Git & Mercurial: Distributed Configuration Management

CSCI 5828
Foundations of Software Engineering

2010-03-19

Dmitry Duplaykin | Paul Madden

Overview

- What is *configuration management*?
- A brief history of CM systems
- State of the art: Subversion
- Intro to distributed CM
- Git & Mercurial: Head-to-head comparison with Subversion
- Tips and Summary

What is *configuration management*?

- A number of aliases:

  - Pilone & Miles call it *version control*:

    “...a tool (usually a piece of software) that will keep track of changes to your files and help you coordinate different developers working on different parts of your system at the same time.”

  - Also known as *revision control* (e.g. rcs = Revision Control System), or *source code management* (e.g. http://git-scm.com).

What is *configuration management*?

- Typically offers features allowing:
  - Code check-out / check-in
  - Branching / Merging
  - Tagging
  - Recovery from mistakes
  - Display of specific code changes
  - Review of historical metadata

- Two architectural flavors:
  - Centralized (CVS, Subversion)
  - Distributed (git, Mercurial)

A brief history of CM systems

- 1972 – Source Code Control System (sccs)
  - The original. Proprietary Unix component.
- 1982 – Revision Control System (rcs)
  - SCCS alternative. Like SCCS, works on single files (not entire projects).
- 1990 – Concurrent Versions System (cvs)
  - Built on and extended RCS: entire project tree support, client/server network model, allowed concurrent work without sccs/rcs-style locking by supporting merging.
- 2000 – Subversion (svn)
  - A “better” CVS. In so many ways. Developer sought CVS pros without its cons.
- 2005 – Git & Mercurial: Distributed CMS
- Many, many others, both free and proprietary

State of the art: Subversion

- Repository creation / maintenance with *svnadmin* tool
  - Either in client/server mode, or on local file:// URIs
- Network access via
  - dedicated server
  - sshd
  - Apache
- Choice of storage databases
  - svn's own fsfs (simple)
  - Berkeley DB (more features)
- Offline diffs
- Copies (tags & branches) “are cheap”
- Updates stored as diffs & “skip deltas”
  - File reconstructed by sequentially applying diffs

State of the art: Subversion

- Client *svn* utility provides functions like:
  - checkout (co) – get files from repository
  - commit (ci) – upload changes to repository
  - copy (cp) – create branches / tags (among other uses)
  - revert – undo local changes to working copy
  - status – show what has changed in working copy or what updates are waiting in the repository
  - update (up) – bring working copy up-to-date with repo
  - merge – e.g. bring branch up-to-date with trunk
  - diff – compare revisions to each other / working copy
  - log – show historical log messages
  - wrappers for system commands like ls, rm, mv, cp, etc.

