Best Practices for HPC Software Developers Webinar Series

Session 3: Distributed Version Control and Continuous Integration Testing

Welcome! We will begin soon

- **Make sure you get counted. Please visit http://bit.ly/hpcbp-s03**
- We want this webinar to be interactive, and **we encourage questions**
  - But we need to keep everyone's mic muted (too many participants)
  - **Please use the Zoom Q&A tool to submit questions**
  - **Or use type them into this Google Doc: http://bit.ly/hpcbp-qa**
  - Use the Zoom Chat tool for other issues
- **Slides and a recording will be available** from the OLCF training web site: https://www.olcf.ornl.gov/training-event/webinar-series-best-practices-for-hpc-software-developers
- We want to improve this series. **Please send feedback to *HPCBestPractices+session03@gmail.com***

Overview

- What is version control (and why do we care)?
- Centralized vs distributed version control
- Git: motivation, basic concepts, usage, learning resources
- GitHub as a collaborative development platform
- Tracking progress and prioritizing issues
- Pull requests as a mechanism for code changes
- Continuous integration

I am *not* going to

- teach you everything about Git(!)
- give you a translation chart from SVN to Git
- tell you how you should be running your projects

I will

- attempt to convince you that you need to use version control for your software projects
- give you some resources to learn more
- show you some examples of successful software development strategies
- hopefully show you something you haven’t seen before

This webinar covers a *lot* of ground.

- These topics belong to an area (software engineering practices) that is not part of the formal training of most “computational scientists.”
- I don’t know what you know, so I’m just trying to make sure you’ve seen these ideas.
- Don’t worry about absorbing all of this at once.
- Sorry if some of this is old news for some of you.

Version control is an **essential** component in software development.

- Also called “source code control,” “revision control,” “source code versioning”

- Has been used by software developers for decades

- Source code lives in one or more *repositories* (repos) available to team members/contributors.

- Developers make changes, incorporate changes from collaborators, merge changes into the “master” version of code in the repository.

- A repo is a computational scientist’s laboratory notebook.

Version control is an **essential** component in software development.

- Establishes a common context for code contributions and the exchange of ideas

- Establishes a chronological sequence of events

- Serves as “ground truth” for a software project

- If you don’t have a common reference for your source code, *there is nothing to for your team to discuss*.

- Results from uncontrolled code are *not reproducible*.

Sharing code with tarballs / file sharing is a recipe for disaster

- Recall your most frustrating document-sharing experience…

- … and imagine it continuing for months or years, with a changing cast of characters, with an ever-expanding set of documents.

- (It doesn’t work.)

Sharing code with version control is easy

- A repo can tell you exactly what version you are looking at (with a unique identifier)…

- … and identify any local changes you have made…

- … so that everyone can agree on whether they are looking at the same thing.

- If there are conflicts, your version control system will tell you, and *you will need to resolve them*.

“What if I’m developing software by myself?”

- Version control offers you the same advantages/legitimacy of a laboratory notebook

- If you’re developing your software on more than one machine, you still need to keep it consistent across these machines.

- If you want to collaborate with someone, congratulations! You’re now in the same boat as software teams.

The simplest version control systems are *centralized.*

- There is one repository containing the master version (the “trunk”) of the source code.

- Everyone syncs with this repository, *checks out* files, *changes* them, and *commits* these changes.

