Version control

E6891 Lecture 4
2014-02-19

Today's plan

- History of version control
  - RCS, CVS, SVN, Git & friends
- Distributed version control
- Best practices for research
  - … aka, Brian's work flow?

What is version control?

- Tracking changes to your project
- Who changed what, when?
- Why do I need this?
  - Systematic journaling
  - Collaboration
  - Release management

Version control for research?

- Document your progress
- Project management
- Backups, and rollback mistakes
- Collaborative development, writing
- Versioning of software
  - and results!

Revision Control System (RCS)

[Tichy, 1982]

- Provides version control for a single file
  - changes tracked by unix `diff`

- Transaction-based:
  - check out/lock `file.ext`
  - edit `file.ext`
  - check in `file.ext`

Drawbacks of RCS

- Each file versioned independently
- No concept of user management
- Manual synchronization
  - via `rsync`
  - or working in the same directory

Concurrent Versions System (CVS)

[1986, 1990]

- **Multiple-file** versioning

- Transactional architecture
  - check out/lock the **repository**
  - edit files
  - check in/unlock

- Changes are only allowed to latest version

Drawbacks of CVS

- Changes can **only** be made against the head
  - In practice, only one person can modify at a time
- Networking is clumsy
- Commits are not atomic
- Poor support for binary files

Subversion (SVN)

[2000]

- Similar to CVS, but with many improvements

- **Centralized** client-server architecture
  - Allows for distributed development
  - … and direct sharing of code via public servers
  - (CVS did via `pserver`, but it was painful)

- Better support for binary files

Drawbacks of SVN

… or centralized VCS in general

- Versioning is done server-side
  - Incremental local development is tricky
  - Possible with branches, but merging is a headache
- Single point of failure
  - Rebuilding a repository from a checkout isn’t fun
- Distributed development from outsiders?

Git

[Torvalds, 2005]

- **Distributed** version control system (DVCS)

- Does not require a centralized server
  - but you can still have one, if you want

- Other DVCSs
  - Mercurial (`hg`)
  - Bazaar (`bzr`)

Client-server git usage

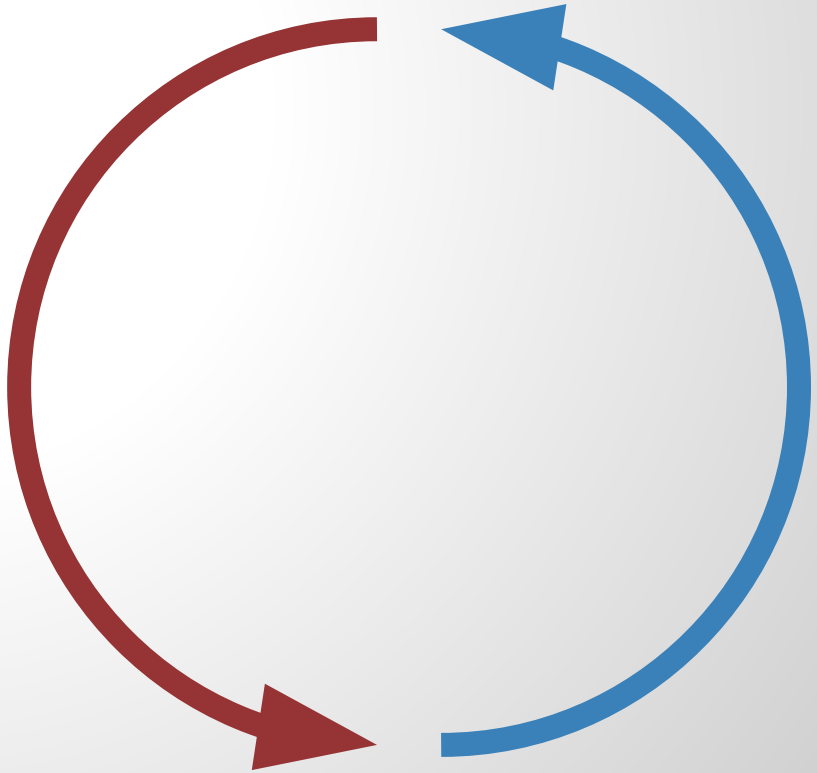
1. `git clone https://server/repository.git`
   Make a local copy of the repository