An svn workflow

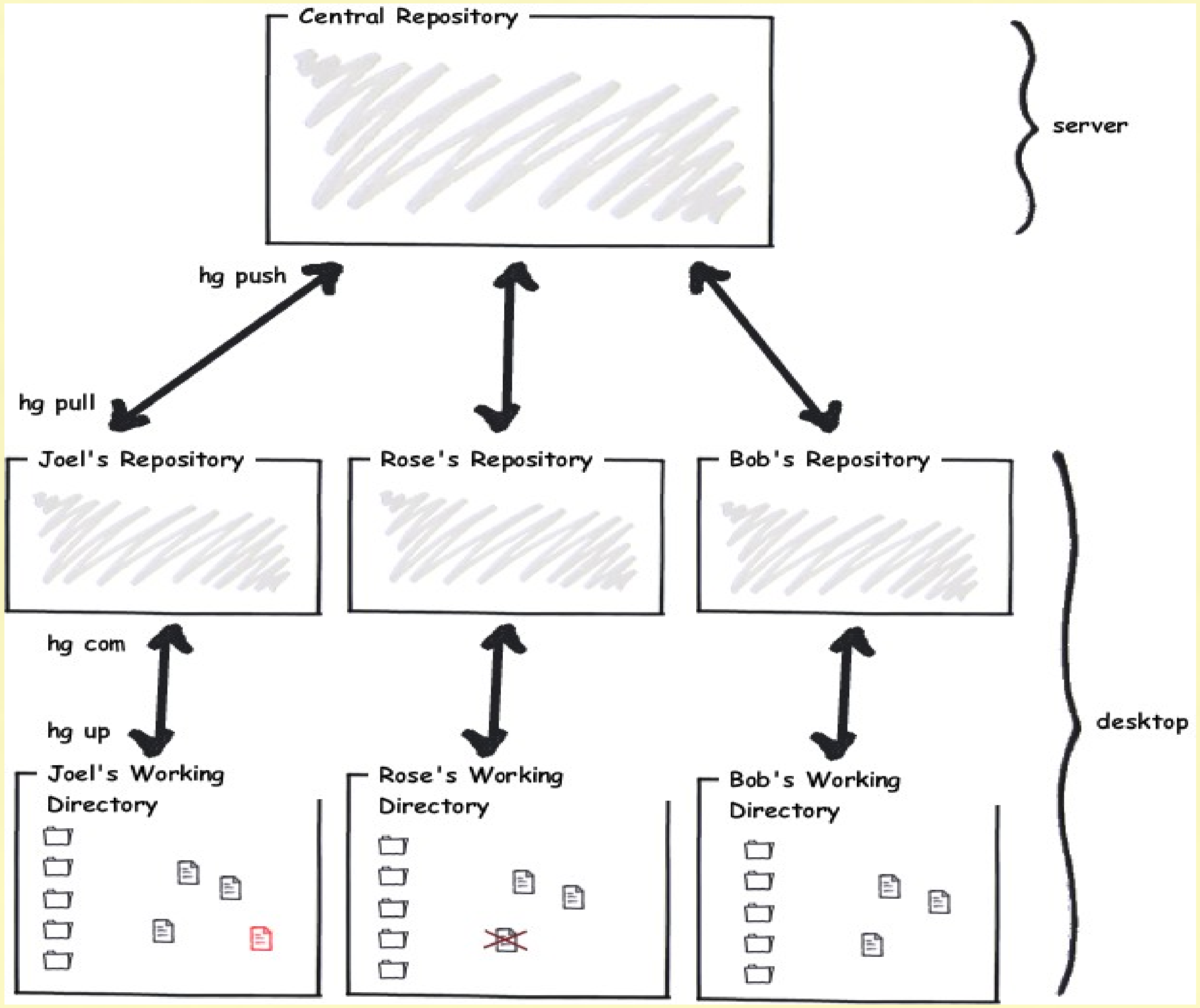
- *svnadmin create* repository (only server-side action)
- *svn import* initial set of files into repository (creates r1)
- *svn co* first working copy
- modify existing files, create new ones
- *svn add* new files to place under revision control
- *svn status* to see what has changed locally
- *svn revert* to undo changes
- *svn ci* new and changed files (creates r2)
- *svn log* to view log-message metadata
- *svn mv* to rename, *svn rm* to delete files
- svn ci these changes (creates r3)
- *svn co* an older (r2) working copy

