Outils de développement - 3A STI

Jean-Francois Lalande - May 2016 - Version 1

Ce cours présente le b.a.-ba des outils d'aide au développement. Il s'agit d'automatiser les processus de compilation, de gestion de versions.

Plan du module

1 Introduction

1.1 De l'automatisation de la compilation

Dans ce qui suit, on présente des benchmarks de performance du processus de compilation datant de 2010. Ces benchmarks ont été réalisés chez Amadeus qui compte plus de 10 000 collaborateurs et qui offre des service notamment pour la gestion des vols pour les compagnies aériennes (entre autre) [GH].

Machine de test: 4 processeurs à 6 coeurs à 2.4 GHz (opteron), 32Go de RAM

Quelques chiffres représentatifs pour le code:

Nb Lines	366 452
Nb functions	15 227
Blank lines	55 878
Nb structs	3483

Temps de compilation + link: 6780s soit presque 2h (avec 4 processus en permanence)

Temps de compilation hacké (concaténation par groupe de 25 des sources): 2854s = 47 min

Temps de re-compilation pour 1 fichier touché: 1 min (calcul du makefile) + quelques secondes (compilation) + 7 min (link)

1.2 De la qualité du code

Sans faire un cours sur la qualité logicielle, des règles de bon sens s'appliquent pour que le code soit compréhensible et maintenables. Toujours en prenant l'exemple du code Amadeus, des indicateurs simples permettent d'apprécier la qualité du code:

Type	Count	Percent	Quality Notice
1	61031	39.22	Physical line length > 80 characters
2	905	0.58	Function name length > 32 characters
4	21	0.01	Assignment "=" within "if" statement
5	9	0.01	Assignment "=" within "while" statement
13	100	0.06	"switch" statement does not have a "default"
14	88	0.06	"case" conditions do not equal "break"
17	5971	3.84	Function comment content less than 10.0%
23	552	0.35	"?" ternary operator identified
29	184	0.12	Number of function parameters > 6
43	497	0.32	Keyword "continue" has been identified
55	1180	0.76	Scope level exceeds the defined limit of 7
107	32	0.02	The closing brace is not on a standalone line

2 Cmake

Références importantes: Cmake

2.1 Les vénérables automake et autoconf

La suite d'outils la plus utilisée pour la génération de la compilation et du fameux **./configure**; **make**; **make install** est la suite des autotools:

- automake: permet d'écrire les squelettes de makefiles
- autoconf: gère les dépendances et génère les makefiles depuis le squelette
- libtool: gestion de création de bibliothèques

La difficulté d'utilisation de ces outils réside en partie dans les langages utilisés, *m4* pour configure.ac utilisé par autoconf, une syntaxe proche du makefile pour makefile.ac utilisé par automatke, tous ces outils étant écrits en *perl*.

La compréhension des messages d'erreur, la faible évolutivité de la suite d'outils la pousse petit à petit vers des chaines de compilation plus évoluées.

Les objectifs de cmake sont donc:

- unifier la syntaxe dans un unique fichier
- unifier les commandes dans un seul outil **cmake**
- augmenter la vitesse d'exécution des opérations de gestion de compilation
- augmenter l'intéropérabilité, par exemple avec kdevelop ou Visual Studio

2.2 CMakeLists.txt

On définit tout d'abord un nom de projet à l'aide de la directive **PROJECT**. Des dépendances peuvent être définies. Il s'agit de bibliothèques du système qui ne sont pas incluses dans la glibc. Dans l'exemple suivant, il s'agit de la librairie zlib:

```
PROJECT(jfl)
FIND_PACKAGE(ZLIB REQUIRED)
```

Le fait d'utiliser zlib impose de passer le répertoire contenant les entêtes au compilateur (gcc). Cela se fait en utilisant la directive **INCLUDE_DIRECTORIES**:

```
INCLUDE_DIRECTORIES(${ZLIB_INCLUDE_DIR})
```

On déclare ensuite l'exécutable à créer et les sources afférentes:

```
ADD_EXECUTABLE(jfl test.c)
```

Nous obtenons, au final:

```
PROJECT(jfl)
FIND_PACKAGE(ZLIB REQUIRED)
INCLUDE_DIRECTORIES(${ZLIB_INCLUDE_DIR})
ADD_EXECUTABLE(jfl test.c)
```

2.2.1 Construction de la chaine de compilation

L'appel à **cmake** permet de générer le Makefile. Il faut préciser le répertoire dans lequel **cmake** trouvera les sources et le fichier CMakeFiles.txt. La suite de la compilation se déroule classiquement en appelant **make**.

```
jf@lalande:cours_outils/codes> cmake .
-- Configuring done Generating done
-- Build files have been written to: .../codes
jf@lalande:cours_outils/codes> make
[100%] Building C object CMakeFiles/jfl.dir/test.o
Linking C executable jfl
[100%] Built target jfl
jf@lalande:cours_outils/codes> ./jfl
HelloWorld !
```

La phase de compilation est silencieuse par défaut. Pour activer les traces il suffit de modifier la variable d'environnement **VERBOSE** à 1.

```
jf@lalande:cours_outils/codes> export VERBOSE=1
[100%] Building C object CMakeFiles/jfl.dir/test.o
/usr/bin/gcc    -o CMakeFiles/jfl.dir/test.o   -c test.c
Linking C executable jfl
/usr/bin/cmake -E cmake_link_script CMakeFiles/jfl.dir/link.txt --verbose=1
/usr/bin/gcc     CMakeFiles/jfl.dir/test.o  -o jfl -rdynamic
```

2.2.2 Les directives SET, IF

La directive **SET** permet de créer des variables réutilisables dans d'autres directives:

```
SET(jfl_src main.c truc.c machin.c)
ADD_EXECUTABLE(jfl ${jfl_src})
```

La variable peut être liée à l'environnement auquel cas, la variable d'environnement est modifiée:

```
set(ENV{PATH} /home/jf)
```

La directive **IF** permet d'introduire des choix conditionnels:

```
IF(ZLIB_FOUND)
  MESSAGE(STATUS "ZLIB présent")
  INCLUDE_DIRECTORIES(${ZLIB_INCLUDE_DIR})
ELSE(ZLIB_FOUND)
  MESSAGE(STATUS "ZLIB absent")
ENDIF(ZLIB_FOUND)
```

La directive **IF** permet de tester de nombreuses choses: le positionnement d'une constante ou d'expressions logiques de constantes, l'existence de fichiers ou répertoires, la comparaison d'entiers ou de chaines.