2. `(edit files)`

3. `git commit`
   Register your changes locally

4. `git push`
   Share changes upstream

5. `git pull`
   Get updates from upstream



Advanced usage: tags

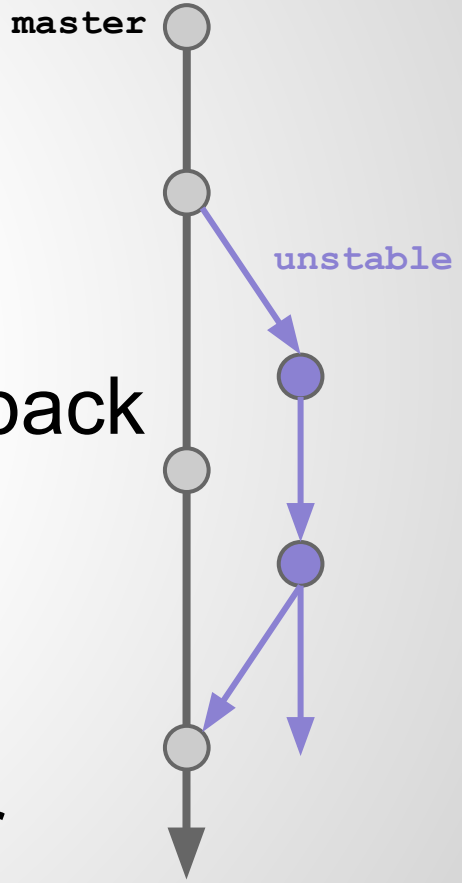
- Some revisions are special:
  - initial paper submission
  - camera-ready submission
  - public software releases (1.0, 1.1, …)
- **Tagging** links semantic versions to revisions
- Example:
  - `git tag -a v1.0`
  - `git push origin --tags`

Advanced usage: branches

- What if you want to develop new features, but retain version control on a stable codebase?
- Work in a **branch** of the source tree
- Merge back when you're ready
- Especially useful for collaborations

Branching

- Example: create a new branch
  - `git checkout -b unstable`
  - `(edits, commits, pushes)`
- Switch to master, bug fix, switch back
  - `git checkout master`
  - `(edits, commits, pushes)`
  - `git checkout unstable`
- Merge **unstable** back into master
  - `git checkout master`
  - `git merge unstable`



GitHub

[2008]

- Free hosting for open source projects
  - Free organization accounts for academics
- Social network integration
  - Surprisingly useful for research
- Extra usability tools:
  - user management
  - **pull requests**
  - issue tracking, comments, wiki
  - release management

My usual work flow

- Pull from github
  - Either `develop` or `master`, depending...
- Develop locally
  - first in ipython notebook
  - then in versioned source
  - **run unit tests**
  - commit
  - keep editing, pulling changes from collaborators
- When it's ready
  - push back to github

Research repositories

- When milestones happen, `tag`
  - Just after submitting the paper
  - When the final camera-ready goes out
  - Subsequent versions

- What's in a typical repository?
  - `README` Description and instructions
  - `code/` Source code
  - `data/` Sometimes: input data
  - `paper/` LaTeX source for the paper
  - `results/` Sometimes: output data, models

Some of my repositories

- LibROSA
  - https://github.com/bmcfee/librosa
  - Python module for audio processing research
- MLR
  - https://github.com/bmcfee/mlr
  - Matlab program for metric learning
  - (imported to git after development)
- Gordon
  - https://github.com/bmcfee/gordon
  - migrated from bitbucket to github

Best practices

- Use meaningful commit messages!
- BAD

```
git commit -a -m “foo”
```

- GOOD

```
git commit -a -m “changed
default lambda parameter to 1.0”
```

Best practices

- Commit often
  - push less often
- Use tags and milestones
- Use issue tracking