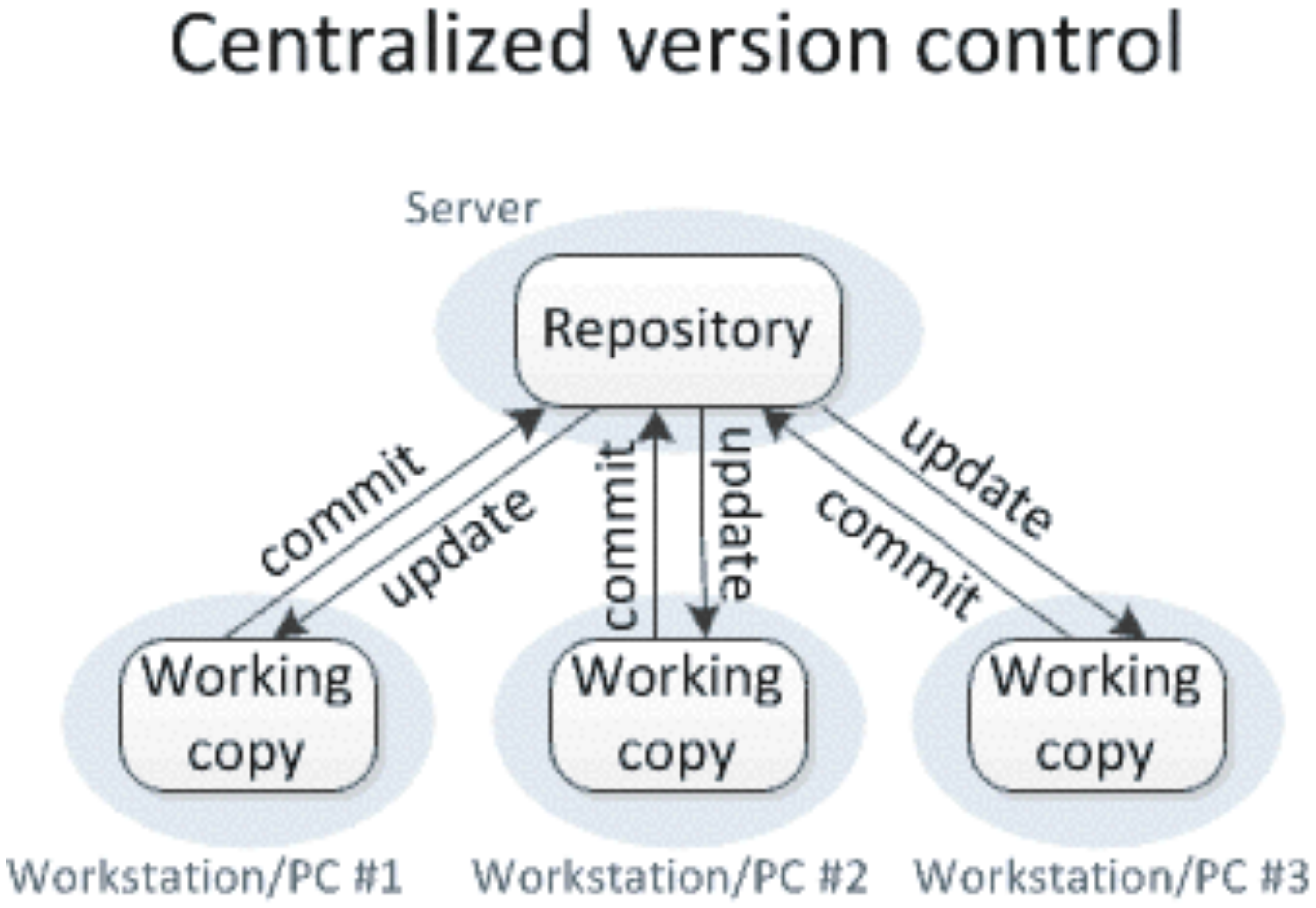
- People must cooperate to make sure their changes don’t conflict with each other.

- Simple, but limited.

  - Most centralized systems (SCCS, RCS, CVS, SVN) don’t allow the creation of separate development branches (though some fake it)

  - Requires coordination to keep people from stepping on each other’s toes.

