Git your Life for Fun and Profit!

Cyril Soldani

Geeks Anonymes, Interface, ULg

19th June 2013

Outline

Outline

Finding the Last Version

Real-life example:

- Hey, can you send me the source of that article?
- Sure, . . . hum, well, . . .

`article.tar.bz2`	There it is!
`article_final.tar.bz2`	No this one is more recent.
`article_final2.tar.bz2`	Wait, this one even more so.
`article_really_final.tar.bz2`	That should be it.
`article_last.tar.bz2`	Wait, was this the last one?
`article-20081023.tar.bz2`	**:'(**

Poor man's versioning

Always use **dates** in archive file names. You can add a short comment too.
e.g. `article-20081023-camera_ready.tar.bz2`

Collaborating on a Document

Only one person manages the document

```
To:  pour-doc-owner@ulg.ac.be
On page 1, third paragraph, it misses an s
to kill.  At the bottom of page 1, replace
'fucking stupid' by 'not optimal'.  ...
```

What an exiting hour in perspective!

Each one changes its copy

Here is the corrected document, can you (manually) merge it with the versions you received from the other 10 participants?

One shared copy

- Hey, just saved the doc with my changes on the DropBox ;-)
- How nice, you just overwrote my last 4 hours of work! :'(

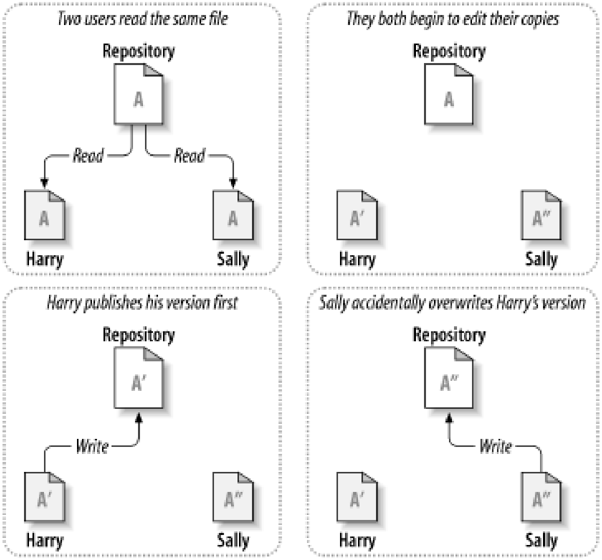
Collaborating with Oneself

A simple way to shoot oneself in the foot:

1. Take a snapshot archive of current stable version (with **date**).
2. Begin implementing your crazy experimental idea.
3. Fix some bugs in old code, revealed during **testing**.
4. Your idea was crap, discard experimental version.
5. Start back from stable version archive.
6. You lost your bug fixes, which also applied to stable version.

One need a way to convey changes (such as bug fixes) across multiple, different versions.

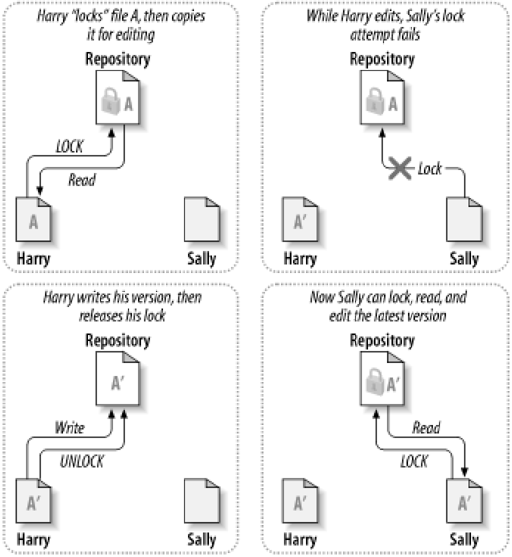
The Collaboration Problem



Two users read the same file
Repository
A
Read
Read
A
A
Harry
Sally
They both begin to edit their copies
Repository
A
A′
A″
Harry
Sally
Harry publishes his version first
Repository
A′
Write
A′
A″
Harry
Sally
Sally accidentally overwrites Harry's version
Repository
A″
Write
A′
A″
Harry
Sally

