USPAS – *Simulation of Beam and Plasma Systems*

Steven M. Lund, Jean-Luc Vay, Remi Lehe, Daniel Winklehner and David L. Bruhwiler

Lecture: Software Version Control

Instructor: David L. Bruhwiler

Contributors: R. Nagler, P. Barbe and P. Moeller

radiasoft

U.S. Particle Accelerator School sponsored by **Old Dominion University**

http://uspas.fnal.gov/programs/2018/odu/courses/beam-plasma-systems.shtml

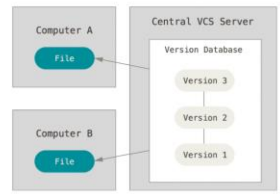
January 15-26, 2018 – Hampton, Virginia

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Offices of High Energy Physics and Basic Energy Sciences, under Award Number(s) DE-SC0011237 and DE-SC0011340.

U.S. DEPARTMENT OF ENERGY
Office of Science

Centralized version control systems (VCS)

- A version control system (VCS) records changes to a set of files
- Manual version control (ad hoc, error prone)
  - copy file versions with some convention for naming, location, etc.
  - ad hoc, error prone, difficult to collaborate
- Centralized software version control
  - enables collaboration
  - reliable recovery of previous states
  - CVS, Subversion (SVN), many others



- Criticisms of centralized systems
  - server is a single point of failure
  - if server goes down for an hour
    - nobody has access
  - if database becomes corrupted
    - all recent work is lost (since backup)
    - except for individual snapshots
  - all these criticisms are addressed by a well-managed system

Distributed vs Central models

- Centralized version control systems
  - focuses on synchronizing, tracking, and backing up files
  - recording/downloading is simultaneous with applying a change
  - primary repo is a database on a central server
    - the entire change history, including branches, is part of the central database
    - user repositories are snapshots that get synched with the central database
- Distributed version control systems
  - focuses on sharing changes; every change has a unique `guid`
  - recording/downloading is separate from applying a change
  - the hierarchical structure is not required
    - one can create a centrally administered location, if it is convenient
    - alternatively, one can treat ***all*** repositories as equal peers
    - this results in new concepts and associated terminology
      - **`push`**: send a change to another repository
      - **`pull`**: grab a change from a repository
  - the change history, including branches, are distributed
    - every user repo is self-contained

git – Getting Started

- It is assumed you are working on the Linux command line
- Establish your git identity (name & email) for the local client
  - every git commit uses this information
  - it's immutably baked into the commits you start creating

```
$ git config --global user.name "My Name"
$ git config --global user.email my_name@example.com
```

  - you need do this only once if you pass the --global option
  - many GUI tools will help you do this when you first run them
- Configure the default text editor

```
$ git config --global core.editor emacs
```

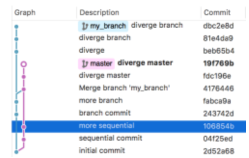
  - used when git needs you to type a message
  - if not configured, git uses your system's default editor

Class discussion:

- Any questions at this point?
- Any concerns about using git from the command line (CL)?
  - git is a distributed VCS implementation
  - the classroom computers provide git on Linux
  - 2 students per computer, but only one Linux login
    - **this means you'll have to share a single git identity**
- Work from your laptop…
  - if it has a good CL environment, with git installed
  - PyCharm supports interaction with git, GitHub and other VCS options
- You can download/install the GitHub desktop application
  - https://desktop.github.com
  - it installs git on your Windows or MacOS laptop
  - it provides an optional command-line terminal for using git
- Today's computer lab exercises will provide some practical experience

git – Underlying Concepts (Part 1)

- The git CLI is not intuitive, compared to central model applications (e.g. svn)
  - it helps if you understand the underlying concepts
- The git commit tree
  - information is representable as a graph
    - **each node results from an operation**
    - **database is immutable and append-only**
- an example git Tree (see figure)
  - each node is associated with…
    - **the developer's commit message**
    - **a unique hash** (`guid`)



- Git references
  - a reference (ref) is a human readable label, pointing to a commit hash
    - **branches, tags, remotes are all forms of refs**
    - **refs facilitate interaction with the commit tree**
  - refs do not hold the information in the git database
    - **all such info is held within the commit tree, which is immutable**
  - suppose the git repository is in a bad state, and we want to back track
    - **all previous states are still present inside the tree**
    - **we need only change the references to the desired commit address**
  - git provides a special reference named HEAD
    - **current address for the state that is checked out in the working directory**

git – Underlying Concepts (Part 2)

