Using Git for Centralized and Distributed Version Control Workflows - Day 3

1 April, 2016
Presenter:
Brian Vanderwende
Git jargon from last time...

- **Commit** - a project snapshot in a repository
- **Staging area** - where additions/modifications are gathered to be packaged into a commit
- **Clone** - a copy of an existing repository
- **HEAD** - the most recent commit of the currently checked out branch
- **Rebasing** - moving the starting point of a branch from an older to a new commit in the parent branch
- **Remote** - an outside repository that is linked to the current repository (can be local or on a server)
- **Push/pull** - send/receive commits to/from a remote
Day 3 - Workflows, Web Servers, and Submodules

1. Git workflows
   a. Distributed and centralized
2. Using a web-based remote repository - Github
3. Git submodules
While Git can be used for local collaboration, it was designed for large *distributed workflows*

- In a distributed workflow, devs can clone an “official” public repository to create private development repos.
- The developer then pushes their changes to their own public repository, from which the official project maintainer can pull to the “official” repo.
- This is also called the *integrator* workflow, because the official maintainer integrates features from developers.
- Implicitly provides security and redundancy.
The maintainer is the development *integrator*
Centralized workflows, involving a central repository, are also possible with Git

- In this model, all developers have a private repository, from which they can push features to a central (typically networked) repository
- Similar to the Subversion workflow
- All developers are responsible for resolving commit conflicts with other pushed changes
  - There is no integrator in this model
Visualizing the centralized workflow

Central Server
Public Repository

➔ Push
➔ Pull/Fetch

Developer
Private Repository

Developer
Private Repository
Public repositories are typically created without a working directory for obvious (?) reasons

- Use the **bare** option to create a repository without a working directory:

  \[
  \textit{git init --bare <repo_name>.git}
  \]

- Note the **.git** at the end of the repository name. This accepted convention is used when no working directory exists.
Of course, hybrid workflows with distributed groups accessing a central server are possible too!
Github and competitors simply provide web-hosting and tools for Git repositories

- Provide an easy method for cross-network collaboration as well as source distribution
- Add visual flair to the Git experience (web GUI)
- Can be incorporated into any workflow:
  - In centralized, the public repository is stored on web server, and all developers are given push access
  - In distributed, developers can store public repositories online (including the integrator and the “official” repository)
You may recall, I hosted the sample repository for this workshop on Github. Let’s explore that process...
Earlier, I wanted to set up a public repository for my workshop roster website project on Github.
Next, I needed to *push* my private repository to the public Github repository

- Git allows for HTTPS or SSH authentication. I find SSH to be more reliable, but you do need to set up an SSH key.

```
git remote add origin
    git@github.com:vanderwb/roster_site.git

git push -u origin master
```
A quick note about Github permissions

- By default, public repositories are open access
- Anyone can view the repository (all users have read/pull access)
- Anyone can issue a pull-request to the repository
  - This enables and follows the integrator workflow
- The owner of the repository can add collaborators
  - These users have write access, meaning they can push commits to the repository
  - Collaborators enable the centralized workflow
What if I want to nest one Git repository within another?
The simplest approach is to nest the subproject within a directory, and add to .gitignore

- Easy to set up
- Repositories act independently
  - Must be pushed/pulled independently
- **Problem:** if another user clones the superproject, subprojects are not cloned with it
We need a way to link the subproject to the superproject without mixing their commits...
First, let’s think about the Subversion solution to this problem: *externals*

- Creating an external in SVN is (relatively) easy
  - Create a directory for the embedded project
  - Set the directory to be an external by associating the repository URL
  - Commit the external to the superproject
- Now, whenever you update the superproject, the external will be updated as well
- If the external is in the same repository, any changes to it will be included in the commit list
  - If not, you have to commit changes separately
In Git, *submodules* provide some of the advantages of externals, with a few important differences

- A submodule is a copy of a single commit from the subproject repository, kept in a subdirectory of the superproject repository
- The directory structure of the superproject and URL of the subproject repository are maintained when cloning
- **However**, unlike SVN:externals, submodules are locked to a single commit at any one time, and don’t automatically track the external project’s HEAD
How do you add a submodule to a repository?