From the SVN Book

Locking



Harry "locks" file A, then copies it for editing
Repository
A
LOCK
Read
A
Harry
Sally
While Harry edits, Sally's lock attempt fails
Repository
A
Lock
A′
Harry
Sally
Harry writes his version, then releases his lock
Repository
A′
Write
UNLOCK
A′
Harry
Sally
Now Sally can lock, read, and edit the latest version
Repository
A′
Read
LOCK
A′
A′
Harry
Sally

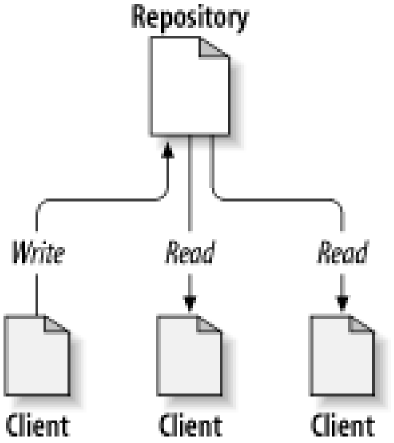
From the SVN Book

Why Use Version Control?

- Backup and restore
- Synchronization
  - between members of a team;
  - between multiple versions.
- Selective undo
- Sandboxing
- Track changes
- Documentation

Outline

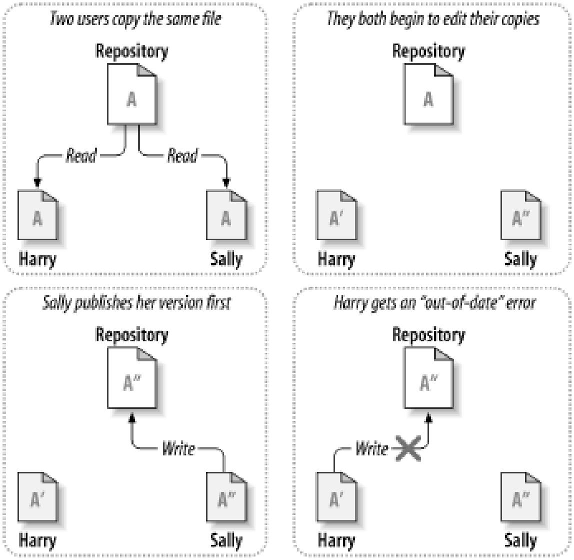
Centralized Version Control Model



From the SVN Book

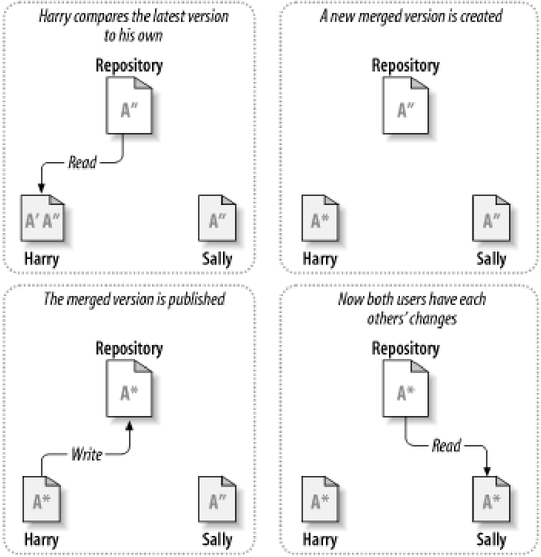
- One central **repository**, on a **server**. Stores the files and their history.
- Many **clients**, *i.e.* users connecting to the repo
- Each client has one or more **working copies**, *i.e.* a local copy of the files, where changes are made
- A **revision** identifies a point in time of the repo, it is denoted by a number.
- **HEAD** denotes last revision.

