CSE 374
Programming Concepts & Tools

Brandon Myers

Winter 2015

Lecture 18 – Version control and git

Where we are

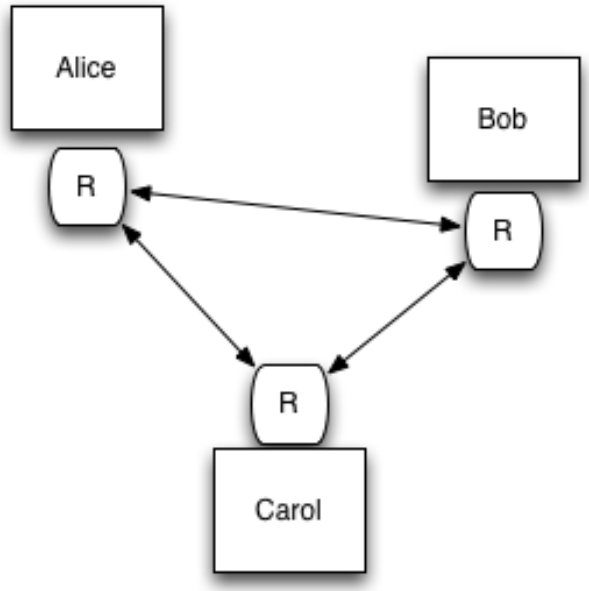
- Learning tools and concepts relevant to multi-file, multi-person, multi-platform, multi-month projects
- Today: Managing source code
  - Reliable backup of hard-to-replace information (i.e., sources)
  - Tools for managing concurrent and potentially conflicting changes from multiple people
  - Tools for managing multiple sets of changes to source code (features)
  - Ability to retrieve previous versions
- Like make, version-control systems are typically not language-specific.
  - Many people use version control systems for everything they do (code, papers, slides, letters, drawings, pictures, . . .)
    - Traditional systems were best at text files (comparing differences, etc.); newer ones work fine with others too

Version-control systems

- There are plenty: scss (historical), rcs (mostly historical), cvs (built on top of rcs), subversion, git (much more distributed), mercurial, sourcesafe, …
- The terminology and commands aren't particularly standard, but once you know one, the others aren't difficult – the basic concepts are the same
- svn was and still is widely used
  - centralized version control (all changes happen at the central server)
- git and mercurial, very popular today
  - distributed version control (every user has their own copy of the repository)

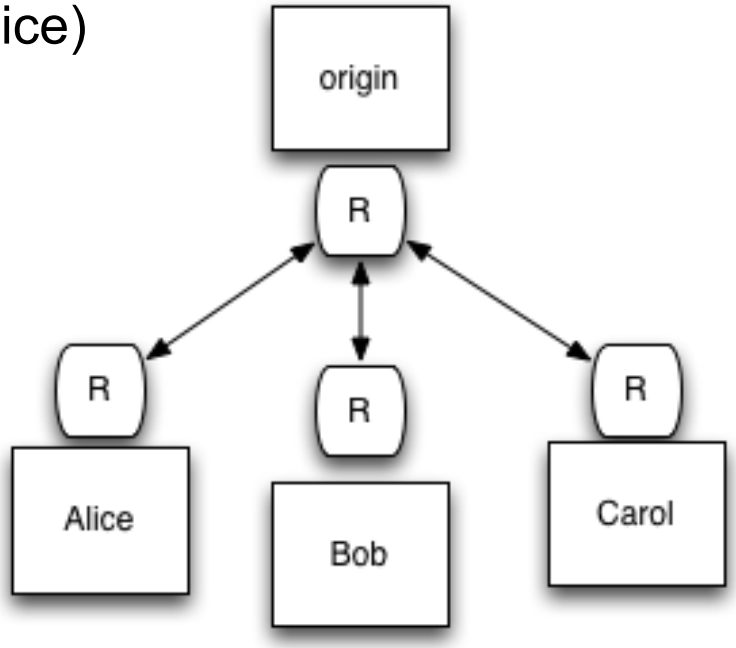
Git basics - Fully distributed

- A project lives in a repository
- Git is a distributed model. Every user has their own copy of the repository.
- Alice **commits** to her copy to “save” her changes
- Bob gets Alice’s changes by **pull**ing from Alice’s clone



Git basics - central copy

- **The model we will use in CSE374**: A very common model of using git is to have a shared repository called *origin*
- To begin, each user **clone**s origin's repository
- Alice **commit**s to her copy to "save" her changes
- Alice shares her changes by **push**ing to origin
- Bob gets Alice's changes by **pull**ing from origin (instead of directly from Alice)



Tasks

Learn the common cases; look up the uncommon ones.
In a production shop using git…