Centralized version control

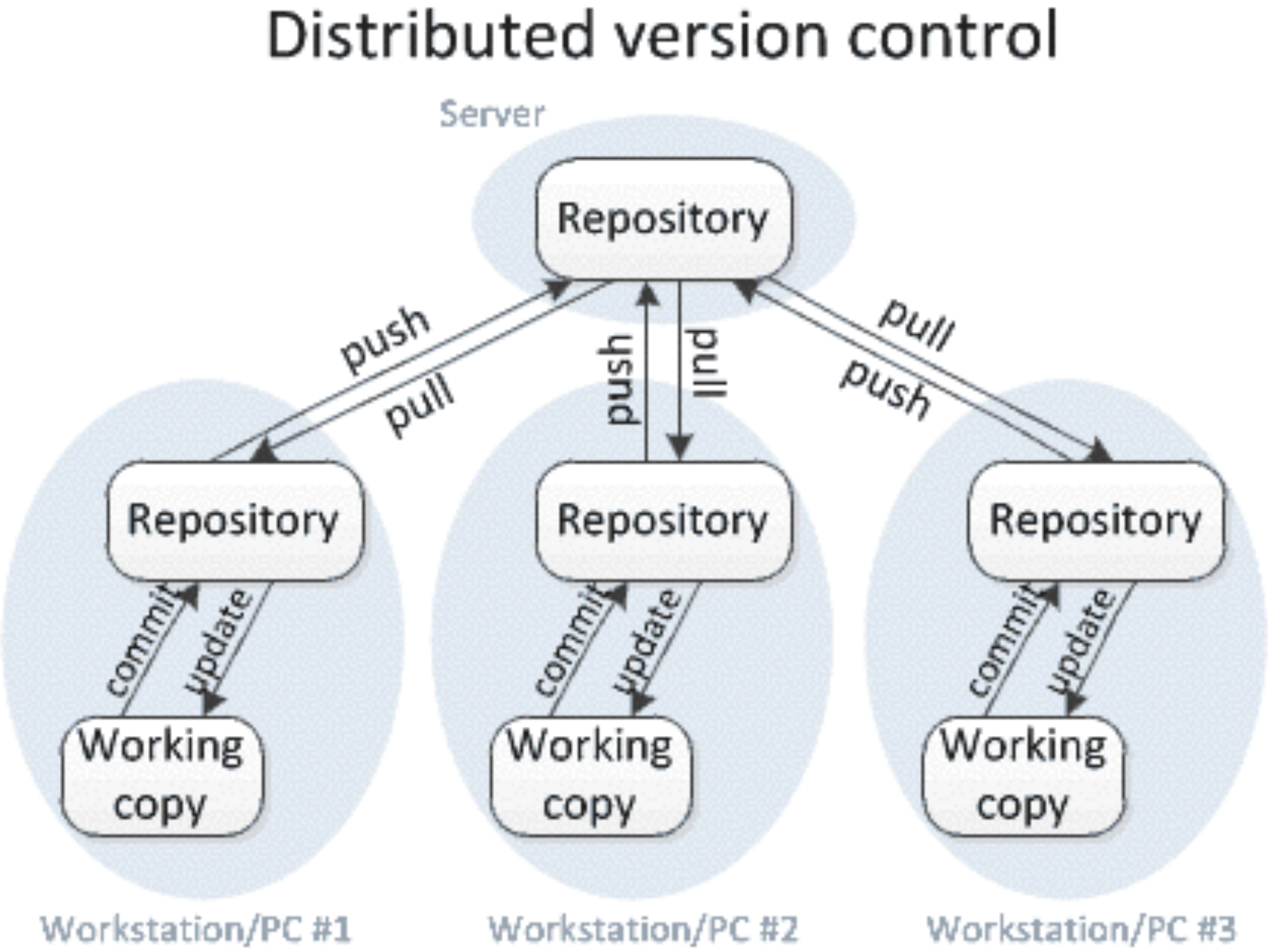


(Courtesy Michael Ernst,
University of WA)

More recently, distributed version control systems (DVCS) have emerged.

- Everyone has a copy of the entire repo and its history(!)
- There is a “main” repo, agreed upon by convention.
- People typically work in development *branches*, with their changes isolated from others until merges are performed.
- Greater flexibility for design development procedures.
- Greater complexity (more concepts, fewer set rules).

Distributed version control



(Courtesy Michael Ernst,
University of WA)

Git and Mercurial are the most popular DCVS tools.

- Git was written by Linus Torvalds, who *hated* Subversion, and has an interface that is alien to SVN users.
- Mercurial caters to Subversion veterans, with similar command syntax.
- Both support similar features.
- Git focuses on power, flexibility, and correctness, while Mercurial favors ease of use.
- More teams are using Git than Mercurial these days.

A version control tool is *just a tool*.

- It will not allow you to write code without communicating with others (including Future You).

- It does not define a process for developing software by yourself or on a team.

- You/your team should choose an approach based on the needs of your project and staff, and a tool that will support this approach.

- DVCS are interesting because they accommodate a wider range of approaches to software development. Even so, some still prefer centralized version control.

Software teams need to think about their process.

- Team software development is hard (because collaborative work in general is hard).
- Different teams have different needs.
    - What should be easy (happens often)?
    - What can be more complicated (happens rarely)?
- Designing a development process takes time, but pays for itself over time.

Let’s talk about Git!