2.2.3 La construction de librairies

La construction d'une librairie partagée se fait de la même manière qu'un exécutable, en utilisant la directive ADD_LIBRARY. Si l'on place les sources de la librairie dans un sous répertoire, il faut alors ajouter un nouveau fichier CMakeLists.txt dans ce sous-répertoire qui contiendra les directives de construction de cette librairie.

```
cmake_minimum_required(VERSION 2.6)
PROJECT(malib)
SET(lib_src malib.c)
ADD_LIBRARY(malib SHARED ${lib_src})
```

Dans le fichier principal du projet, il faut alors préciser que **cmake** doit évaluer le fichier CMakeLists.txt dans le sous répertoire et l'informer que l'exécutable à produire dépend de la librairie construite dans le sous répertoire:

```
ADD_SUBDIRECTORY(malib)
TARGET_LINK_LIBRARIES(jfl malib ${ZLIB_LIBRARY})
```

Si la librairie est partagée, elle est placée dans le répertoire SHARED:

```
jf@lalande:cours_outils/codes> rm SHARED/libmalib.so
jf@lalande:cours_outils/codes> ./jfl
./jfl: error while loading shared libraries: libmalib.so: cannot open
shared object file: No such file or directory
```

2.2.4 Paramétrer la compilation

Commend dans l'outil d'autotools au travers du script **./configure**, il est possible de passer des options au générateur afin d'activer une option lors de la compilation. On utilise pour se faire la directive **OPTION**:

```
OPTION(WITH_GUI "Compiler l'interface graphique" OFF)
IF(WITH_GUI)
  MESSAGE(STATUS "Compilation de l'interface graphique activée")
ENDIF(WITH_GUI)
```

Lors de l'appel à **cmake**, l'option **-D** permet de passer la valeur de l'option:

```
cmake -DWITH_GUI=ON
```

2.2.5 Fichier CMakeLists.txt final

```
cmake_minimum_required(VERSION 2.6)
PROJECT(jfl)
# Dépendances
FIND_PACKAGE(ZLIB REQUIRED)
INCLUDE_DIRECTORIES(${ZLIB_INCLUDE_DIR})
# Variable des sources et cible
SET(jfl_src test.c)
ADD_EXECUTABLE(jfl ${jfl_src})
# Information sur ZLIB
IF(ZLIB_FOUND)
  MESSAGE(STATUS "ZLIB présent")
  INCLUDE_DIRECTORIES(${ZLIB_INCLUDE_DIR})
ELSE(ZLIB_FOUND)
  MESSAGE(STATUS "ZLIB absent")
ENDIF(ZLIB_FOUND)
# Librairies
ADD_SUBDIRECTORY(malib SHARED)
TARGET_LINK_LIBRARIES(jfl malib ${ZLIB_LIBRARY})
# Option GUI
OPTION(WITH_GUI "Compiler l'interface graphique" OFF)
IF(WITH_GUI)
```

```
  MESSAGE(STATUS "Compilation de l'interface graphique activée")
ENDIF(WITH_GUI)
```

2.3 Installation

La génération des scripts d'installation du Makefile se fait au travers de la directive **INSTALL**. Elle permet de placer automatiquement les binaires, librairies, headers dans l'arborescence du système. Classiquement, un exécutable est placé dans /usr/bin. Lorsque l'on compile et installe un paquet "à la main", celui-ci est classiquement placé dans /usr/local.

```
INSTALL(TARGETS jfl DESTINATION "bin")
INSTALL(FILES mon_header.h DESTINATION "include")
INSTALL(TARGETS malib DESTINATION "lib")
```

Le chemin de destination est relatif par rapport à la variable **CMAKE_INSTALL_PREFIX**. Celle-ci contient une valeur par défaut mais peut-être modifiée à loisir.

Pour installer le binaire, il faut disposer des droits root et exécuter:

```
make install
```

Des permissions spécifiques peuvent êtres spécifiées, par exemple:

```
INSTALL(TARGETS jfl DESTINATION "bin" PERMISSIONS OWNER_READ OWNER_WRITE OWNER_EXECUTE)
```

2.4 Ant

Apache ant est un autre outil de *build* qui n'est pas "orienté shell" c'est-à-dire qu'il ne repose pas sur bash ou sh. C'est un projet Apache, fait pour compiler du Java. Le fichier de configuration par défaut, build.xml, est assez explicite. Il peut ressembler à:

```
<project>

  <target name="clean">
      <delete dir="build"/>
  </target>

  <target name="compile">
      <mkdir dir="build/classes"/>
      <javac srcdir="src" destdir="build/classes"/>
  </target>

  <target name="jar">
      <mkdir dir="build/jar"/>
      <jar destfile="build/jar/HelloWorld.jar" basedir="build/classes">
          <manifest>
              <attribute name="Main-Class" value="oata.HelloWorld"/>
          </manifest>
      </jar>
  </target>

  <target name="run">
      <java jar="build/jar/HelloWorld.jar" fork="true"/>
  </target>

</project>
```

Qui s'utilise ensuite pour par exemple construire le .jar de l'aplication:

```
ant compile
ant jar
ant run
```

2.4.1 Tâches

La balise *target* permet de déclarer une cible de construction, dont l'attribut *name* donne le nom. Une *target* se réalise ensutie par une suite de tâches qui utilisent des balises prédéfinies:

- delete, mkdir, copy, ...: manipulation de fichier
- java, jar, javac: exécution, construction d'un jar, compilation java
- exec: exécution d'une ligne de commande

La liste des tâches possibles est impresionnante:

Ant AntCall ANTLR AntStructure AntVersion Apply/ExecOn Apt Attrib Augment Available Basename Bindtargets BuildNumber BUnzip2 **BZip2** Cab Continuus/Synergy Tasks CvsChangeLog Checksum Chgrp **Chmod** **Chown** Clearcase Tasks Componentdef Concat Condition Supported conditions **Copy** **Copydir** Copyfile **Cvs** CVSPass CvsTagDiff CvsVersion Defaultexcludes **Delete** Deltree Depend Dependset Diagnostics Dirname Ear **Echo** Echoproperties EchoXML EJB Tasks Exec Fail Filter FixCRLF FTP GenKey Get **GUnzip** **GZip** Hostinfo Image Import Include Input Jar Jarlib-available Jarlib-display Jarlib-manifest Jarlib-resolve Java Javac JavaCC Javadoc/Javadoc2 Javah JDepend JJDoc JJTree Jlink JspC JUnit ..... Replace ReplaceRegExp ResourceCount Retry RExec Rmic Rpm SchemaValidate Scp Script Scriptdef Sequential ServerDeploy Setproxy SignJar Sleep SourceOffSite Sound Splash Sql Sshexec Sshsession Subant Symlink Sync **Tar** Taskdef Telnet Tempfile **Touch** Translate Truncate TStamp Typedef **Unjar** **Untar** Unwar **Unzip** Uptodate VerifyJar Microsoft Visual SourceSafe Tasks Waitfor War WhichResource Weblogic JSP Compiler XmlProperty XmlValidate XSLT/Style **Zip**

2.4.2 Tâches et attributs ou sous éléments

Chaque tâche possède ou peut posséder des attributs et des sous-éléments. La documentation renseigne des possibilités. Prenons par exemple la tâche **javac**.

- Attributs:
  - srcdir: le répertoire contenant les sources
  - debug: avec ou sans information de débug
- Sous-éléments: la tâche **javac** est comme un **FileSet**

FileSet: un groupe de fichiers. Par exemple:

```
<fileset dir="${server.src}" casesensitive="yes">
  <include name="**/*.java"/>
  <exclude name="**/*Test*"/>
</fileset>
```

Donc on peut faire pour javac:

```
<javac sourcepath="" srcdir="${src}"
       destdir="${build}" >
  <include name="**/*.java"/>
  <exclude name="**/Example.java"/>
</javac>
```

2.4.3 Et pour le C ?

Il existe une contribution pour déclarer des tâches **cpptasks:cc** dans un fichier de build. Il faut pour cela ajouter *cpptasks-1.0b5.jar* dans la variable d'environnement CLASSPATH et déclarer un nouveau *namespace*:

```
<project name="hello" default="compile" xmlns:cpptasks="antlib:net.sf.antcontrib.cpptasks">
   <target name="compile">
      <mkdir dir="target/main/obj"/>
      <cpptasks:cc outtype="executable" subsystem="console" outfile="target/hello" objdir="target/main/obj">
         <fileset dir="src/" includes="*.c"/>
      </cpptasks:cc>
</target> </project>
```

On obtient alors:

```
ant compile
Buildfile: /home/jf/Dropbox/Cours/outils_developpement/cours_outils/codes/ant/build.xml
compile:
[cpptasks:cc] 1 total files to be compiled.
[cpptasks:cc] Starting link
BUILD SUCCESSFUL

./target/hello
HelloWorld !
```

3 Subversion

3.1 Gestion de version et développement coopératif

Le but de la gestion de version est d'être capable d'administrer l'évolution d'un développement logiciel. Les modifications successives des fichiers, l'historique des raisons de ces modifications et un système de numérotation sont la bases de la gestion de version. Cet historique permet notamment de retourner à des version antérieurs du logiciel ou de ses parties.

La seconde caractéristique principale d'un logiciel de gestion de version est d'apporter une solution au développement coopératif. Il s'agit de gérer les conflits de développements, d'administrer le partage des mises à jour des parties du code et de permettre le développement à distance.

Différents outils existent et résolvent de manière différentes certaines caractéristiques des objectifs précédemment évoqués. Cependant, les principes de base sont souvent similaires. Citons notamment les plus connus:

- Concurrent versions system (CVS)
- Subversion (SVN)
- GIT
- GNU Arch
- Bitkeeper

3.2 Subversion

Dans subversion, un certain nombre de concepts sont issus de l'expérience de CVS, notamment afin de faciliter la transition de l'un à l'autre. Nous définissons ci-dessous, les principaux concepts de subversion:

Dépot (*repository*): Il s'agit de l'emplacement, souvent distant, ou les codes sources, leurs différentes versions et informations sont stockéees. On accède au dépot en HTTP, HTTPS ou SSH.

Projet: La notion de projet correspond à un répertoire du *repository*. Il peut s'agir par exemple des sources de tout un logiciel. Dans chaque projet on trouve en général les sous-répertoires suivants:

- trunk: la branche de développement principale
- branches: les différentes branches du projet
- tags: les versions du logiciels taguées

Copie de travail: Il s'agit de la copie locale de l'utilisateur qui évolue localement par rapport au dépot.

Révision: Il s'agit du système de numérotation de svn. A chaque modification envoyée au serveur, la numérotation est incrémentée.

3.2.1 Opérations du client subversion

Checkout

L'opération checkout récupère les sources depuis le serveur subversion. Elle écrase la copie de travail locale à l'utilisateur.

```
svn checkout http://serveur/projet .
svn co http://rst2pdf.googlecode.com/svn/trunk/ rst2pdf-read-only
Révision 2154 extraite
```

Import

L'importation consiste à envoyer, la première fois, les fichiers locaux dans un nouveau projet du serveur subversion:

```
svn import -m "New import" monprojet http://serveur/monprojet
Adding monprojet/truc.txt
```

Update

L'opération *update* met à jour le *repository* local à partir du serveur svn. A la différence de *checkout*, l'opération *update* n'écrase pas les fichiers modifiés localement.

```
svn update
U truc.txt
```

Add

L'opération *add* permet d'annoncer l'ajout d'un fichier au *repository*, sans pour autant le faire réellement. L'ajout ne sera effectif que lorsque l'opération *commit* ne sera effectué. Une conséquence de cette opération est que deux fichiers locaux peuvent coexister dans la copie de travail, l'un étant synchronisé avec le *repository* et l'autre jamais envoyé (si l'on n'a pas fait de *add* dessus).

```
svn add truc.txt
A truc.txt
```

Commit

C'est l'opération la plus importante: elle envoie les modification de la copie de travail au serveur svn. Lors de l'envoi, l'utilisateur saisi une description des modifications apportées à la copie locale.

```
svn commit
Sending truc.txt
Transmitting file data .........
Committed revision 19.
```

Pour obtenir la liste exhaustive des opérations: **svn help**.

