CS 520/620: Introduction to Version Control Systems

Tommy Nguyen
What is Version Control?

● Version control is a system that records changes to a set of files over time so that one can go back to look at specific versions later

● Simple Example:
  ○ Jill writes a research paper and stores it in version control - Version 1.0
  ○ Jill edits research paper to correct grammatical mistakes - Version 1.1
  ○ Jill discovers new findings and completely rewrites her paper - Version 1.2
  ○ Jill’s findings are wrong - Revert back to Version 1.1
Why use version control?
Why use Version Control? Story Time!

Disclaimer: I am in no way intentionally pushing religion onto anyone. This bit is merely a joke to encourage saving and using version control in software.

Jesus Saves

Jesus and Satan have a discussion as to who is the better programmer. This goes on for a few hours until they come to an agreement to hold a contest, with God as the judge.

They sit themselves at their computers and begin. They type furiously, lines of code streaming up the screen, for several hours straight. Seconds before the end of the competition, a bolt of lightning strikes, taking out the electricity. Moments later, the power is restored, and God announces that the contest is over.

He asks Satan to show what he has come up with. Satan is visibly upset, and cries, "I have nothing. I lost it all when the power went out."

"Very well, then," says God, "let us see if Jesus fared any better."

Jesus enters a command, and the screen comes to life in vivid display, the voices of an angelic choir pour forth from the speakers. Satan is astonished.

He stutters, "B-b-but how? I lost everything, yet Jesus' program is intact. How did he do it?"

God smiled all-knowingly, "Jesus saves."
Centralized Version Control (SVN)

- The idea of having “central” copy of a project where programmers will commit changes there
- Each programmer gets a working copy. As soon as you commit, then can update their copies
Example: Alice and Bob

Suppose both Alice and Bob start with the same file:

```
#!/usr/bin/env python
print "Hello world!"
```

Now Alice decides that the message should have a comma in it:
```
#!/usr/bin/env python
print "Hello, world!"
```

Meanwhile, Bob decides that the message is too overstated and wants to tone it down a bit:
```
#!/usr/bin/env python
print "Hello world."
```

Now Alice commits, so the current version in the repository agrees with her. Bob tries to commit and gets a message about not having the current version. So he updates and subversion informs him that there is a conflict. Subversion will ask what to do; sometimes you can resolve this right away, but assume Bob answers p to postpone (this process varies from version to version).
Bob's working copy of the file has been edited with the conflict markers:
```
#/usr/bin/env python
<<<<<<<<<<<  .mine
print "Hello world."
=======
print "Hello, world!"
>>>>>>> .r4
```

The <<<<<<<<< and >>>>>>>>> denote a conflict, and ======= separates the versions. The top version is Bob's local copy, and the bottom version is the version that Bob pulled from Alice (.r4 is the current revision in the repository).

Bob then looks at what Alice changed and decides how he wants to integrate the changes. Perhaps like this:
```
#/usr/bin/env python
print "Hello, world."
```

He manually makes this change in an editor, saves the file, and then commits.
Problems with Centralized Version Control

- What happens when there is no internet connection?
- What happens when wanting to explore project history?
- What happens when one wants to implement a large change?
Distributed Version Control (Git)

- The idea of having each user having their own repository and working code with a central repository.
- When committing, no one has access to your changes until you push to the central repository.
- You do not see others’ updates until you pull their changes.
Distributed Version Control (Git) Cont.

For others to see your changes:

a. You commit
   - `git add /*Insert directory or file here*/`
   - `git commit -m "*Insert Commit Message Here*"

b. You push
   - `git push *remote *branch`

c. They pull
   - `git pull *remote *branch`

d. They update
   - In git, pulling and updating are both one operation in git pull.
Committing

- Generally, we add the files that we want to commit to the repository using the command: `git add`
  - `git add *directory_name*`
  - `git add *file_name*`

- As a general rule of thumb, we don’t add binary files so to do that, we can create a `.gitignore` file to ignore certain files:
  - `*.pyc`

- Once we’ve added the files we want to commit, we type: `git commit -m “Commit Message”`

- Afterwards, we can then push to a repository: `git push *remote* *branch_name*`
Remotes

- When working with multiple remote repositories, you might specify different servers that you want to update the code on.
- Remote for GitHub
  - If you create a repository from GitHub, your remote name is usually `origin`.
- Remote for BitBucket
- [http://gitref.org/remotes/](http://gitref.org/remotes/)
Relationship between local repositories and remotes
Branches

- Master branch is the main branch where working commits are made
- To add a new feature, we often created branches to test and develop our branches without modifying the behavior of our main product. This allows for software stability

