Distributed Version Control Systems (DVCS) & Mercurial

Lan Dang
SCaLE 13x
2015-02-22

http://bit.ly/1EiNMao

About the Talk

- Prepared originally for SGVLUG monthly meeting about a year ago
- Product of intense research for new task at work
- Structured so people of all skill levels could learn something
- Slides meant to serve as a reference
- Google Drive location: http://bit.ly/1EiNMao

Overview

- Version Control Systems
- DVCS Concepts
- Mercurial basics
- git vs Mercurial
- Mercurial under the hood
- Case Studies
- References

My background

- Subversion -- very basic knowledge
- git -- for small single-user projects
- Mercurial -- limited experience as user, moderate experience as administrator
- Focused on use on Linux/Unix; no experience on Windows or with tools like TortoiseHg

About Version Control Systems

Version control systems (VCS) gives you the power to

- time travel
- more easily collaborate with others
- track changes and known good states

Eric Raymond defines VCS as a tool that gives you the capabilities of reversibility, concurrency, and annotation

http://www.catb.org/esr/writings/version-control/version-control.html

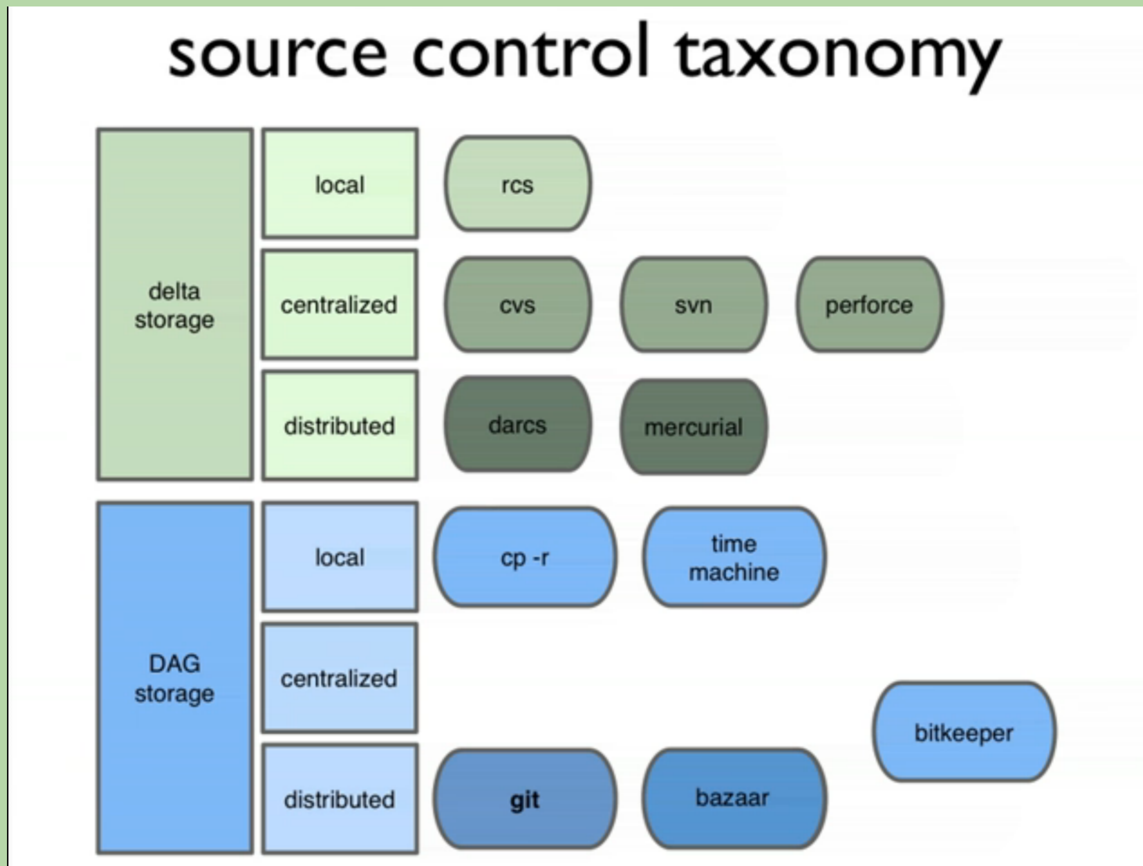
History of Version Control

According to Eric Raymond, and summarized by Eric Sink: there are three generations of VCS