- The state of a git repository has three components
  - Working Directory
    - **result of cloning a git repository**
    - **a directory with everything contained within the git repository**
  - Staging Index
    - **an intermediate space to add changes from the working directory**
    - **(without adding them to the commit tree)**
  - Commit Tree
    - **changes in the staging index are (when ready) added to the commit tree**
    - **each change is given a hash address**
- Cloning a repository

  Docs for 'git clone', https://git-scm.com/docs/git-clone
  - Create a local copy
    - **this is complete and independent from the source**
  - git supports various protocols:

    `$ git clone [<options>] <repo> [<dir>]`
    - **If no [<dir>], git creates a new directory with the same name as the repo**
  - local filesystem clone

    `$ git clone /Path/To/Git/Repo/Dir`
  - remote HTTPS clone from GitHub

    `$ git clone https://github.com/radiasoft/devops.git`

git – the Checkout command

- It changes the `HEAD` reference, making it point to a new address
  - affects only the working directory
  - secondary use: undo changes in the working directory

    `$ git checkout [<options>] <branch>`

Docs for 'git checkout', https://git-scm.com/docs/git-checkout

- Useful examples:
  - get latest commit from the `master` branch for use in currently active branch

    `$ git checkout master`
  - get an address (e.g. `2d52a68`) and label it as branch `new_branch_name`

    `$ git checkout -b new_branch_name 2d52a68`
  - force a checkout from `master` branch, throwing away local modifications

    `$ git checkout -f master`
  - revert changes in file `my_file.py`

    `$ git checkout path/to/my_file.py`
  - revert file `my_file.py` to its state in the branch `my_branch`

    `$ git checkout my_branch -- path/to/my_file.py`

git – how to Stage and Commit

- Staging – add changes from the working directory to staging index
  - add new (untracked) file to staging index (or new changes to a tracked file)
    `$ git add path/to/file`
  - add all changes of tracked files to the staging index
    `$ git add -u`

Docs for 'git add', https://git-scm.com/docs/git-add

- Commit – store changes within the commit tree
  - changes may come from the staging index or directly from the working directory
  - each commit requires a message to document the changes being recorded
- Some examples:
  - commit the staging index, and document with a message
    - if don't specify an inline message, an editor will be invoked
    `$ git commit -m 'this is my commit message'`
  - commit all changes in tracked files
    `$ git add -a`
  - commit changes within a specific file
    `$ git commit /path/to/file -m 'file is better now'`

Docs for 'git commit', https://git-scm.com/docs/git-commit

git – Push & Pull

- Pull – performs a 'fetch' and 'merge' in one step
  - pull the remote tracking branch into the current working directory
    - if you clone a repo, it's 'master' is your 'remote tracking branch'
    - we do not discuss 'fetch' and 'merge' here
  `$ git pull`

Docs for 'git pull', https://git-scm.com/docs/git-pull

- Push – send changes from the local branch to a remote repo
  - push to the remote tracking branch
  `$ git push`

Docs for 'git push', https://git-scm.com/docs/git-push

- There are many sophisticated uses of `push` & `pull`
  - e.g. one can push to (or pull from) arbitrary branches in remote repos

git – Creating a Branch

- A branch tracks a set of (logically connected) changes
  - no conflicts with concurrent modifications to the same part of the repo
    - conflicts can manifest when merging two branches with overlapping changes
  - a branch is a ref
    - points to latest commit in corresponding 'branch' of the commit tree
- In our example repo (see figure on slide #6), we start with two branches
  - `my_branch` & `master`
  - both initially point to the same address, `2d52a68`
  - after changes in each branch occur separately, we see they have diverged
    - addresses `243742d` & `04d25ed` respectively.
- Examples of using the branch command:
  - Create new branch `branch_name` pointing to same address as `HEAD`
    `$ git branch branch_name`
  - List local branches
    `$ git branch`
  - Delete branch named 'branch_name'
    `$ git branch -d branch_name`
  - Rename the branch `branch_name` to new name: `new_branch_name`
    `$ git branch -m branch_name new_branch_name`