3.2.2 Retour arrière

Après avoir travaillé sur la copie locale, on peut souhaiter retrouver l'état d'un fichier particulier avant sa modification. Une solution est de procéder à un nouveau *checkout* de ce fichier; cependant une commande existe pour remettre le fichier à l'état du dernier *update*, sans aucune connexion réseau nécessaire au serveur svn:

```
svn revert truc.txt
```

Si un *commit* a été effectué, la version récupéré par un *checkout* ou un *update* sera forcément celle envoyée par le *commit*. On ne peut donc revenir en arrière sans spécifier la révision souhaitée pour le fichier:

```
svn update -r 20 truc.txt
U    truc.txt
Actualisé à la révision 20.
```

3.2.3 Gestion des fichiers

Une amélioration notable de subversion par rapport à CVS est la meilleure gestion des opérations de déplacement et suppression des fichiers.

Suppression

Etant donné qu'il faut pouvoir revenir à des versions antérieurs d'un fichier, il était délicat de gérer la suppression d'un fichier puisqu'il n'est pas réellement supprimé du serveur. Dans svn, la suppression s'opère à l'aide de la ocmmande delete:

```
svn delete truc.txt
D truc.txt
svn commit -m "Suppression d'un fichier inutile"
Deleting truc.txt
Committed revision 88.
```

Déplacement

Dans CVS, le déplacement d'un fichier n'était pas possible. Il fallait copier le fichier sur le système de fichier, puis supprimer du repository le premier fichier et ajouter un nouveau fichier. La conséquence de cette manipulation est la perte des informations associées au premier fichier puisque le second fichier apparait comme nouveau en révision 1. Dans svn, cette abération est enfin disparue au profit de la commande *move* qui opère comme attendu.

```
cp truc.txt chose.txt
cvs remove truc.txt
cvs add chose.txt
cvs commit -m "Moved truc.txt to chose.txt"

svn move truc.txt chose.txt
A chose.txt
D truc.txt
svn commit -m "Moved truc.txt to chose.txt"
```

Copie

Une copie d'un fichier peut aussi être faite en conservant son historique antérieur:

```
svn copy truc.txt chose.txt
cvs commit -m "Copied file truc.txt to chose.txt"
```

Création de répertoire

Un répertoire peut être ajouté au repository:

```
svn mkdir rep
cvs commit -m "Creating a new directory"
```

3.2.4 Apparition de conflits

Lors d'un commit, un message d'erreur peut apparaitre signalant un conflit entre un fichier local et la version du repository. En effet, un autre utilisateur a probablement déjà envoyé ses modifications concernant ce fichier sur le serveur. Un *update* du *repository* local éclaire la situation:

```
svn update
U  INSTALL
G  README
C  bar.c
Updated to revision 46.
```

- U signifie que le fichier INSTALL est correctement mis à jour

- G signifie que le fichier README a été modifié localement et sur le serveur et qu'un *merGe* a été possible
- C signifie qu'un conflit ne pouvant être résolu par un *merge* subsiste

A ce point, un commit devient impossible:

```
svn commit --message "Add a few more things"
svn: Commit failed (details follow):
svn: Aborting commit: '/home/jf/bar.c' remains in conflict
```

3.2.5 Gestion des conflits

Lors de la détection d'un conflit, svn produit un certain nombre de fichiers:

- bar.c.r45: il s'agit de la version de la révision 45 du fichier tel qu'il était lors du dernier *update*.
- bar.c.r46: il s'agit de la version de la révision 46 du fichier tel qu'il est actuellement sur le *repository*.
- bar.mine: il s'agit de la copie locale de travail que l'on aurait souhaité commiter.
- bar.c: il s'agit du fichier contenant à la fois les modifications locales et les modifications de la révision 46.

A partir de tous ces fichiers, il faut éditer bar.c pour trouver une solution au conflit. Il peut s'agir d'écraser la révision 46 avec sa copie locale (on perd ce que l'autre développeur a commité), d'abandonner ses propres modifications (cp bar.c.r46 bar.c), ou d'éditer bar.c en enlevant les parties adéquats pour résoudre "à la main" le conflit.

On signale enfin la résolution du conflit afin de nettoyer l'espace de travail avant le *commit* par:

```
svn resolved bar.c
Resolved conflicted state of 'bar.c'
svn commit
Sending bar.c
Transmitting file data .........
Committed revision 47.
```

3.2.6 Historique

Subversion fournit des commandes pour explorer l'historique:

```
svn log
------------------------------------------------------------------------
r2 | harry | Mon, 15 Jul 2002 17:47:57 -0500 | 1 line
Added main() methods.
------------------------------------------------------------------------
r1 | sally | Mon, 15 Jul 2002 17:40:08 -0500 | 1 line
Initial import
------------------------------------------------------------------------
```

On peut toujours dénommer les révisions concernées:

```
svn log --revision 5:19   # shows logs 5 through 19 in chronological order
svn log -r 8              # shows log for revision 8
$ svn log -r 8 -v
------------------------------------------------------------------------
r8 | sally | 2002-07-14 08:15:29 -0500 | 1 line
Changed paths:
M /trunk/code/foo.c
A /trunk/code/doc/README
------------------------------------------------------------------------
```

3.2.7 Différences

Subversion fournit aussi un moyen simple de comparer des révisions:

```
svn diff
Index: rules.txt
===================================================================
--- rules.txt        (revision 3)
+++ rules.txt        (working copy)
@@ -1,4 +1,5 @@
 Be kind to others
 Freedom = Responsibility
 Everything in moderation
-Chew with your mouth open
+Chew with your mouth closed
+Listen when others are speaking
```

On peut toujours dénommer certaines révisions ou même comparer deux révisions du repository:

```
svn diff --revision 3 rules.txt
svn diff --revision 2:3 rules.txt
```

3.2.8 Synthèse des différences avec CVS

Une meilleure gestion des répertoires

La gestion des répertoires dans CVS est déplorable. Il n'est par exemple pas possible de supprimer un répertoire dans CVS: il subsiste, vide, dans la copie locale. Si l'on décide de le supprimer à la main du coté serveur, on perd les anciens fichiers et leur historique contenus dans ce répertoire.

Dans SVN, les répertoires sont considérés comme les fichiers et supportent les opérations de suppression, déplacement.

Une gestion offline de certaines opérations

Certaines opérations, comme le *tagging* ou le *branching* se font sans besoin de la connexion réseau au serveur. CVS n'a pas de commande *off-line*.

Une gestion atomique des commit

Si une opération de commit concernent plusieurs fichiers, le commit d'un fichier peut échouer à cause d'une version plus récente présente sur le *repository*. Dans CVS, les fichiers sans conflits sont commités sur le serveur et les fichiers en conflits non. Cela laisse le logiciel dans un état instable, par exemple potentiellement non compilable. Si le développeur ne peut réparer le conflit rapidemment, il a à moitié commité ses fichiers ce qui est le contraire de l'idée de *commit* atomique.