Generation	Networking	Operations	Concurrency	Examples
First	None	One file at a time	Locks	RCS, SCCS
Second	Centralized	Multi-file	Merge before commit	CVS, SourceSafe, Subversion, Team Foundation Server
Third	Distributed	Changesets	Commit before merge	Bazaar, Git, Mercurial

From Chapter 1 of "Source Control By Example" by Eric Sink:
http://www.ericsink.com/vcbe/html/history_of_version_control.html

Source Control Taxonomy



Lifted from Scott Chacon's "Getting Git" screencast: http://vimeo.com/14629850 (look at time 5:06)

DVCS at work

"Centralized vs Distributed Version Control in 90 seconds" by Intland Software

http://youtu.be/_yQlKEq-Ueg

Uses for DVCS

- Keep track of
  - source code
  - scripts
  - configuration files
  - notes (my personal use case ;)
  - other files (beyond scope of this talk)
- Synchronizing files across multiple computers
- powering a website or wiki
  - See github
- working offline
  - Don't need network connection to make changes

DVCS Basic Concepts

- The heart of Version Control Systems is the **repository**, which contains the history and versions of all the files (aka **changesets**) under version control.
- In DVCS, each developer has a copy of the entire repository. It is usually **local** to their **working directory**.
- Modifications are not recorded in the **repository** until they have been **committed** or **checked in**. These **commits** are known as **changesets**.

DVCS Basic Concepts

- **Changesets** are identified by a **changeset id**, usually based on a hash of the **changeset**.
- **Changesets** belong to one or more **parents**, these being the **changesets** that represented the state of the repository before the latest commit.
- The **working directory** is where the plain, editable files are. It can be **updated** to a particular **changeset** in the repository's history, or to the most current changeset.

DVCS Basic Concepts

- **Merging** is the bread and butter of DVCS, as that's how **concurrency** is supported. It is therefore supposed to be painless and fast.
- However, it is important to synchronize often with **remote** or **upstream** repositories to avoid surprises and too many **merge conflicts**.
  - **Pull changesets** from **remote repositories** into **local working directory**
  - **Merge**
  - **Commit** the merge
  - **Push changesets** to **remote repositories**

DVCS Basic Concepts

- **Branches** represent parallel lines of development. They may be required for long-running feature changes, support of previous releases, or just for the sake of experiment.

DVCS Basic Concepts in summary

- Repositories are **cloned**, rather than **checked out**.
- Developers do their development in their **local repositories** and may **commit changesets** as often as they wish
- Developers should periodically synchronize with upstream (**remote**) repositories by **pulling** in upstream **changesets**, **merging**, and then **pushing**.

Getting started with Mercurial

- Configuration is done through user-level ~/.hgrc
- Here is a basic config

```
[ui]
username = <your_username>
```

Setting up Mercurial prompt

hg-prompt adds Mercurial-specific information to your normal shell prompt. This gives you a visible indicator of the state of your working directory. This is important if you do a lot of branching.

Download the Python extension somewhere on your computer

http://mercurial.selenic.com/wiki/PromptExtension

Setting up Mercurial prompt (.hgrc)

Add the following line to your .hgrc:

```
[extensions]
prompt = (path to)/hg-prompt/prompt.py
```

Setting up Mercurial prompt (bash)

Add following lines to .bashrc

```
# Add Mercurial prompt
# This is the prefix that appears before your normal prompt
hg_ps1() {
    hg prompt "({branch})[{status}] " 2> /dev/null
}

# Prepend dynamic Mercurial prompt to normal prompt
export PS1='$(hg_ps1)[\u@\h \w]$ '
```

Setting up Mercurial prompt (csh)

Create "hgprompt.csh" and place it in your home directory

```
setenv HG_PROMPT 'hg prompt "\({branch}\)[{status}]"'
set prompt="`$HG_PROMPT` [`whoami`@`hostname -s` $cwd]$ "
```

Add the following to .cshrc or .tcshrc. Edit the line depending on location of hgprompt.csh

```
alias precmd "source ~/bin/hgprompt.csh"
```

NOTE: precmd is a special alias that is run every time a command is executed in csh

Mercurial basic commands

- Create repository

hg init

- Add files to repository (aka register them for version control)

hg add <file(s)>

- Remove files from repository (aka stop tracking them)

hg remove <file(s)>

Mercurial basic commands

- Rename or move files in repository

hg mv <file> <newfilename>

NOTE: It is important to do this, so that you can more easily track changes to files in the repository, even if they have moved around or been renamed.