Solution to the Collaboration Problem


Two users copy the same file
Repository
A
Read
Read
A
Harry
A
Sally
They both begin to edit their copies
Repository
A
A′
Harry
A″
Sally
Sally publishes her version first
Repository
A″
Write
A′
Harry
A″
Sally
Harry gets an "out-of-date" error
Repository
A″
Write
A′
Harry
A″
Sally

From the SVN Book

Solution to the Collaboration Problem (cnt'd)



Harry compares the latest version to his own
Repository
A″
Read
A′ A″
Harry
A″
Sally
A new merged version is created
Repository
A″
A*
Harry
A″
Sally
The merged version is published
Repository
A*
Write
A*
Harry
A″
Sally
Now both users have each others' changes
Repository
A*
Read
A*
Harry
A*
Sally

From the SVN Book

Subversion

- Subversion is a popular centralized versioning system
- It is free open-source software
- It is cross-platform (Linux, FreeBSD, Mac OS X, Windows)
- Its functionality is exposed through two commands:

svn	allows to manipulate working copies
svnadmin	manages repositories

- There are also GUIs (*e.g.* Tortoise SVN) and support for IDEs

Obtaining help

```
svn help [COMMAND]
```

List available commands, or describe a command.

Updating your working copy

Obtain a working copy for the first time

This action is called **checkout** :

```
svn checkout REPOSITORY_URL DEST_DIR
```

This creates the local copy folder with additional `.svn` folders holding (part of) the history, and subversion settings.

Updating an existing working copy

```
svn update
```

Incorporates changes in the repository into the working copy.

- Always update your working copy prior to making changes, as subversion will refuse new changes to out-of-date files.

Making changes

- To edit a file, simply open it in your favorite editor (VIM, of course), and make your changes
- To delete, copy or rename a file, use `svn`

Tree changes

`svn add FOO`	adds FOO to the repo
`svn delete FOO`	removes FOO from the repo
`svn copy FOO BAR`	copies FOO to BAR
`svn move FOO BAR`	moves FOO to BAR
`svn mkdir FOO`	creates and adds folder FOO

- Changes on the working copy do not impact the repository directly

Review your changes

- Before sending your changes to the repository, check exactly what you have changed:
    - Allows to detect errors
    - Helps making good **log** messages

Inspecting the working copy

`svn status` list the files that have changed since last update

`svn diff FOO` shows what has changed in file FOO since last update

- Fix eventual problems, either by modifying a file again, or by **reverting** to the version from last update

Reverting changes

```
svn revert FOO
```

Restores FOO to whatever it was after last update.

Resolve any conflict

- Update your working copy. If some files have changed both in the repository and in your working copy, there is a **conflict**
- It is your responsibility to fix conflicts, by inspecting the diverging changes and:
  - choose your own version, or
  - choose repository version, or
  - choose previous version, or
  - mix both versions
- You can fix conflicts interactively during update, or postpone them for later treatment.

Resolving a postponed conflict

```
svn resolve --accept STATE FOO
```

Choose how to resolve the conflict, see `svn help resolve`

Publish your changes to the repository

Publish your changes to others

This action is called **commit**:

```
    svn commit
```

When committing, you give a message indicating what the change is about, which is consigned in the **log** file.

- Subversion will refuse to commit if you have pending conflicts
- If there are new conflicts (the repository could have been modified since last update), you will need to resolve them
- Once the commit is complete, your changes are recorded in the repository, the **revision** number is incremented and your changes become visible to others

Summary

1. Checkout or update your working copy (checkout, update)
2. Make changes (edit, add, copy, delete, . . .)
3. Review your changes (status, diff)
4. Fix your mistakes (edit, revert)
5. Resolve conflicts (update, resolve)
6. Publish your changes (commit)

Examining history

Showing the log

```
svn log [FOO]
```

Shows log (relevant to FOO).

Showing differences

```
svn diff -rA:B FOO
```

Shows what changed in FOO between revision A and revision B.

Obtain an old version of a file

```
svn cat -rA FOO > FOO.old
```

Recover the content of FOO at revision A into FOO.old.

Time traveling

```
svn update -rA
```

Make the working copy what the repository was at revision A.