- Git is difficult to learn without putting in some time.
  - The command syntax is pretty confusing.
  - Git evangelists sometimes talk about “the DAG” as if everyone knows what one of those is.
  - It’s difficult to understand how Git works without knowing the underlying concepts.
- Teams that use Git well often have one or more “Git people” that help the others.

Git can definitely help you do what you want to do, *and it works.*

- It’s usually easy to fix mistakes if you find them early.
- Operations are not left in an intermediate state unless they can’t be finished.
- It can support *arbitrarily elaborate* workflows.
- It doesn’t get in your way once you know what you’re doing.
- Perversely, it’s easier to learn Git (and DVCS in general) if you haven’t used SVN/CVS.

Git nouns and verbs

- *Repos*
- *Clones*/*cloning* of repos (making a copy of a repo)
- *Commits*/*committing* within repos (making code changes)
- *Branches*/*branching* within repos (isolated development)
- *Remotes*: references to other repos
- *Pulls*/*pulling* changes from one repo to another
- *Pushes*/*pushing* changes from one repo to another
- *Revisions* <—> commits (*hashes*)
- Workspace (*index*)
- *History* / the *graph* / the “DAG”

Git mechanics: creating a new repo

```
% mkdir example
% cd example
% echo “This is file A” > A
% echo “This is file B” > B
% ls
% git init
% git status
```

Git mechanics: adding files

```
% git add A
% git status
% git add B
% git status
% git commit -am “First commit.”
% git status
```

Git mechanics: changing files

```
% echo “ xtra stuff” >> A
% git status
% git diff
% git checkout A
% git status
% echo “ Xtra stuff” >> A
% git commit -am “Xtra A stuff”
```

Git mechanics: using the log/history

```
% git log
% git log —graph
% git show HEAD
% git show HEAD~1
% git reset —hard HEAD~1
% git log
```

(Long dashes are double dashes)

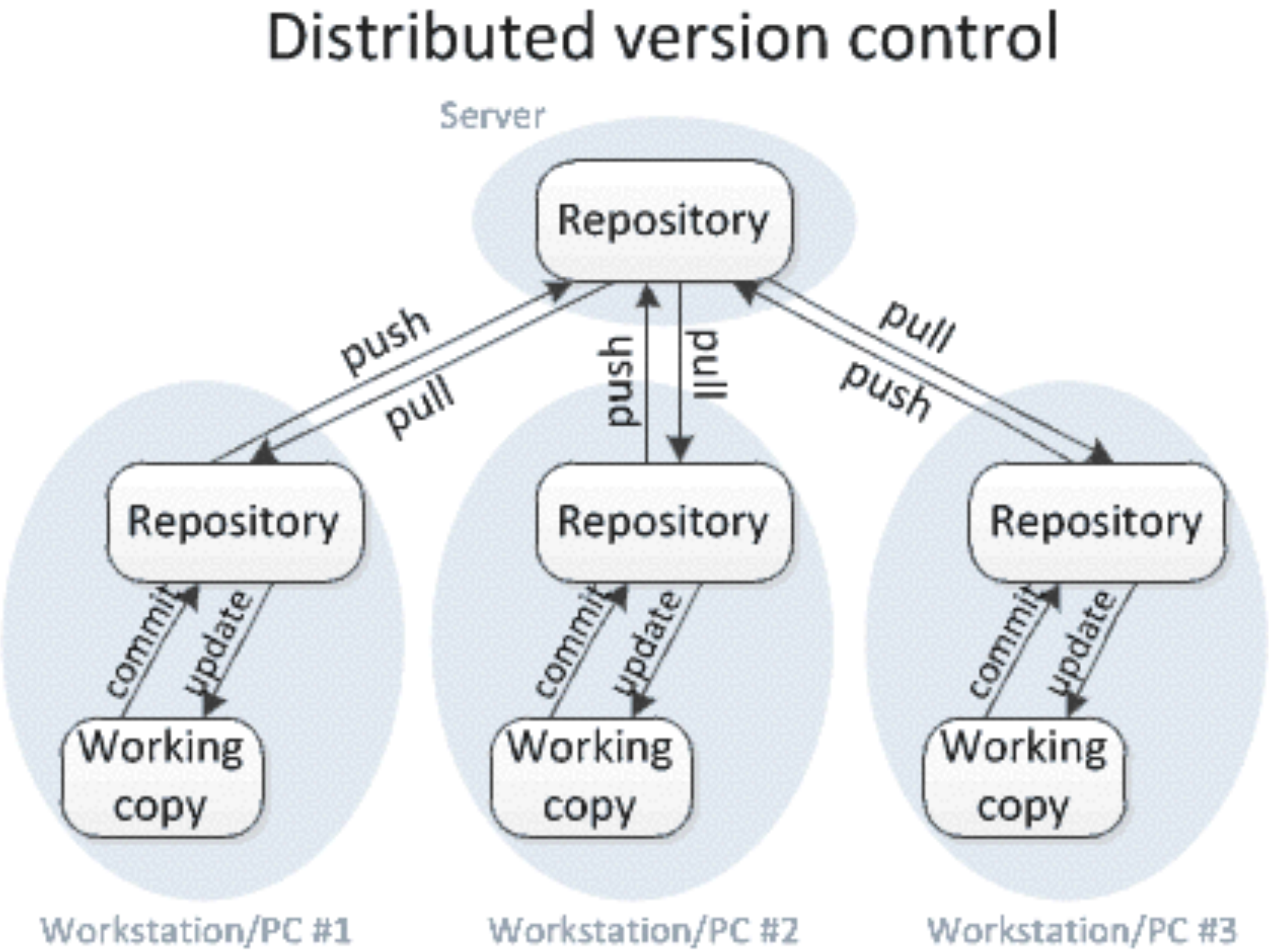
Git mechanics: creating a new branch

```
% git branch newA
% git status
% git branch
% git checkout newA
% git status
% echo “New A stuff” >> A
% git diff
% git commit -am “New A stuff”
% git log
```