Dans SVN, le commit est atomique: tous les changements sont commités, ou aucun.

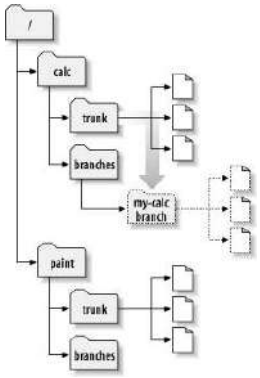
3.2.9 Branches

Cette partie du cours est basée sur "Version Control with Subversion (for Subversion 1.7)", par Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. Creative Commons Attribution License v2.0.

Une branche dans SVN est une copie du dépôt vers un autre répertoire (copie avec des *hard* links). Souvent, il s'agit du répertoire "branches" et non plus dans le répertoire "trunk". La copie peut-être locale ou bien faite sur le serveur distant. Par exemple:

```
$ svn copy http://svn.example.com/repos/calc/trunk \
         http://svn.example.com/repos/calc/branches/my-calc-branch \
         -m "Creating a private branch of /calc/trunk."

Committed revision 341.
```

Ce qui correspond à:

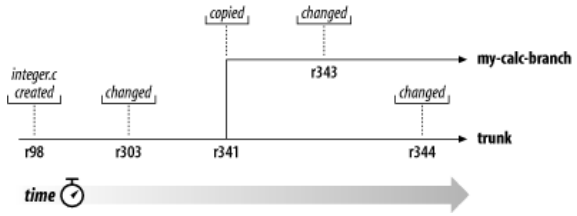


3.2.10 Travailler avec sa branche

A partir de là, on peut télécharger la branche de façon naturelle:

> $ svn checkout http://svn.example.com/repos/calc/branches/my-calc-branch A my-calc-branch/Makefile A my-calc-branch/integer.c A my-calc-branch/button.c Checked out revision 341.

D'un point de vue des commits, les numéros vont alterner, en fonction de l'endroit d'où l'on fait le commit:



La branche et le trunk vivent donc une histoire indépendante avec le temps. Comme précisé par Collins-Sussman et al. l'historique en atteste:

```
$ svn log -v integer.c
------------------------------------------------------------------------
r343 | user | 2002-11-07 15:27:56 -0600 (Thu, 07 Nov 2002) | 2 lines
M /calc/branches/my-calc-branch/integer.c
------------------------------------------------------------------------
r303 | sally | 2002-10-29 21:14:35 -0600 (Tue, 29 Oct 2002) | 2 lines
M /calc/trunk/integer.c
```

3.2.11 Garder une branche à jour

En cas de développement long dans une branche, il peut être utile de rapatrier les derniers commit du *trunk* dans votre branche. Ainsi, si plus tard vous décidez de pousser vos modifications de la branche vers le *trunk* moins de conflits surviendront.

Par exemple, pour la branche "my-calc-branche", on ferait:

```
$ pwd
/home/user/my-calc-branch
$ svn merge ^/calc/trunk
--- Merging r345 through r356 into '.':
U    button.c
U    integer.c
--- Recording mergeinfo for merge of r345 through r356 into '.':
```

Une fois cela fait, les modifications prenvenant du *trunk* sont vues comme des modifications locales à valider:

```
$ svn status
M       button.c
M       integer.c
```

```
$ svn commit -m "Merged latest trunk changes to my-calc-branch."
Sending        button.c
Sending        integer.c
Committed revision 357.
```

3.2.12 Merge d'une branche vers le trunk

La dernière étape pour une branche consiste à pousser les modifications de la branche vers le trunk. On procède alors de la façon suivante:

Etape 1: vérifier la synchro avec le trunk

```
$ svn merge ^/calc/trunk
--- Merging r381 through r385 into '.':
U    button.c
U    README
--- Recording mergeinfo for merge of r381 through r385 into '.':
U   .

$ # build, test, ...

$ svn commit -m "Final merge of trunk changes to my-calc-branch."
Sending        .
Sending        button.c
Sending        README
Transmitting file data ..
Committed revision 390.
```

Etape 2: le merge

```
$ pwd
/home/user/calc-trunk

$ svn update  # (make sure the working copy is up to date)
Updating '.':
At revision 390.

$ svn merge --reintegrate ^/calc/branches/my-calc-branch
--- Merging differences between repository URLs into '.':
U    button.c
U    integer.c
U    Makefile
--- Recording mergeinfo for merge between repository URLs into '.':

$ # build, test, verify, ...

$ svn commit -m "Merge my-calc-branch back into trunk!"
Sending        .
Sending        button.c
Sending        integer.c
Sending        Makefile
Transmitting file data ..
Committed revision 391.
```

4 Git

4.1 Création du repository

Par défaut, git créé un repository local:

```
git init
Initialized empty Git repository in .git/
```

L'importation du projet se fait comme dans SVN:

```
cd project
git add .
```

La commande add ajoute le répertoire courant et ses fichies dans l'index des fichiers à envoyer lors du prochain commit. Il est possible de vérifier le status courant des ajouts:

```
git status
# On branch master
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#     new file:   test.txt
```

4.1.1 Clone

Lorsqu'on créé un repository local à partir d'un serveur git, on utilise la commande **clone** pour créer une copie locale du repository distant. A la différence du **checkout** de SVN, git fonctionne avec un équivalent de serveur local, sur lequel on *checkout* la branche principale. Le fait de cloner le *repository* distant va créer un repository local contenant toutes les branches et va, par défaut, faire un *checkout* de la branche principale active.