Starting a New Project

Creating the repository

On the server:

```
    svnadmin create REPO
```

The repository is the subversion database. It cannot be modified directly (only through working copies).

Importing a project into a repository

```
    svn import DIR REPO_URL
```

Recommended repository layout

trunk	main line of development
branches	contains alternative versions
tags	contains precise releases

This is only a convention, these folders are not treated specially by subversion.

Multiple parallel branches

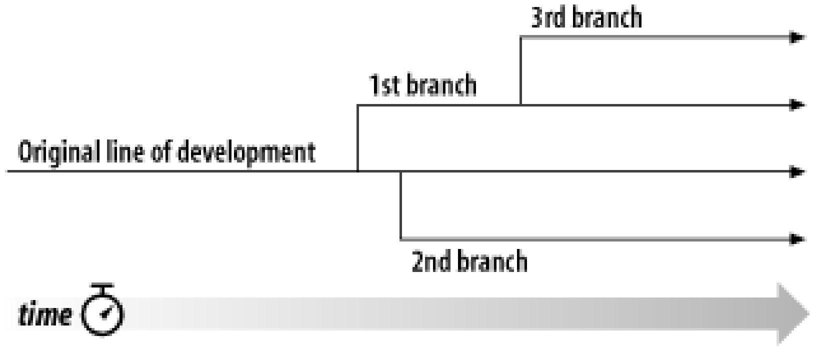


Figure from the SVN Book

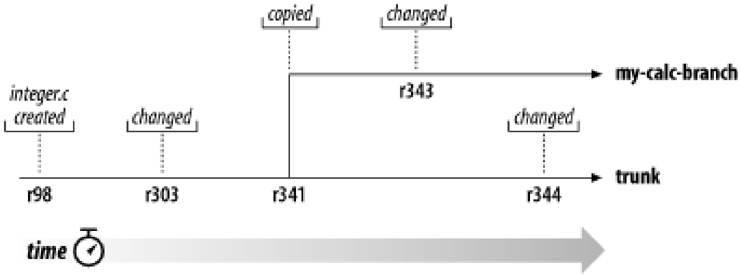
Branching

- In subversion, a branch is simply a copy:

Branching

```
svn copy REPO/calc/trunk \
    REPO/calc/branches/my-calc-branch
```

- You work on a branch as you would on any other folder
- File histories branch too



From the SVN Book

Merging

- Merging is synchronizing two branches
- When developing a branch, you'll want to synch with trunk from time to time

Merging from an ancestor branch

```
svn merge ^/calc/trunk
```
(when in branch)

- When merging, you can encounter conflicts, to be resolved as before
- If you want to integrate a branch back to trunk, you can merge it back

Merging from a child branch

```
svn merge --reintegrate \
     ^/calc/branch/my-calc-branch
```
When in trunk

What else?

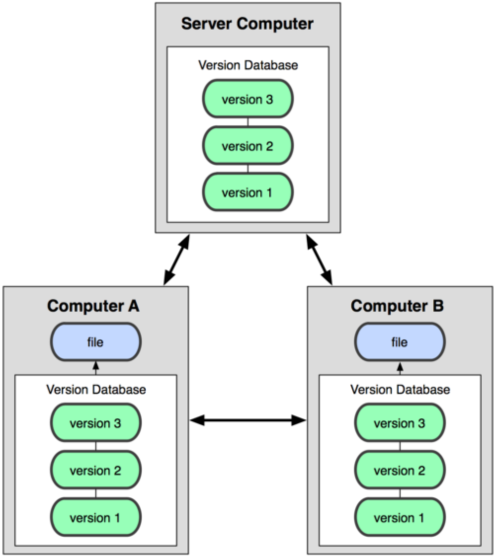
- Keyword substitutions (*e.g.* $Id: . . . $)
- Properties (*e.g.* files to ignore)
- Tags
- Combining files from multiple repositories
- Advanced branch and merge
- Repository administration
- Server configuration
- . . .