Git mechanics:
merging the branch

```
% git checkout master
% git status
% git log
% git merge newA
% git log
% git branch -d newA
```

Git mechanics: remotes



Git mechanics: remotes

```
% git remote -v
% git remote add upstream http://
www.example.com/example-us.git
% git remote add downstream http://
www.example.com/example-ds.git
% git remote -v
% git pull upstream master
% git push downstream master
```

It’s a good time to learn Git.

- Tutorials
  - Getting Git Right:
    https://www.atlassian.com/git/
  - Interactive play space from Code School:
    https://try.github.io/levels/1/challenges/1
  - Exploring Git’s branching model:
    http://learngitbranching.js.org/
  - Video course:
    https://www.codeschool.com/courses/git-real
  - Git: The Simple Guide:
    http://rogerdudler.github.io/git-guide/

It’s a good time to learn Git.

- References
  - “The Git book”
    https://git-scm.com/book/en/v2
  - Learning Version Control With Git
    https://www.git-tower.com/learn/git/ebook/en/command-line/introduction
  - “The Git Reference”
    http://gitref.org
  - Git Magic
    http://www-cs-students.stanford.edu/~blynn/gitmagic/ch01.html
  - Git Cheatsheet
    http://www.ndpsoftware.com/git-cheatsheet.html

It’s a good time to learn Git.

- Tools
  - Tower (Mac OS X)
    https://www.git-tower.com/
  - Tortoise Git (Windows)
    https://tortoisegit.org/
  - Editor / IDE integration
    - Magit (emacs)
    - Fugitive (vim)
    - (your favorite IDE here)

It’s a good time to learn Git.

- Check out offerings in your local community!
  - Git/software engineering “bootcamps,” often cheap or free
  - Software workshops / conferences
  - Your CS/IS department would probably love to tell you more about this stuff
  - *You don’t need to learn it all by yourself!*

If you decide to use Git, check out GitHub.

- http://www.github.com
- Free repositories for Open Source projects
- Implements several helpful process “building blocks” (in easy-to-use forms)
  - Pull requests
  - Forks
- Includes some simple niceties (issue tracker, wiki, pretty log/graph visualizations)

If you decide to use Git, check out GitHub.

- Integrates with several interesting services
  - JIRA / Confluence (project management tools)
  - Slack / HipChat (team communication tools)
  - Travis CI (continuous integration)
  - many others
- Other similar services exist (Bitbucket, GitLab, …)
  - Mostly differ in how payment plans are organized, service integrations offered

GitHub provides some useful items for collaborative development.

- Issue tracking: a database for bugs and feature requests
- Fork: a clone of a repository, to be used for a specific purpose (e.g., by a single developer, or to create an alternative implementation of a piece of software)
- Pull request: a formal mechanism for reviewing a set of changes to be merged into the master branch

Sample GitHub project

https://github.com/jnjohnsonlbl/example

```
% git clone https://github.com/jnjohnsonlbl/example.git
```

Issue tracker

https://github.com/jnjohnsonlbl/example

A *fork* is just a clone of a repo with its own identity on GitHub.