Docs for 'git branch', https://git-scm.com/docs/git-branch

git workflow – create, then merge a branch

- Create a new branch, named 'issue03'
  - perhaps the goal is to address issue #3 from GitHub repo
  `$ git checkout -b issue03`
  - the above is shorthand for the following two commands:
  `$ git branch issue03`
  `$ git checkout issue03`
- Add a new file to the branch (trivial example)
  `$ touch dummy.txt`
  `$ git add dummy.txt`
  `$ git commit -m 'this file is empty'`
  `$ git push`
- Merge this branch into the 'master' branch
  `$ git checkout master`
  `$ git merge issue03`

Docs for 'git merge', https://git-scm.com/docs/git-merge

More workflow details here, https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

Class discussion:

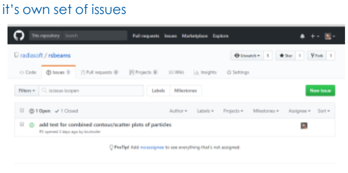
- Any questions at this point?
- Why would you want to create a branch?
- What is a ‘ref’ in the world of git?
- Today’s computer lab exercises will provide some practical experience

GitHub overview

- GitHub & Bitbucket are two of the largest web-based hosting services
  for a comparison, see https://www.upguard.com/articles/github-vs-bitbucket
  - they are targeted towards software development projects
    - can be used for proposals, papers or any collection of documents
  - neither supports Subversion (SVN)
    - GitHub exclusively supports `git`; Bitbucket supports `git` and `mercurial`
- GitHub provides the following features (and more):
  - an integrated issue tracker
  - branch comparison views
  - native applications for Windows and Mac desktops
    - https://desktop.github.com/
  - support for over 200 programming languages and data formats
  - GitHub pages, a feature for publishing and hosting
  - SSL, SSH & https for data transmission; two-factor authentication for login
  - API integration for 3rd-party tool and other platforms
  - partial support is provided for `SVN`
    - import SVN repos into git
    - GitHub repos can be cloned directly via the SVN client.

The GitHub ‘issues’ feature

- Creating issues is a good thing
  - most other tracking systems call them ‘tickets’
  - every GitHub repo has it’s own set of issues



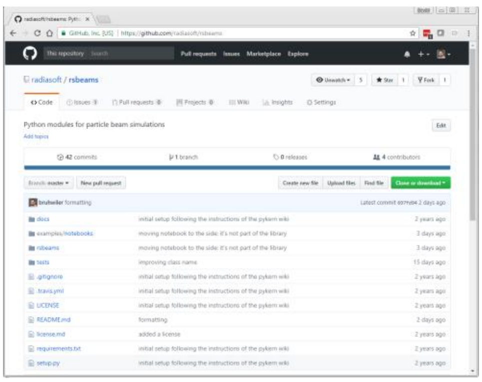
- Issues help you (or a team) keep track of
  - tasks, enhancements and bugs
- They are a very good alternative to email
  - they can be shared and discussed with the team
  - individuals can turn notifications on/off
  - they can be closed and later re-opened
  - provides a searchable archive

Docs for GitHub issues,
https://guides.github.com/features/issues/

An example GitHub code repository

- `rsbeams` is a python library for 3D particle beams
  `rsbeams:` https://github.com/radiasoft/rsbeams
  - not specific to any particular tracking code
- `rsbeams` is used by other Python libraries, which ***are*** code specific
  `rswarp:` https://github.com/radiasoft/rswarp
  `rssynergia:` https://github.com/radiasoft/rssynergia
- In the Computer Lab this afternoon & tomorrow, you will
  - fork this repo to your own GitHub account
  - clone this forked repo to your laptop or desktop
  - decide what part of the code you would like to test
  - create an ‘issue’ in the original repo regarding your plan to create a test
  - create a branch in your working directory
  - create/add/commit the test in your branch
  - merge your branch into the ‘master’ branch of your forked repo on GitHub
  - Issue a ‘pull request’ to the original repository
- We won’t cover all this material today

An overview of the `rsbeams` *repository*



Wrap up

- Any final questions regarding the material in this lecture?
- In the Computer Lab this afternoon, you will
  - fork this repo to your own GitHub account
  - clone this forked repo to your laptop or desktop
  - document each of the following with an issue:
    - **run the existing tests**
    - **create a branch**
      - create a new example, based on one of the existing tests
      - merge the branch back into 'master'
  - decide what part of the code you would like to test
    - **create an 'issue' in the original repo regarding your plan to create a test**