References

Subversion	`http://subversion.tigris.org/`
The SVN Book	`http://svnbook.red-bean.com/` This presentation is mostly derived from this freely available book
Tortoise SVN	`http://tortoisesvn.net/`

Outline

Decentralized Version Control Model



From the Git Book

Decentralized Version Control Pros and Cons

Pros

- Every working copy is a full backup of the data
- You can work off-line
- You can work with several repositories within the same project
- You can rewrite (local) history:
  - You can do microcommits
  - (You can hide your mistakes and look smarter than you are)
- Allows private work, eases experimental jump in
- Often faster

Cons

- More complex
- Less control on project evolution
- Less sharing?

Git

- Git is an increasingly popular distributed versioning system
- It is free open-source software
- It is cross-platform (although better support for Linux)
- It is very efficient :-)
- It is very powerful ;-)
- It can be very complex :'(
- It has GUIs and IDEs plugins (although less than subversion)

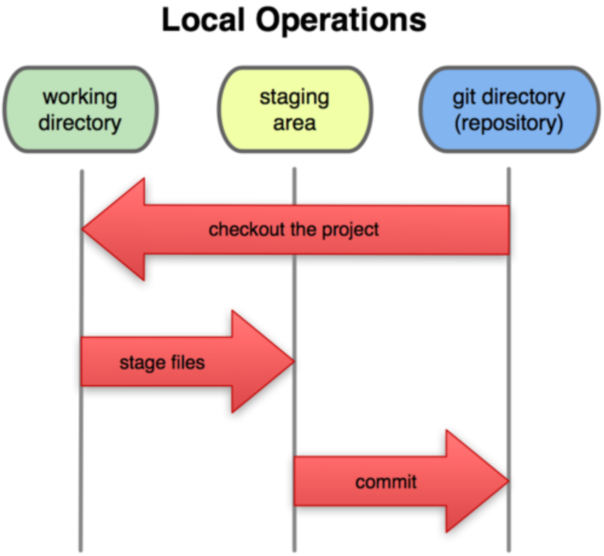
- There is no global **revision** numbers, hashes instead
- Hashes may be abbreviated, and revisions may be given relatively to HEAD (*e.g.* HEAD^, HEAD~4)

Obtaining help

```
git help [COMMAND]
```

List available commands, or describe a command.

The Three States



From the Git Book

Updating your working copy

Obtain a working copy for the first time

This action is called **clone** :

```
git clone REPOSITORY_URL DEST_DIR
```

This creates a working copy of the remote repository with a complete copy of the repository itself in `.git`.

Updating an existing working copy

```
git pull [REPO]
```

Incorporates changes from a remote repo into the current branch.

- As before, always update your working copy prior to making changes.

Making changes

- To edit a file, simply open it in your editor, and make your changes. Then **stage** your changes with `git add`.
- To delete, copy or rename a file, use `git`

Tree changes

`git add FOO`	adds FOO to the index
`git rm FOO`	removes FOO
`git mv FOO BAR`	moves FOO to BAR

- These changes only impacted the working directory and index so far, the repository is still untouched.

Review your changes

- Before committing changes to the repository, check exactly what you have changed:
  - Allows to detect errors
  - Helps making good **log** messages

Inspecting the working copy

`git status` list changes in the working directory and index

`git diff FOO` shows what has changed in file FOO compared to index version

`git diff --cached FOO` shows what has changed in file FOO compared to repository version

- Fix eventual problems, either by modifying a file again, or by **reseting** to the version from the repository (or index)

Forgetting changes

```
git reset FOO
```

Restores FOO to a previous state.

Publish your changes to the repository

Publish your changes to your own repo

This action is called **commit**:

```
git commit
```

When committing, you give a message indicating what the change is about, which is consigned in the **log** file.

Publish your changes to others

```
git pull [REPO]
```

Integrates changes from remote repo (this may lead to conflicts).

```
git push [REPO]
```

Upload your changes to remote repository.