- Commit changes (aka add changeset to repository)

hg commit -m "<commit message>"

- Check for files to be added or committed in working directory

hg status

Mercurial basic commands

- View the current state of working directory (last changeset, branch, commit state)

```
hg summary
```

- View repository history/log

```
hg log | head
```

- View repository history for a particular branch

```
hg log -b <branch>
```

Mercurial basic commands

- Update working directory to latest changeset in default or current branch

hg update

- Update working directory to a particular branch

hg update <branch>

- Update working directory to main branch. (In Mercurial, this branch is called "default")

hg update default

Mercurial basic commands

- Clone existing repository

hg clone </path/to/repo>

- See if there are new changesets in remote repo

hg incoming

- See if there are changesets in your local repository that need to be synchronized

hg outgoing

Mercurial basic commands

- Pull changesets from remote repository

hg pull

- Merge changeset into current working directory.

hg merge

hg commit -m "YARM"

NOTE: Mercurial should prompt you if merging is required. You must **commit** merges for them to be recorded in the repository.

Mercurial basic commands

- Push changes from local repository to remote repository

hg push

- Branch

hg branch <branchname>

<make changes>

hg commit -m "Branch <branchname> to do some development on X feature"

NOTE: In Mercurial, named branches are just labels that are attached to a changeset. They don't fully exist until you commit the changeset. Once used, the branch names cannot be changed.

Mercurial basic commands

- View the tags in the repository. Tags are basically meaningful aliases for particular changeset ids

```
hg tags
```

- Tag the current changeset

```
hg tag <tagname>   # tags current changeset
```

NOTE: Tags are kept in .hgtags file, which is version controlled. The above command automatically updates .hgtags and commits the change. However, it requires that you are on the latest changeset (aka head) of your particular branch.

Mercurial basic commands

- Tag a particular changeset with a meaningful name. This version of the command gives you more control over what changeset is tagged and what the tag commit message says.

```
hg tag -r <changeset id> -m "Tagging <changeset id> as tag <tagname>, per release document XYZ" <tagname>
```

Mercurial basic commands

- Update working directory to a particular changeset (or point in history)

hg update -r <changeset id>

hg update -r <revision number>

NOTE: Revision numbers are provided as a convenience for the user. They are basically aliases for changeset ids and local to that particular repository. The revision number may not map to the same changesets in other repositories.

git vs Mercurial (origins)

- **Bitkeeper** -- commercial DVCS used for Linux kernel development. Started at 1999; adopted for Linux kernel development circa 2002. No longer available to Linux kernel developers starting in 2005.
- **git** (C & Perl) -- written by Linus Torvalds in 2005 to replace Bitkeeper
- **Mercurial** (Python)-- written by Matt Mackall in 2005 to replace Bitkeeper
- Both **git** and **Mercurial** are being actively developed and reaching feature parity.

git vs Mercurial (git)

- git
  - written in C and Perl.
  - fast performance and good merging algorithm
  - integrity of repository despite untrustworthy sources (the hash is key)
  - clean history through rebase
  - super lightweight branches
  - staging area for commits
  - extensions can be written in any language
  - exposes more of the complexity to the user
  - not always intuitive to use

git vs Mercurial (Mercurial)

- Mercurial
  - written in Python
  - history is immutable, except through extensions. (modification of repository history strongly discouraged.)
  - less complicated interface, easier to use
  - More focus on cross-platform support and tool integration, i.e., much easier to set up on Windows
  - Extensible through plugin architecture
  - Explicit focus on backwards-compatibility, except that Mercurial API may change between versions.
  - branches as repositories, named branches
  - "clean history" through patchsets rather than rebase.
  - merge mostly relies on external tools like meld, kdiff3

Under the Hood

"Version control tools are more like cars than clocks.

" Clock users have no need to know how a clock works behind the dials. We just want to know what time it is. Those who understand the inner workings of a clock can't tell time any more skillfully than the rest of us.

