers ou répertoires
- Manipulation par les commandes `propdel`, `propedit`, `propget` et `propset`
- Deux propriétés importantes
  - `svn:ignore` sur un répertoire. Liste les fichiers et sous-répertoires qui ne doivent pas être considérés par Subversion. Utile pour marquer les fichiers générés par la compilation, par exemple. Des wildcards comme `*` sont acceptés
  - `svn:mime-type` pour indiquer le type MIME d'un fichier versionné. Utile pour les fichiers binaires qui doivent être traités spécialement par Subversion. Dans ce cas, la valeur de la propriété svn:mime-type doit être `application/octet-stream`

Règles d'utilisation

- L'outil de gestion de version n'est qu'un outil
- Des règles d'utilisation doivent être prises par l'équipe pour utiliser efficacement cet outil
- Quelques règles « courantes »
  - Committer souvent.
  - Un commit doit représenter un changement logique et un seul. On ne mélange pas une correction de bug et une nouvelle fonctionnalité.
  - Le message de commit doit donner le plus d'informations possible sur ce qui est réalisé (numéro de bug, détail de la nouvelle fonctionnalité, etc.)
  - On ne committe pas dans un tag
  - Des responsables identifiés pour chaque branche de développement, responsables du merge répété

Connexion avec les autres outils



Gestion de version distribuée

- Depuis quelques années, nouvelle génération d'outils de gestion de version
- Gestion de version distribuée plutôt que centralisée
- Principes
  - Pas de dépôt techniquement central
  - Tous les participants peuvent créer des branches, lancer des développements en parallèle, sans nécessiter d'accord d'une autorité
  - Des fonctionnalités de branching – merging évoluées
- De plus en plus utilisé dans les projets Logiciels Libres
- Exemples: Git, Mercurial