Resolve any conflict

- When you do a `git pull` (or push), you may get conflicts if the same files were modified on both your repository and the remote one.
- Conflicting files will contain both versions, with markers
- You have to edit the files to merge the changes and remove the markers
- You can then add and commit the result of the merge
- If you were trying to pull or push, you can now try again

Summary

1. Checkout or update your working copy and repo (clone, pull)
2. Make changes (edit, add, cp, rm, . . .)
3. Review your changes (status, diff)
4. Fix your mistakes (edit, reset)
5. Stage changes (add)
6. Commit changes (commit)
7. Resolve conflicts (pull, edit, add, commit)
8. Publish your changes (push)

Examining history

Showing the log

```
    git log [FOO]
```

Shows log (relevant to FOO).

Showing differences

```
    git diff [REV1] [REV2] [FOO]
```

Shows what changed in FOO between revisions REV1 and REV2.

Obtain an old version of a file

List file hashes in a commit with

```
    git git ls-tree COMMIT_HASH
```

Then recover file with

```
    git git cat-file blob FOO_HASH > FOO.old
```

Time traveling

```
    git checkout COMMIT_HASH
```

Starting a New Project

Creating the repository

In the project directory:

```
git init
```

You then need to add and commit files to import them.

Creating and populating a public repository

On the server:

```
mkdir project.git; cd project.git
git --bare init
```

On the initial client:

```
git remote add origin URL
git push origin master
```

- master is the main branch, the equivalent of trunk in SVN
- origin is a conventional name for the main remote repository
- there is no need for branches or tags folders, `git` has built-in support for them

Branching

Creating a new branch

```
git checkout -b NEW-BRANCH
```

Creates a new branch and immediately switch to it

Switch to another branch

```
git checkout BRANCH
```

Will change the working copy and index to match the given branch. New changes will now go into that branch.

Checking branches

```
git show-branch
```

Will show the different branches and a summary of their last changes

Merging

Merging from a branch

```
    git merge BRANCH
```

Reflect changes from given branch into the current branch. This may lead to conflicts, which you can then fix and commit.

Cherry-picking

Cherry-picking is selective merging.

```
    git cherry-pick COMMIT_HASH
```

Merges only the single commit identified by its hash.

Other Random Topics

Tags

With git, you can tag a release with

```
git tag TAG_NAME
```

This gives a symbolic name to the corresponding hash. You must push tags explicitly with

```
git push --tags [REPO]
```

Undoing things

To cancel changes introduced by a commit, use:

```
git revert COMMIT_HASH
```

This will undo the changes in the working copy, you can then commit the new version.

Other Random Topics (cnt'd)

Ignoring files

Just create a `.gitignore` file. On each line, the name of a file to be ignored by `git`. Those can contain wildcards.
You can have a `.gitignore` in a subdirectory.

Customizing git

She comes in colors:

```
    git config --global color.status auto
    git config --global color.branch auto
    git config --global color.interactive auto
    git config --global color.diff auto
```

Setting user info:

```
    git config --global user.name 'Your Name'
    git config --global user.email 'you@your.domain'
```

References

git	`http://git-scm.com/`
The git book	`http://git-scm.com/book` Freely available book
The git book (fr)	`http://alx.github.io/gitbook/` Old, incomplete French version

Rewriting history

- Have you ever dreamed of rewriting history? Yes, you can!
- You can merge microcommits, or mutually cancelling ones, into a more coherent set of commits, for a nicer log.
- Don't do it just to show off (never did, . . . well, maybe once or twice)
- Never do it if you pushed to a remote repository (or you'll seriously mess everything up)

Rewriting history

To reorder/merge/modify/. . . last 10 commits:

```
    git rebase -i HEAD~10
```

Giting Your Life

- You can also store binary files in a version control system (of course, merge becomes tougher)
- I store my whole `/home/soldani` directory under git
- Fun and useful
- I ignore sensitive and cache files
- I use multiple repositories to improve performance (git is not designed for huge and numerous binary files)

mgit

```
#!/bin/sh
while read dir; do
  echo "======== $dir"
  cd "$dir"
  git $*
done < ~/.mgit
```

Outline

Questions and (Possibly) Answers

?