An svn workflow

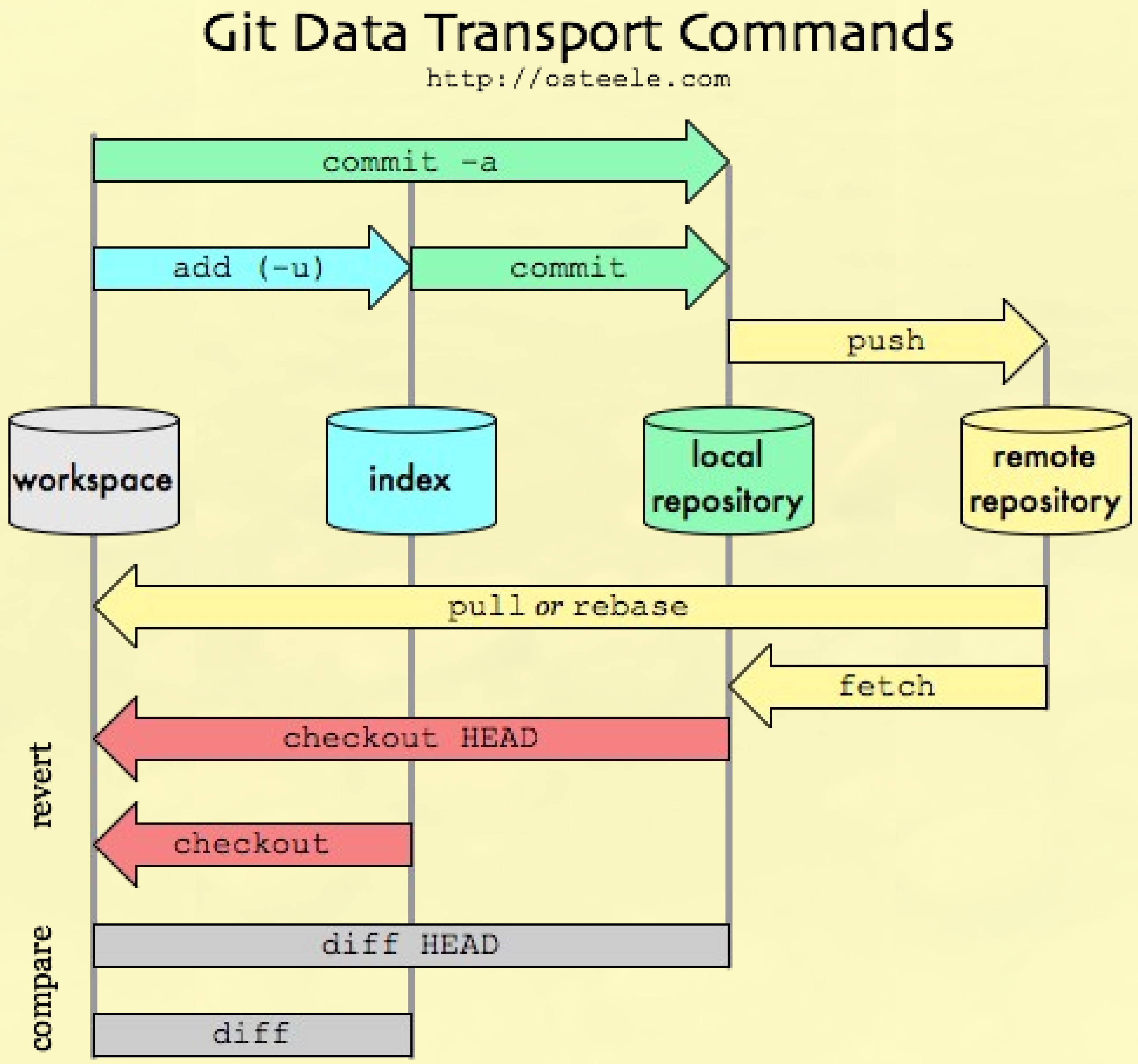
- *svn diff* r1 and r2 of modified file to see changes
- change file in 2nd copy of head, *svn ci*, look for changes in 1st copy of head, *svn up*.
- *svn cp* to create a tagged revision
- *svn cp* to create a branch
- *svn co* branch, make two commits, svn merge onto trunk, commit

Transition to distributed CM

- Conceptual model looks like this



More complexity with git



git background

- Developed by Linus Torvalds in 2005
  - Linux Kernel team needed a new CM solution after BitKeeper licensing changed
- Design requirements:
  - fast
  - distributed (no central server, every copy has complete development history in its .git directory)
  - secure (essentially impossible to “change history”)
- Git differences entire project trees, not individual files
- Revisions tracked with a SHA1 hash of information from the current project state
- Two storage locations:
  - changeable *index*
  - immutable *object database*

Projects using git

- Among many others:

Android	jQuery
Debian	Perl
Clojure	Samba
Digg	Ruby on Rails

- And of course, famously, the Linux kernel. Linus Torvalds developed Linux because Andrew Tannenbaum wouldn't let him use Minix. He developed git because BitKeeper revoked their free license. Lesson? Linux Torvalds will eat your lunch.

Mercurial background

- Started by Matt Mackall at the same time as git
- Properties:
  - Written in Python

    (95% in Python, core routines in C)
  - It's distributed
  - Fast
- Design features:
  - Uses SHA-1 hashes (like git)
  - Uses HTTP-based protocol
- All above matches “Google land religion”
- Will be referred to as hg, since commands start with it

Projects using Mercurial

- The list is quite long, the most famous ones are:

Mozilla	Symbian OS
OpenJDK	Go
OpenSolaris	GNU Octave
OpenOffice.org	Netbeans

- The Python developers have announced that they will switch from Subversion to Mercurial when hgsubversion – an extension that allows using Mercurial as a Subversion client and that has been under development since September 2009 – is released.