Git Commands:
- `git checkout -b *new branch`
- `git checkout *branch`
Conflicts

- When two users make a simultaneous change to the same line of a file and attempt to pull or merge these changes to another branch
- When possible, git will merge changes automatically without needing to do anything
- However, when conflicts arise, the user needs to choose which lines to keep.
Merging

- When a user wants to merge their changes to a branch they:
  a. Check out to the desired branch: `git checkout *desired_branch`
  b. Merge their branch into the desired branch: `git merge *merging_branch`
  c. Resolve any merge conflicts
  d. If there are merge conflicts, commit those fixes and push
Looking Through History

- Use git log to look through our commits:
  - “git log” gives us a list of commits in the terminal. Navigation is similar to looking through man pages in Linux. Use up/down or page up/page down to scroll and q to quit
- Use git checkout to dive into a previous commit and compile it:
  - All commits have a hash
  - To checkout, we can use “git checkout *commit hash*”, where we can find the commit hash from using “git log” and see the commit for that entry. You often only need the first six characters of the commit hash
Git cheatsheet

- Check it out at: [https://www.git-tower.com/blog/git-cheat-sheet/](https://www.git-tower.com/blog/git-cheat-sheet/)

<table>
<thead>
<tr>
<th>CREATE</th>
<th>BRANCHES &amp; TAGS</th>
<th>MERGE &amp; REBASE</th>
</tr>
</thead>
</table>
| Clone an existing repository  
$ git clone ssh://user@domain.com/repo.git | List all existing branches | Merge branch into your current HEAD  
$ git merge <branch> |
| Create a new local repository  
$ git init | Switch HEAD branch | Rebase your current HEAD onto <branch>  
Don't rebase published commits  
$ git rebase <branch> |
| LOCAL CHANGES | LOCAL CHANGES | LOCAL CHANGES |
| Changed files in your working directory | Add all current changes to the next commit  
$ git add . | Abort a rebase  
$ git rebase --abort |
| Changes to tracked files | Add some changes in `<file>` to the next commit  
$ git add -p `<file>` | Continue a rebase after resolving conflicts  
$ git rebase --continue |
| Add all current changes to the next commit  
$ git add . | Commit all local changes in tracked files  
$ git commit -a | Use your configured merge tool to solve conflicts  
$ git mergetool |
| LOCAL CHANGES | LOCAL CHANGES | LOCAL CHANGES |
| Commit previously staged changes  
$ git commit | Commit the last commit  
Don't amend published commit!  
$ git commit --amend | Use your editor to manually solve conflicts and (after resolving) mark file as resolved  
$ git add <resolved-file> |
| LOCAL CHANGES | LOCAL CHANGES | LOCAL CHANGES |
| Show all commits, starting with newest  
$ git log | Show changes over time for a specific file  
$ git log -p `<file>` | Revert a commit (by producing a new commit with contrary changes)  
$ git revert `<commit>` |
| Show changes over time for a specific file  
$ git log -p `<file>` | Who changed what and when in `<file>`  
$ git blame `<file>` | Reset your HEAD pointer to a previous commit  
...and discard all changes since then  
$ git reset --hard `<commit>  
...and preserve all changes as unstaged changes  
$ git reset `<commit>`  
...and preserve uncommitted local changes  
$ git reset --keep `<commit>` |

UPDATE & PUBLISH

- List all currently configured remotes  
$ git remote -v
- Show information about a remote  
$ git remote show `<remote>`
- Add new remote repository, named `<remote>`  
$ git remote add `<shortname>` `<.git>`
- Download all changes from `<remote>`, but don't integrate into HEAD  
$ git pull `<remote>`
- Download changes and directly merge/integrate into HEAD  
$ git pull `<remote>` `<branch>`
- Publish local changes on a remote  
$ git push `<remote>` `<branch>`
- Delete a branch on the remote  
$ git branch -D `<remote/branch>`
- Publish your tags  
$ git push --tags
- Use your configured pull tool to solve conflicts  
$ git pull
Step by Step Tutorial

Create a repo on Github/Bitbucket by group

Clone the repository

Create a new file for each person in the group. Example: name.txt

Add your new file: `git add name.txt`

Commit your new file: `git commit -m "Added my name file name.txt"`

Push: `git push remote origin`
Git pull

Open one of the files (same file for each person in the group)

Change the first line to create a conflict.

Experiment with branches: git checkout -b newBranch

Experiment with git’s log history: git checkout -hash commit-
Resources


http://blogs.atlassian.com/2012/02/version-control-centralized-dvcs/


http://stackoverflow.com/questions/1057564/pretty-git-branch-graphs

https://www.git-tower.com/blog/git-cheat-sheet/

http://gitref.org/