"Version control tools are more like cars. Lots of people, including me, use cars without knowing much about how they work. However, people who really understand cars tend to get better performance out of them."   -- Eric Sink

http://www.ericsink.com/vcbe/html/internals.html

NOTE: Highly recommend understanding how the tools work, so you will understand why some things are easy and some things are hard.

Under the Hood: : How DVCS ensures integrity of repository

In both git and Mercurial, history is represented as a **Directed Acyclic Graph (DAG)**.

Changesets represent the state of the system at a particular time. Changesets may include a number of metadata, but all changesets refer to 0,1, or more **parents**.

Under the Hood: : How DVCS ensures integrity of repository

- **Changeset ids** are **cryptographic hashes** of the changesets. Therefore, changesets cannot be modified without modifying changeset ids.
- Since changesets include their **parents' changeset ids**, the **changeset id** ensures integrity through root of changeset graph
- This means a known **changeset id** can guarantee the **integrity** of the codebase up to that point in history.

Under the Hood: Merging

The way that the merges resolve differences is to look at what changed in the changeset and what changed in your current branch since the common branch point, and it will use that to decide which parts of the file change.

http://stackoverflow.com/questions/9532823/mercurial-branch-specific-changes-keep-coming-back-after-dummy-merge

http://stackoverflow.com/questions/9598704/consequences-of-using-graft-in-mercurial

This table describes how the 3-way merge works. "Ancestor" refers to the common branch point, "local" to your working directory, and "other" to the changeset that you are merging with.

Ancestor	**Local**	**Other**	**Merge?**
old	old	old	old (nobody changed the hunk)
old	old	new	new (they changed the hunk)
old	new	old	new (you changed the hunk)
old	new	new	new (hunk was cherry picked onto both branches
old	foo	bar	<!> (conflict, both changed hunk but differently)

Under the Hood: YARM

"Yet Another Random Merge"

Due to concurrent commits in different repositories, merging is required to integrate all the changes into a common, somewhat linear commit history.

This means that you will often have to merge, even if you haven't been editing the same files. **Think of the merge as a repository-level merge, rather than a file-based merge.**

Case Studies: Corrupt Repository

Issue: User was unable to pull from central repository after colleague pushed his changes.

Investigation: Colleague had accidentally corrupted central repository on push. His repository had become corrupted somehow.

Resolution: Caught this early. Got agreement from stakeholders to rollback the change to central repository.

Gave instructions on how to recover local repo:

1. backup local repo
2. make fresh clone
3. copy over changes to clone
4. commit changes
5. synchronize with central repo.

Case Studies: Corrupt Repository

Mitigation: The repository information is stored in hidden directory .hg

Certain file system commands can corrupt the contents of .hg -- say "find" with "rm"

Periodically run the following command to verify integrity of repository before pushing.

hg verify

Case Studies: Repo Reorganization

Issue: Team Lead wanted to move a directory from one repository to another.

Resolution: Repository migrations or reorganizations require the use of the "convert" extension, which is used to migrate from other VC tools to Mercurial, or it can be used to subset Mercurial repositories.

http://mercurial.selenic.com/wiki/ConvertExtension

Case Studies: Repo Reorganization

The "convert" extension is distributed with Mercurial. Therefore, it is easy to activate it for use:

Add the following line to ~/.hgrc under

```
[extensions]
convert=
```

Case Studies: Repo Reorganization

The directory being migrated had been renamed several times since it was created. In order to migrate **all** the relevant changesets, needed to track down directory name variants.

hg log <directory name> # Look for oldest changeset involving this directory

hg log -v -r <oldest changeset> # Verbose log shows files involved in commit.

See if there was a possible rename. Record the directory name. Repeat above steps until you find the original commit that created the directory.