Head-to-head: Create a repository & initialize with files

svn

```
svnadmin create /repos/demo
mkdir -p import/branches import/tags import/trunk
cp source_files/* import/trunk
svn import ./import file:///repos/demo
```

git

```
cd source_files
git init
git add .
git commit # editor will open for commit message
```

hg

```
cd source_files
hg init
hg add
hg commit -m “Initial version”
```

Head-to-head: Get a working copy

In Subversion, we need to check out a working copy from the repository...

svn

```
svn co file:///repos/demo/trunk wc
```

git

We already have a versioned working copy!

hg

We have it!

Head-to-head: Edit & add files, view changes

svn

```
svn add fruit
svn status # concise view
svn diff # show actual deltas
```

git

```
git add fruit
git status # concise view
git diff # show actual deltas
```

hg

```
hg add fruit
hg status # show changed, added, deleted files
hg diff # show actual deltas
```

Head-to-head: Revert changes

We've changed our minds about changing *numbers*...

svn

```
svn revert numbers # to checked-out revision
```

git

```
git checkout numbers # fetches from index
git checkout HEAD number # from database
```

hg

```
hg revert numbers
# What if you had committed? Use this
hg rollback
# It will help, but only if you haven't pushed
# this change to central repository. Then you
# have to think about it...
```

Head-to-head: Commit changes

Let's commit the changes we've decided to keep:

svn
```
svn ci
svn up # update working copy's revision info
```

git
```
git add letters fruit # stage to index
git commit # commit index to database
```
 -or-
```
git commit -a # stage + commit in one
```

hg
```
hg add letters fruit
hg commit
```

Head-to-head: View current status & log

log commands show most recent actions first...

svn

```
svn info # shows revision number
svn status # shows local changes
svn log # shows commit messages
```

git

```
git status # shows changes & pending commit info
git log # git doesn't have revision numbers
```

hg

```
hg status # shows local changes
hg log # revision history
hg parent # changeset you're working from
```

Head-to-head: Remove and rename files

```
svn
svn rm fruit
svn mv numbers digits # an add + a delete
svn commit
svn up

git
git rm fruit
git mv numbers digits
git commit -a

hg
hg remove fruit
hg rename numbers digits
hg com -m “Remove and rename.”
```

Head-to-head: Get another project-head working copy

```
svn
cd ..
svn co file:///repos/demo/trunk wc2

git
cd ..
git clone source_files wc2

hg
cd ..
hg clone source_files wc2
```

Head-to-head: Working copy hierarchy

Create a chain of working copies, each of which pushes its changes to the working copy from which it was cloned.

svn

Impossible!

git

```
git clone wc2 wc3
# “git push” in wc3 -> wc2
# subsequent “git push” in wc2 -> source_files
```

hg

```
hg clone source_numbers crazy_numbers
# crazy_numbers is a playground for experiments,
# “hg push” by default will push changes
# back to source_numbers
```

Head-to-head: Moving in time

svn
```
svn co -r2 file:///repos/demo/trunk wc3
# Creates duplicate working copy
```

git
```
git checkout <SHA1-ID>
git checkout HEAD^  # set to parent
git checkout HEAD^^ # or grandparent
git checkout master # return to latest
```

hg
```
hg update -r <n> # set working copy to revision n
hg update -r 103994
# "...and get some really cool anti-gravity sci-
# fi futuristic version of your source code"
# (Joel Spolsky)
```

Head-to-head: Show differences between 2 revisions

svn
```
svn diff -r2:3 file:///repos/demo
# show deltas between revisions 2 and 3
```

git
```
git diff <SHA1-ID-1> <SHA1-ID-2>
```

hg
```
hg diff -r 3:5 fruit
```

Head-to-head: Apply updates to working copy

```
svn
svn update # maybe deal with merge conflicts

git
git fetch origin # just get updates
git merge origin # apply updates
 -or-
git pull # fetch and apply (merge) updates

hg
hg update # updates from local repo (or clone)
 -or-
hg pull # updates from central repo
hg update # apply updates
```