- A submodule is basically a special remote embedded in the superproject repository. So we add it:

  ```
  git submodule add [-b <branch>] <URL>
  ```

- By default, the submodule directory will have the same name as the source repository.
- A new, version controlled, file called `.gitmodules` stores the mapping of the repository to the directory.
- You can manually track a submodule branch using `-b`
  - If not set, the submodule will default to the master branch.
Use recursive cloning when copying a repository with submodules

- To properly clone a repository with submodules, use:

  `git clone --recursive <source-URL> <dest-URL>`

- Otherwise, you will get an empty submodule folder. You can recover from that by running:

  `git submodule init`
  `git submodule update`
First way to update the submodule - pull the commit tracked by the superproject

- The superproject tracks a single commit for each subproject/submodule
- After pulling a superproject commit that points to a newer submodule commit, update the submodule contents using (i.e. load the commit):

  \texttt{git submodule update [--merge/--rebase]}

- If your submodule diverges from the updates, a merge or rebase will be required
First way to update the submodule - pull the commit tracked by the superproject

Initial state

Superproject dj194bs

Submodule aab42a3

Subrepo aab42a3

Pull new superproject

Superproject chw3l91

Submodule c273ab8

Subrepo aab42a3

Update submodule

Superproject chw3l91

Submodule c273ab8

Subrepo c273ab8
Second way to update the submodule - pull the latest commit from the subproject branch

- The submodule is itself a remote repository
- You can pull the latest changes from the targeted branch of the subproject by doing a remote update

```
git submodule update --remote
```

- The tracked branch can be changed as follows:

```
git config -f .gitmodules
    submodule.<path>.branch <branch>
```
Second way to update the submodule - pull the latest commit from the *subproject branch*

**Initial state**

- **Superproject** `dj194bs`
  - **Submodule** `aab42a3`
  - **Subrepo** `aab42a3`

**Update submodule**

- **Superproject** `dj194bs`
  - **Submodule** `aab42a3`
  - **Subrepo** `c273ab8`

**Commit to superproject**

- **Superproject** `chw3l91`
  - **Submodule** `c273ab8`
  - **Subrepo** `c273ab8`
Changes made in the submodule must be committed within the subproject AND superproject

- If you make modifications to the subproject, you must commit them, and then stage and commit the submodule itself within the superproject

```
subproject$ echo "TBD" > hello_gpu.f90
subproject$ git add hello_gpu.f90
subproject$ git commit -m "Added stub file for hello world GPU program"
subproject$ cd ..
superproject$ git diff
diff --git a/subproject b/subproject
index 208f4f8..3479225 160000
--- a/subproject
+++ b/subproject
@@ -1 +1 @@
-Subproject commit 208f4f884c99ff26f012dbe36b50e3a7411af9f8
+Subproject commit 3479225ec1636dae3f29902200a8980a589c270d
superproject$ git commit -am "Updated submodule"
superproject$ git diff
```
Submodule pushing can be done recursively from within the superproject

- Submodule changes **must** be pushed before superproject changes!
- If you want Git to simply check for submodule changes, and terminate the push if they are found:

  ```
git push --recurse-submodules=check
  ```

- If you want Git to first push submodule changes:

  ```
git push --recurse-submodules=on-demand
  ```
CAUTION: Git’s heavy focus on branches can cause problems when submodules are introduced...

```
$ git checkout -b add_sub
$ git submodule add ../subproject
$ git commit -am "Added submodule"
$ git checkout master
warning: unable to rmdir subproject: Directory not empty
Switched to branch 'master'
$ git status
On branch master
Untracked files:

    subproject/

$ rm -rf subproject
$ git status
On branch master
nothing to commit, working directory clean
$ git checkout add_sub
$ ls subproject/
$ git submodule update
Submodule path 'subproject': checked out
'208f4f884c99ff26f012dbe36b50e3a7411af9f8'
$ ls subproject/
hello_mpi.f90  hello_serial.f90
```
Making life easier when using submodules

- Git aliases come in handy as many submodule commands are long and cumbersome:

  ```
  git config alias.spush ‘push --recurse-submodules=on-demand’
  git config alias.supdate ‘submodule update --remote --merge’
  ```

- The `foreach` command can be used to send any command to all submodules. For example:

  ```
  git submodule foreach ‘git checkout -b <branch>’
  ```
Integrating a subversion repository into a Git project using submodules requires a Git clone

- If Git, SVN, and Alien::SVN are installed, can use git-svn to clone an SVN repository in Git. On Yellowstone:

  module load git
  module load git-svn
  git svn clone -s <SVN-URL> <clone-path>

- Then, in the Git superproject, make the clone a submodule using `git submodule add <clone-path>"
Updating the SVN submodule is a clunky process

- Need to **resync** with upstream SVN repo, **pull** changes to the submodule, and **update** the superproject:

  ```
  cd <clone_path>
  git svn rebase
  cd <superproject_path/subproject>
  git checkout master
  git pull
  cd ..
  git add <subproject>
  git commit -m "Updated submodule to vX.X"
  ```
Github can make the Git/SVN transition easier, as both programs can interact with Github repositories.
For more information, check out man pages and:

https://git-scm.com/doc
http://rypress.com/tutorials/git/index
http://nvie.com/posts/a-successful-git-branching-model/
https://www.atlassian.com/git/tutorials/

My contact information:

Brian Vanderwende
CISL Consulting Services Group
ML-55L (x2442)
vanderwb@ucar.edu