Par exemple, pour cloner le repository du noyau linux, on peut faire:

```
git clone git://git.kernel.org/pub/scm/.../linux-2.6 my2.6
cd my2.6
make
```

La commande peut aussi permettre de faire un clonage local:

```
git clone -l git git3
Initialized empty Git repository in git3/.git/
```

4.2 Synchronisation avec le repository

Les **nouveaux** fichiers doivent êtres ajoutés, puis *commités*:

```
git add file1 file2 file3 test.txt
git commit
```

On peut vérifier à tout moment l'état des fichiers du *repository* local:

```
... des modifs à test.txt ...
... création du fichier machin ...
$ git status
Modifications qui ne seront pas validées :

modifié :         test.txt

Fichiers non suivis:

machin
```

4.2.1 Add et commit

Attention à la différence de comportement entre SVN et git pour les commandes **add** et **commit**. A la différence de SVN, Git ne commit pas un fichier mis à jour s'il n'a pas été ajouté à nouveau:

```
git commit
# On branch dev2
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes)
#
#     modified:   test.txt
#
no changes added to commit (use "git add" or "git commit -a")
```

Il est possible de faire l'**add** avec le **commit**:

```
git commit -a
[dev2 b6ca24a] Bug fix.
 Committer: JF <jf@eeepc.(none)>
```

4.2.2 Push

Push permet d'envoyer l'état de son repository local sur le serveur distant. Par défaut, on *push* toutes les branches *committées* vers le serveur:

```
git push
Counting objects: 10, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (6/6), 490 bytes, done.
Total 6 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
To ./git
   04b041b..745243e  master -> master
   b6ca24a..692003a  dev2 -> dev2
```

La magie de git est que l'on peut pousser un repository vers un autre serveur. Il suffit de précser son url:

```
git push https://github.com/jflalande/GitTest2.git
```

4.2.3 Remote

Cependant, le fait d'avoir pousser un commit vers un autre serveur distant ne modifie pas "l'attache" du repository au remote originel, ce qui se voit ici:

```
git push https://github.com/jflalande/GitTest2.git
origin        https://github.com/jflalande/GitTest.git (fetch)
origin        https://github.com/jflalande/GitTest.git (push)
```

Pour attacher un autre dépot distant, il faut faire:

```
git remote add  autre https://github.com/jflalande/GitTest2.git
git remote -v
autre https://github.com/jflalande/GitTest2.git (fetch)
autre https://github.com/jflalande/GitTest2.git (push)
origin        https://github.com/jflalande/GitTest.git (fetch)
origin        https://github.com/jflalande/GitTest.git (push)
```

par défaut cela pousse vers origin, mais on peut maintenant pousser vers autre:

```
git push autre
```

4.2.4 origin

origin est le nom du serveur distant par défaut lorsque vous clonez un repository. S'il n'y a pas encore de dépot distant, il faut l'ajouter comme vu précédemment.

```
$ git clone https://github.com/jflalande/GitTest2.git
$ git remote -v
origin        https://github.com/jflalande/GitTest2.git (fetch)
origin        https://github.com/jflalande/GitTest2.git (push)
```

On voit ici que deux associations sont déclarées:

- l'association en cas de *fetch* (ou *pull*) c'est à dire de quel serveur on reçoit les modifications distantes
- l'association en cas de *push* c'est à dire vers quel serveur on envoie les modifications locales

4.2.5 .config

Le *remote* **origin** et les éventuelles autres *remote* ne sont qu'une configuration dans le fichier .git/config:

```
$ cat .git/config
[core]
  repositoryformatversion = 0
  filemode = true
  bare = false
  logallrefupdates = true
[remote "origin"]
  url = https://github.com/jflalande/GitTest.git
  fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
  remote = origin
  merge = refs/heads/master
[remote "autre"]
  url = https://github.com/jflalande/GitTest2.git
  fetch = +refs/heads/*:refs/remotes/autre/*
```

La ligne fetch fait l'association entre les branches distantes et locales:

- heads/*: toutes les branches locales
- remotes/origin/*: toutes les branches de **origin**

4.2.6 Pull

L'opération de *pull* permet de fusionner la branche localement active avec la version distante, en générale la tête du repository distant (HEAD).

```
git pull
remote: Counting objects: 5, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From repository
  ef90a43..04a5588  master     -> origin/master
Updating ef90a43..04a5588
Fast-forward
 test.txt |    1 +
1 files changed, 1 insertions(+), 0 deletions(-)
```

Cela peut éventuellement échouer si les modifications locales ne sont pas comitées:

```
git pull origin
Updating 04a5588..34fa2de
error: Your local changes to 'test.txt' would be overwritten
by merge.  Aborting.  Please, commit your changes or stash them
before you can merge.
```

Dans ce cas, il faudra fusionner les modifications locales et la tête du repository distant.

4.2.7 Fetch

Si l'on suppose que quelqu'un pousse des modifications sur le serveur **origin**:

```
... des modifs locales ...
... des modifs poussées sur le serveur distant ...
git diff master origin/master
   ... RIEN ...
```

En effet, git travaille en locale sur votre repository (en mode non connecté): il faut rafraichir ce repository local c'est à dire l'image de ce qu'il y a sur le serveur **origin**. L'opération **fetch** réalise cela: elle permet de récupérer toutes les modifications depuis le serveur **origin** mais en ne les appliquant pas à votre copie de travail locale. Elles sont stockées dans votre dépot locale (le .git) mais n'**écrasent pas** vos fichiers courants.

```
git fetch
Depuis https://github.com/jflalande/GitTest
 2d39248..efb73d5  master     -> origin/master
```

On peut alors visionner la différence entre votre copie locale et le **HEAD** de **origin**:

```
git diff master origin/master
  -// une ligne écrite pendant que je travaille et poussée
  +// une ligne écrite en arrière plan
```

4.2.8 Fetch et commit

Cependant, on peut toujours visualiser la différence entre votre copie locale et l'index:

```
git diff
+// des choses en cours...
+// plein
```

Si vous commitez votre modification locale, vous ne pourrez pas la pousser.

```
git commit -a
git diff
.. RIEN ..
git diff master origin/master
  -// une ligne écrite pendant que je travaille et poussée
  +// une ligne écrite en arrière plan
  +// des choses en cours...
  +// plein

git push
 ! [rejected]        master -> master (non-fast-forward)
astuce: Les mises à jour ont été rejetées car la pointe de
la branche courante est derrière astuce: son homologue distant.
```

4.2.9 Fetch et merge

En effet, il y a une différence entre le pointeur de tête de votre branche locale et le pointeur de tête du serveur **origin**:

```
$ git show-ref
3a702dbd2206b27b6a551e4157b4c3a05a3198f8 refs/heads/master
5937bd5a0450628df4d6dfcd551d88adeb7d5a9f refs/remotes/origin/master
```

Et on peut voir que **3a702db** est la tête locale:

```
git log --graph --decorate --oneline
* 3a702db (HEAD -> master) ma modif
```