Head-to-head: Create tag / Retrieve tagged version

In svn, tags and branches are just copies, and the difference is a matter of convention.

svn

```
svn copy file:///repos/demo/trunk \
 file:///repos/demo/tags/my_tag
svn co file:///repos/demo/tags/my_tag
```

git

```
git tag -m “message” <tag_name> <SHA1-ID>
git checkout <tag_name>
git show <tag_name> # show info about commit,
                    # including “message”
```

hg

```
hg [-l] Version_1.1 # create tag locally/globally
hg update Version_1.1 # move to the tag
```

Head-to-head: Branching

svn
```
svn copy file:///repos/demo/trunk \
 file:///repos/demo/branches/my_branch
svn co file:///repos/demo/branches/my_branch
```

git
```
git branch <branch_name> <SHA1-ID> # create
git checkout <branch_name> # switch to branch
```

hg
```
# Branch?
# In Mercurial land it's another repository
hg clone source_numbers call_it_branch_if_U_want
```

Head-to-head: Apply changes from branch onto trunk

svn

```
svn merge -r3:6 \
 file:///repos/demo/branches/my_branch .
```

git

```
git merge [head | SHA1-ID]
git pull . [head]
```

hg

```
# Continuing previous example:
# Apply to local repository in /source_numbers
/call_if_branch_if_U_want> hg push
# Or straight to central
/call_if_branch_if_U_want> hg push
http://user.host.com:8000/
```

git tips

- Commit often
  - Unlike with Subversion, nobody sees your commits until you push them to a central repository
- Use the index as your staging area
  - Undo changes without creating log history or new commit IDS
  - *git diff* no longer shows deltas for items *git add*'ed to the index (*git diff –cached* shows those)
- Use tags
  - Unlike with Subversion, tags can apply to multiple commits, and a commit can have multiple tags
  - Better than remembering (or looking up) SHA1 hashes!

git summary

- A simple git workflow may look like this:
  - Coding, file operations, etc.
  - `git status         # to see what's changed`
  - `git diff [file] # to see change details`
  - `git commit -a -m “message” # to commit`
- Lots of powerful, advanced operations are available.

hg tips

- Joel Spolsky, who looks like a big fan of hg, teaches this:
  - Feel free to branch: It's pain-free
  - Mercurial is better than Subversion, so use it in your team
  - Use it if you working by yourself
  - Do everything the “Mercurial way”
- “Subversion Story #1”:
  - Six programmers around a single computer working for two weeks trying to manually reapply every single bug fix from the stable build back into the development build

hg summary

- Workflow with Mercurial should look like this:

  1. Get the latest version that everyone is working off of:

```
hg pull
hg up
```

  2. Hack a bunch
  3. Commit OFTEN locally
  4. Repeat 2-3 until you’re ready to share:

```
hg pull   # to get others' changes (if any)
hg merge  # better test after this
hg commit # save your changes
hg push   # finally share
```

Epilogue

“And here is the most important point, indeed, the most important thing that we’ve learned about developer productivity in a decade. It’s so important that it merits a place as the very last opinion piece that I write, so if you only remember one thing, remember this:

...

[Distributed revision control] is too important to miss out on. This is possibly the biggest advance in software development technology in the ten years I’ve been writing articles here.”

(Joel Spolsky, March 17, 2010)

Acknowledgements & References

- *Head First Software Development*, Dan Pilone & Russ Miles, O'Reilly Media
- http://cssc.sourceforge.net/old-cyclic/sccs.html
- http://www.cs.purdue.edu/homes/trinkle/RCS/
- http://svn.apache.org/repos/asf/subversion/trunk/notes/skip-deltas
- http://www.eecs.harvard.edu/~cduan/technical/git/git-1.shtml
- http://hginit.com/index.html
- http://mercurial.selenic.com/guide/
- http://www.versioncontrolblog.com/2007/06/26/video-bryan-osullivan-mercurial-project/
- http://code.google.com/events/io/2009/sessions/MercurialBigTable.html
- Wikipedia