Case Studies: Repo Reorganization

Generate a filter (filemap) to determine changesets to be migrated. In this case, I just had to indicate the directory name. I usually do it from oldest to newest name.

filemap.txt contents:

exclude .

include <first dir name>

include <second dir name>

include <third dir name>

Case Studies: Repo Reorganization

Generate a filter (branchmap) to determine how to map the branch names in the changesets. Basically, because I did not want changesets to spuriously show up in existing branches in the new repository, I mapped them all to the default branch.

branchmap.txt contents:

<old branchname> <newbranchname>

Case Studies: Repo Reorganization

Set up the two repositories before the conversion. Clone both repositories and update the working directory to the tip.

```
hg convert <oldrepo> <newrepo> --filemap filemap.txt --branchmap branchmap.txt
```

Because of the branchmap, now have multiple heads for default. Need to do dummy merges, so it is only one head.

```
hg heads -c default   # Gives a list of heads for default branch
hg update tip
hg merge <extra head>
hg revert -a -r tip --no-backup  # reverses effect of above merge
hg commit -m "MIGRATION: Dummy merge to resolve extra head (changeset <changeset id>) from migration"
```

Case Studies: Repo Reorganization

In the old repo, remove the directory that was migrated

cd <oldrepo>

hg remove <migrated dir>

hg commit -m "Removing <migrated dir>, which can now be found in <newrepo>"

Case Studies: Repo Reorganization 2

Issue: Team Lead wanted to extract a directory from one repo and make it its own repo.

Resolution: Used the "convert" extension again. Slight changes to convert subdirectory to top level of repository using "rename" directive. In addition, no issues with merging or need to remap branch names.

filemap.txt contents:

exclude .

include <dirname 1>

include <dirname 2>

rename <dirname 2> .

Case Studies: Repo Reorganization 2

Initialized a new repo in the central repo area. Cloned both the new repo and the old repo. Updated both to tip.

hg convert <old repo> <new repo> --filemap filemap.txt

Removed directory from old repo

cd <old repo>

hg remove <old dir>

hg commit -m "Migration: Removing <old dir>, which has now been promoted to top level repository <newrepo>"

Case Studies: Strip Files from Repo

Issue: Developer accidentally checked in some large files into repository. It was not the appropriate use of the VC tool and resulted in slow performance for everyone using the repository. Needed to strip the files from repository's history.

Resolution: Once again, this requires the "convert" extension. However, this involved the editing of the central repository, so had to get the stakeholders to arrange a quiet time and agree to abandon old repository and make a fresh clone.

Case Studies: Strip Files from Repo

Determine the files to strip. This needs to be in relation to top level directory of repository.

```
cd <repo>
find ./ -name "*.mat" -or -name "*.bin" -or -name ".svn" | \
  while read file; do echo exclude $file; done | \
  tee ../filemap.txt
```

Archived old repo. Make a fresh new directory.

```
hg convert <archived repo> <new repo> --filemap filemap.txt --sourcesort
```

NOTE: If there are filenames with spaces in them, just quote them.

```
exclude "<filename with spaces>"
```

Case Studies: Strip Files from Repo

Mitigation: You can prevent accidental commits of certain filetypes using .hgignore in the top level of the repository.

.hgignore contents:

```
syntax: glob
*~
*.o
*.pyc
*.mat
*.bin
```

Case Studies: Strip Files from Repo

Binary files may cause the repository format to expand greatly with each new version, due to the storage format.

Even with text files, there are limitations on individual file size.

There appears to be a 2GB limitation due to internal limitations such as wire protocol, etc.

There is huge memory overhead as Mercurial handles files as huge Python strings in memory. The memory requirement is estimated to be 3-5x the size of the filesize.

There are limitations on address space, according to platform, which further restrict individual file size.

- 400MB limitation (32-bit Windows)
- 1GB limitation (32-bit Linux)
- 2GB limitation (64-bit Mercurial).

http://mercurial.selenic.com/wiki/HandlingLargeFiles

Case Studies: Migrating Changesets into Branch

Issue: Team lead asked that certain changesets on the default branch be reverted and the changesets moved to an existing branch.

Resolution: Revert the default branch to the desired state using "hg revert". Because so few changesets were involved in both default and the branch, decided to use patches to resolve the issue.