- Create
  - a new repository/project (infrequent - once or twice a year)
  - a new *branch* off your working copy (days to weeks)
    - think, one *feature* worth of changes, e.g. “created a test program for the trie”
  - a new commit (daily to multiple times a day)
    - think, one change, e.g. “Added a new test for word with a single # ”

- Working with files
  - Get changes, add or remove files, commit changes to your working copy
  - Check version history, differences
  - pushing changes from your working copy to origin

Repository access

A repository can be:

- Local: specify repository directory root via a regular file path name url (/path/...)
- Remote: lots of remote protocols supported (ssh, https, …) depending on repository configuration
  - Specify user-id and machine
  - Need git and ssh installed locally
  - Need authentication (ssh password or ssh key)
- HW6 uses ssh access to remote server (gitlab.cs.washington.edu)
- Feel free to experiment with private repos on gitlab, or local repos on your own computer

Getting started (gitlab)

- Set up a repository (we'll do this step for you on hw6; if you do it yourself you get to pick name, location
  - +New Project (on gitlab dashboard)
- Clone a working copy of the repository to your local machine
  - git clone git@gitlab.cs.washington.edu:path/to/project
  - the URL above comes from the gitlab page for your project

How to use git (edit a file)

```
# edit a file
vim shout.c

# add the change to the next commit
git add shout.c

# commit all added changes
git commit -m “changed shout message”

# At this point we have stored changes to
# our own copy, but we have not touched
origin
```

How to use git (share your changes)

```
# Suppose now I have done a few commits and
# I need to share my changes with my co-
workers.

# Others may have changed the origin copy
while I’ve been working
# So first, apply and changes in origin
# to my copy
git pull

# now share my changes
git push
```

Some examples

- **Update my local copy** to match origin copy
  git pull
- Make changes
  git add file.c
  git mv oldfile.c newfile.c
  git rm obsolete_file.c
- Commit changes **to my local copy**
  git commit -m “fixes bug #441”
- Examine your changes
  git status # see un-committed changed files
  git diff # see un-committed changes *in* files
  git log # see history of commits
- **Update the origin** copy to match my copy
  git push

Merging and Conflicts

- Everything works great if only Alice is making changes beyond the origin copy
- But when Alice and Bob make changes beyond the origin copy, the two versions must be **merged**
  - git will merge automatically when you do a “git pull”
  - Will usually be successful if Alice and Bob changed different lines or different files
- But sometimes git fails to automatically merge changes
  - you have to do the merge manually, but git helps!
  - 1. determine what file has the conflict
    - git status
  - 2. vim foo.c
    - you’ll see something like

```
<<<<<<<<<<<<HEAD
for (int i=0; i<10; i++)
==========
for (int i=1; i<=10; i++)
>>>>>>>>>>>> master
```

    - change these lines to be what you actually want
  - 3. add the file and commit the merge
    - git add foo.c
    - git commit

git gotchas

- Do not forget to add and commit files or your group members will be very unhappy.
  - you can check this with “git status”
- If your group members don’t see your commits, you need to share them
  - git pull; git push
- Keep in the repository *exactly* (and *only*) what you need to *build* the application!
  - Yes: foo.c foo.h Makefile
  - **No**: foo.o a.out

gitlab website

- https://gitlab.cs.washington.edu
- Files tab
  - examine all your code on the web interface
  - you can even edit files, but don't do this for HW6 (you must learn to use the command line)
- Commits tab
  - similar to "git log", it shows you the history
- Issues tab
  - create issues to track tasks
  - assign tasks to group members
  - We recommend using this!

The next step: branching

- The workflow shown previously is highly recommended for working on HW6. Don't use the following until you've mastered those commands.
- Git is really powerful with local *branches*
- Idea: use one local branch for each isolated feature you are working on

```
git checkout -b test-program
# create commits…

# push the new branch to origin repository
git push -u origin test-program
```

When ready to merge your feature, on gitlab, create a pull request, have a group member review and merge into master branch…

Summary

- Another tool for letting the computer do what it's good at:
  - Much better than manually emailing files, adding dates to filenames, etc.
  - Managing versions, storing the differences
  - Keeping source-code safe
  - Preventing concurrent access, detecting conflicts

- git: full documentation is online, free, downloadable
  also there is a book! https://progit.org/
  - Chapters 1 & 2 have most of what you'll need

Getting started (local repo)

- **We will use gitlab for HW6, but this slide is for reference for when not using gitlab for hosting your git repo**
- Set up a repository
  - git init --bare /path/to/myrepo
- Clone a working copy of the repository
  - git clone URL
  - URL for gitlab comes from the homepage of the project
  - URL if the repo is on your

- Check out a copy of the project to a *working directory*

  cd working-directory

  svn checkout svn://path/svnrepos/proj proj
  - Working directory remembers repository location and password for future checkin, update, etc.
- HW6: path to repository server is on cse server – see writeup