C'est ici qu'intervient le **merge**:

```
git merge -no-ff
git log --graph --decorate --oneline
*   f344f37 (HEAD -> master) Merge remote branch
                             'refs/remotes/origin/master'
| \
| * 5937bd5 (origin/master) modif
* | 3a702db ma modif
| /
*   254855d Merge remote branch 'refs/remotes/origin/master'
```

Il ne reste plus qu'à pousser ce qui va ramener les pointeurs de tête du repository local sur le même que sur le serveur:

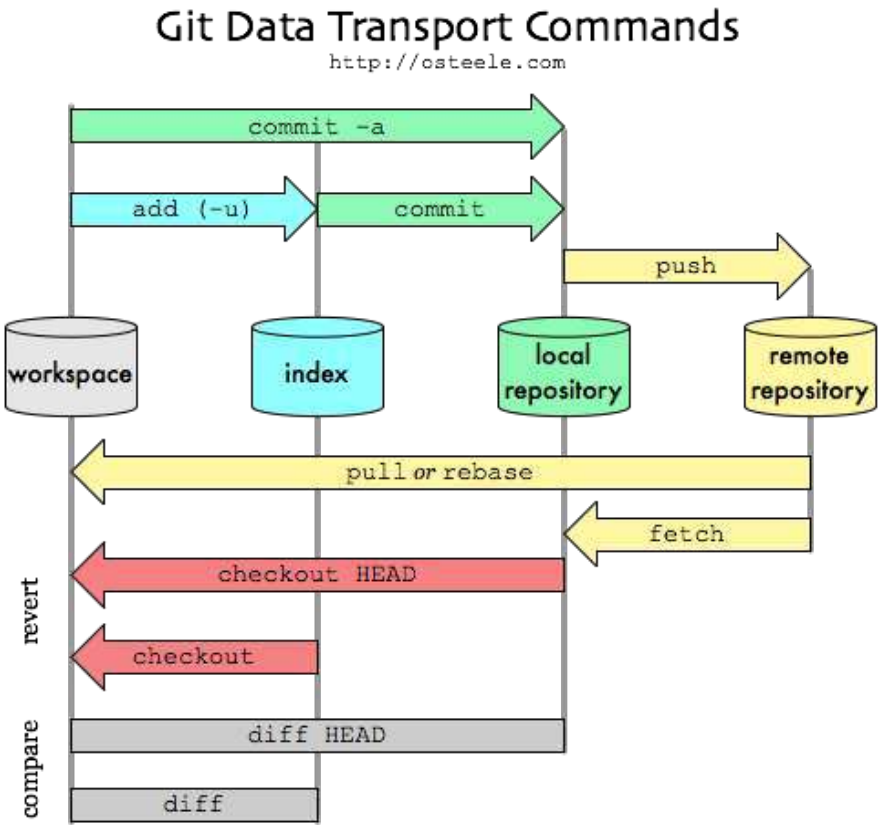
```
$ git push
5937bd5..f344f37  master -> master

$ git show-ref
f344f379b6039a36280bb3cd4c01ff9d7c3e6aa7 refs/heads/master
f344f379b6039a36280bb3cd4c01ff9d7c3e6aa7 refs/remotes/origin/master
```

L'option *no fast forward* demande à git de créer un commit même si vous n'avez rien modifié dans votre branche, comme expliqué par Kevin Ballard:

The --no-ff flag prevents git merge from executing a "fast-forward" if it detects that your current HEAD is an ancestor of the commit you're trying to merge. A fast-forward is when, instead of constructing a merge commit, git just moves your branch pointer to point at the incoming commit. This commonly occurs when doing a git pull without any local changes.

4.2.10 Synthèse

(Oliver Steele CC-BY-SA)



4.2.11 Gestion d'un conflit et merge automatique

Si un **push** échoue c'est qu'il existe des modifications plus récentes sur le serveur:

```
  ... des modifs locales ...
$ git commit
$ git push
# conflits !
! [rejected]        master -> master (fetch first)
error: impossible de pousser des références vers
  'https://github.com/jflalande/GitTest.git'
astuce: Les mises à jour ont été rejetées car la branche distante
```

```
contient du travail que astuce: vous n'avez pas en local. Ceci est
généralement causé par un autre dépôt poussé astuce: vers la même
référence. Vous pourriez intégrer d'abord les changements distants
astuce: (par exemple 'git pull ...') avant de pousser à nouveau.

git pull origin
Auto-merging test.txt
Merge made by recursive.
$ git push
```

4.2.12 Gestion d'un conflit et merge manuel

Parfois, le merge n'est pas automatique:

```
$ git pull
 9e24de6..8142cf9  master     -> origin/master
 Fusion automatique de test.c
 CONFLIT (contenu) : Conflit de fusion dans test.c
```

Il faut le résoudre à la main:

```
int main()
{
      printf("Hello Git !");
      <<<<<<< HEAD
              printf("Une modif :)");
      printf("Ma modif continue...");
      =======
              printf("Une modif de quelqu'un d'autre :)");
      >>>>>>> 8142cf920e9bad3f22ce733d0334dc112abac5a1
```

```
git add test.c
git commit
git push
```

4.3 Branches

Création d'une branche:

```
git branch dev2
git branch
  dev2
* master
```

Changement de branche:

```
git checkout dev2
Switched to branch 'dev2'
branch
* dev2
  master
```

Changer de branche suppose d'avoir commité les modifications de la branche courante (car les modifications locales seraient écrasées sinon):

```
(édition d'un fichier)
git checkout master
error: You have local changes to 'test.txt'; cannot switch branches.
```

4.3.1 Fusion de branches

Evidemment, travailler dans une branche sert à fusionner un développement avec la branche **master** en général:

```
$ git checkout -b correctif
Switched to a new branch 'correctif'
$ vim index.html
$ git commit -a -m "correction de l'adresse email incorrecte"
[correctif 1fb7853] "correction de l'adresse email incorrecte"
 1 file changed, 2 insertions(+)
$ git checkout master
$ git merge correctif
Updating f42c576..3a0874c
Fast-forward
index.html | 2 ++
1 file changed, 2 insertions(+)
```

Vous noterez la mention fast-forward lors de cette fusion (merge). Comme le commit pointé par la branche que vous avez fusionnée descendait directement du commit sur lequel vous vous trouvez, Git a simplement déplacé le pointeur (vers l'avant). Autrement dit, lorsque l'on cherche à fusionner un commit qui peut être atteint en parcourant l'historique depuis le commit d'origine, Git se contente d'avancer le pointeur car il n'y a pas de travaux divergents à fusionner — ceci s'appelle un fast-forward (avance rapide).