Case Studies: Migrating Changesets into Branch

Revert default branch to desired state. This would be the parent of the oldest changeset to be backed out.

hg revert --all -r <desired changeset>

hg commit -m "Reverting default branch back to state before changesets <changeset ids> were commited, as those changesets belong on branch <branch name>"

Case Studies: Migrating Changesets into Branch

Export all the changesets that need to be on the new branch as patches. Order matters, so number them from oldest to newest

```
hg export -o ../patch1 <oldest changeset>
hg export -o ../patch2 <next oldest changeset>
hg export -o ../patch3 <third oldest changeset>
hg export -o ../patch4 <last changeset, which was on branch>
```

Case Studies: Migrating Changesets into Branch

Revert the new branch to right before it branched, i.e., the parent of changeset exported as patch4.

```
hg update <new branch>
hg revert --all -r <branch point>
hg commit -m "<branch> FIX: Reverted branch <branch> to rev <branch point>
to change where branch starts, so we can replay all commits belonging to
branch"
```

Verify that <new branch> head is now identical to default branch head

```
hg diff -r default
```

Case Studies: Migrating Changesets into a Branch

Apply the exported patches to <new branch>. Because we used Mercurial to generate the patches, it retains the original commit message and author, so it's very much like replaying commits.

```
hg update <new branch>
hg import ../patch1
hg import ../patch2
hg import ../patch3
hg import ../patch4
```

References

"Understanding Version-Control Systems (DRAFT) by Eric Raymond

http://www.catb.org/esr/writings/version-control/version-control.html

NOTE: This is a draft, and some things may be inaccurate or out-of-date. The writing is very formal, almost academic in nature. It really gets into the history of VCS, from its origin in the 1970s.

References

Eric Sink has two resources:

"Source Control HOWTO" (focuses on "centralized" version control tools)

http://www.ericsink.com/scm/source_control.html

"Version Control by Example" (with more focus on decentralized or distributed version control tools)

http://www.ericsink.com/vcbe/

References

"So you want to know more about Distributed Version Control," e-mail from Lan Dang sent to SGVLUG listserv listing a series of videos with annotations:

http://sgvlug.net/pipermail/sgvlug/2012-October/010206.html

NOTE: Scott Chacon link expired. Look for video "Getting Git" by Scott Chacon, a screencast based on talk given at RailsConf 2008 here: http://vimeo.com/14629850

References

"Hg Init Subversion Re-education" by Joel Spolsky

http://hginit.com/00.html

NOTE: This is an entertaining and illustrated tutorial explaining conceptual differences between Subversion and Mercurial. Also gives you a handle on Mercurial basics.

References

"Mercurial for Git Users" from Mercurial wiki

http://mercurial.selenic.com/wiki/GitConcepts

NOTE: This posits that Mercurial and Git are pretty much the same thing, with different philosophies and ways of accomplishing the same tasks. Please note that this is doing a heads-on comparison of particular versions of Mercurial and git; the two of them evolve so much, it is necessary to mark the versions.

References

"The Real Difference Between Git and Mercurial" by Xentac (Jason Chu)

http://xentac.net/2012/01/19/the-real-difference-between-git-and-mercurial.html

NOTE: This article is a little over a year old. It's interesting in describing the origins of Git and Mercurial and the different architectural and design decisions made. Comments are interesting too.

References

"Understanding Mercurial" from Mercurial wiki

http://mercurial.selenic.com/wiki/UnderstandingMercurial

NOTE: An illustrated view of Mercurial basic concepts. It makes so much more sense when you can view things as a directed acyclic graph.

References

"Mercurial Working Practices"

http://mercurial.selenic.com/wiki/WorkingPractices

NOTE: It's always good to understand the typical workflows with a tool, as this will enable you to work with the tool's strengths. Also, tool enhancements and bugfixes are probably focused on people's typical use cases.

References

"A Guide to Branching in Mercurial" by Steve Losh

http://stevelosh.com/blog/2009/08/a-guide-to-branching-in-mercurial/

NOTE: The article is old, but I believe it is still relevant. It's a detailed description of different ways to branch in Mercurial, and how it compares to git.

References

"Mercurial Kick Start" by aragost Trifork, a Mercurial consulting company

http://mercurial.aragost.com/kick-start/en/

NOTE: This gives a lot of examples for different tasks and workflows.

References

"Mercurial: The Definitive Guide" by Bryan O'Sullivan

http://hgbook.red-bean.com/

NOTE: The online version of the book has a comment system so folks can directly add feedback for each section of the book. The source code for the book is a Mercurial repository, so people can either fork the book, or they can make modifications and send them back to the author.

Questions?

Lan Dang
l.dang@ymail.com