4.4 Le remisage (stash)

(cf "Pro Git book", Scott Chacon and Ben Straub CC BY NC SA.)

Souvent, lorsque vous avez travaillé sur une partie de votre projet, les choses sont dans un état instable mais vous voulez changer de branche pour travailler momentanément sur autre chose. Le problème est que vous ne voulez pas valider un travail à moitié fait seulement pour pouvoir y revenir plus tard. La réponse à cette problématique est la commande git stash.

Remiser prend l'état en cours de votre répertoire de travail, c'est-à-dire les fichiers modifiés et l'index, et l'enregistre dans la pile des modifications non finies que vous pouvez réappliquer à n'importe quel moment.

```
# Editer quelques fichiers
$ git status
  # Changes not staged for commit:
  modified:   lib/simplegit.rb
$ git stash
Saved working directory and index state \
"WIP on master: 049d078 added the index file"
HEAD is now at 049d078 added the index file
$ git status
# On branch master
nothing to commit, working directory clean
```

On peut alors visualiser ce qu'il y a dans la pile de remisage. Dans cet exemple il y a trois remisages:

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051... Revert "added file_size"
stash@{2}: WIP on master: 21d80a5... added number to log
```

Vous pouvez alors faire autre chose sur votre repository, mais plus tard, vous pouvez dépiler la première modification remisée avec la commande "stash apply":

```
$ git stash apply
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   lib/simplegit.rb
```

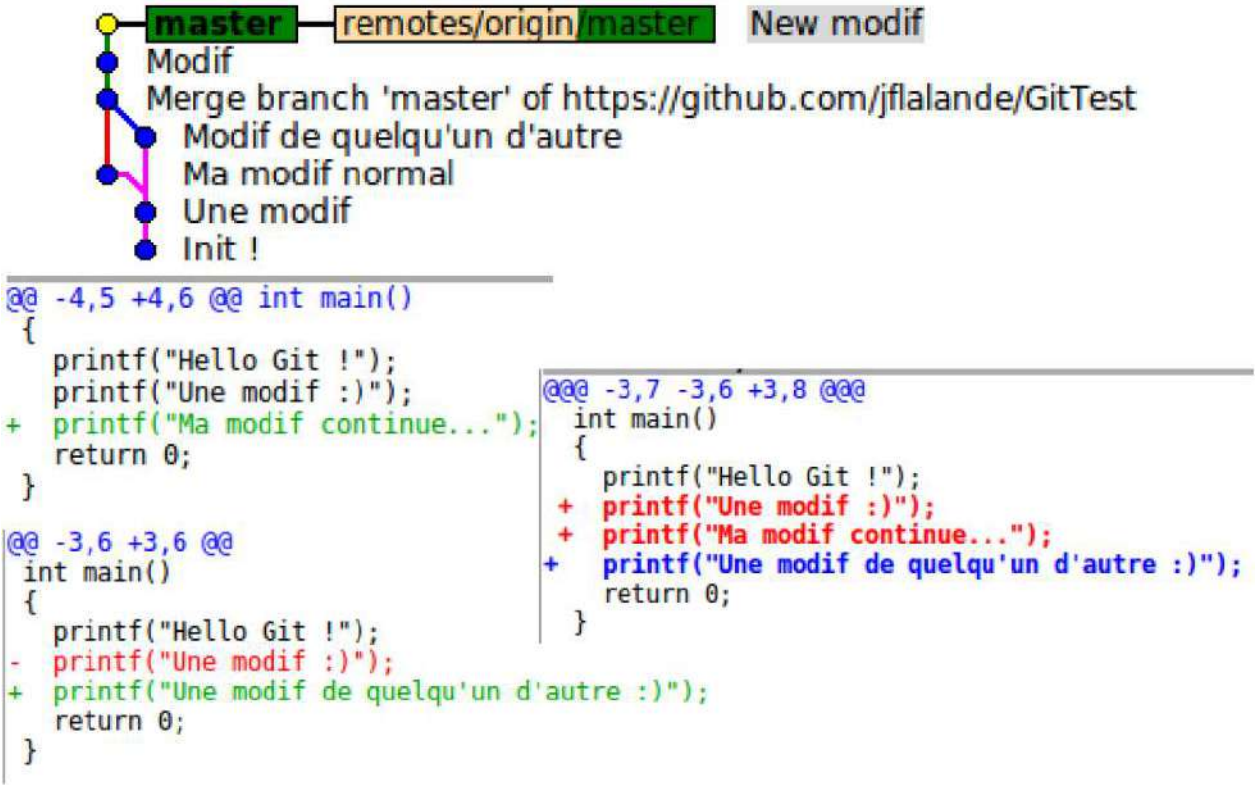
Mais à quoi cela peut-il bien servir ?

4.4.1 Remisage et pull

Le remisage permet de résoudre les conflits quand on ne souhaite pas commiter tout de suite, ou bien parce qu'on sait que quelqu'un à déjà poussé des modifications sur le serveur. Ainsi, on va privilégier la récupération des modifications du serveur en mettant de côté nos propres modifs. Puis, on va essayer de réappliquer ces modifications.

```
  ... des modifs locales ...
$ git pull
# Conflit car des modifs locales pas commitées
$ git stash
Saved working directory and index state WIP on master:
8142cf9 Modif de quelqu'un d'autre
HEAD est maintenant à 8142cf9 Modif de quelqu'un d'autre
$ git pull
Fast-forward
  test.c | 3 +++
  1 file changed, 3 insertions(+)
$ git stash apply
  Fusion automatique de test.c
  Sur la branche master
$ git add test.c
$ git commit
$ git push
```

4.5 gitk



5 Bibliographie

Cmake	CMake: la relève dans la construction de projets, Linux Magazine 92, 3 février 2009.
Doc28	CMake 2.8 Documentation
SVNJB	Introduction à Subversion, Julien Barnier, 2 août 2005.
SVNSR	A Novices Tutorial on Subversion, Sean Russell
SVNbook	Version Control with Subversion, Ben Collins-Sussman, Brian W. Fitzpatrick, C. Michael Pilato.
GH	Benchmarks Amadeus, Gurvan Huiban.
GitBook	Pro Git book.
GitHub	GitHub workflow: "ThinkUp Contributor Workflow".