- Useful if you are doing work that requires more isolation, or if you have your own process and don’t want to inflict it upon others.

- Can be used to submit changes/fixes to repos for which you don’t have direct write access.

- Use with caution if your team isn’t using forks as part of their process.

Fork the example repo

- Create a GitHub account
- Log into your account
- Navigate to https://github.com/jnjohnsonlbl/example
- Press the Fork button on the upper right
- In practice, there’s more to setting up a fork, but this illustrates the basic mechanism

A *pull request* formalizes the process of incorporating changes to software.

- A developer does some work in a branch, which exists in several commits on that branch.

- The developer wants to merge those commits into the master branch.

- He or she creates a *pull request*, with a description of the changes, helpful tags (“bug”, “testing”, “enhancement”, “data”).

- Colleagues can be notified of the request and asked to review changes using GitHub’s “diff” views.

A *pull request* formalizes the process of incorporating changes to software.

- *One or more automated events can be triggered by a pull request!*

- A reviewer may ask for changes to be made before the merge proceeds

- If/when reviewers are satisfied with the changes, the developer (or someone else assigned to merge the changes) can perform the merge, which closes the pull request.

Submit a pull request to the example repo

- Clone your fork of the example repo (to your workstation/ laptop).

- Modify a file within your workspace and commit the change.

- Push your change to your fork:

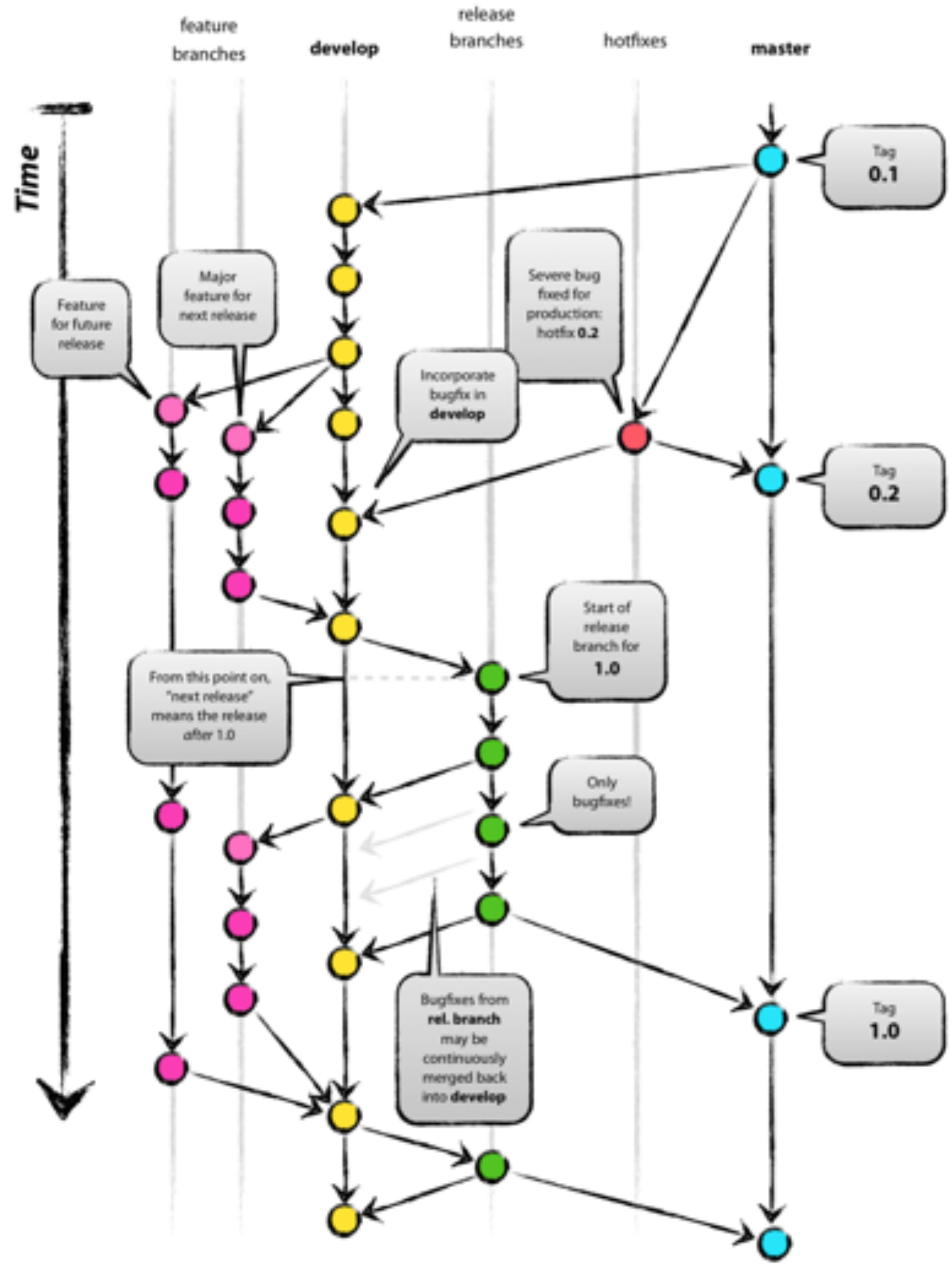
```
% git push origin master
```

- Navigate to your fork's GitHub page: https://github.com/yourname/example

- Click the "Pull request" button to the right of "this branch is 1 commit ahead of jnjohnsonlbl:master."

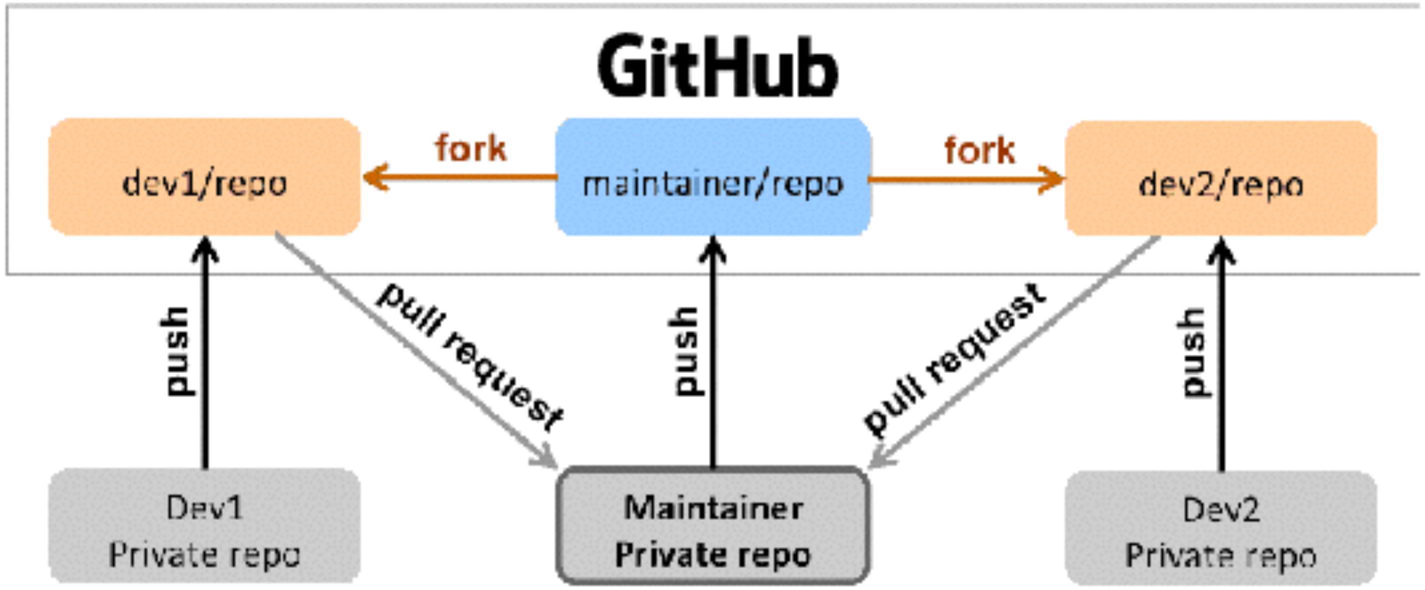
GitHub’s popularity has spawned some interesting development processes

- “Git flow” model and variants


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Major feature for next release
Feature for future release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

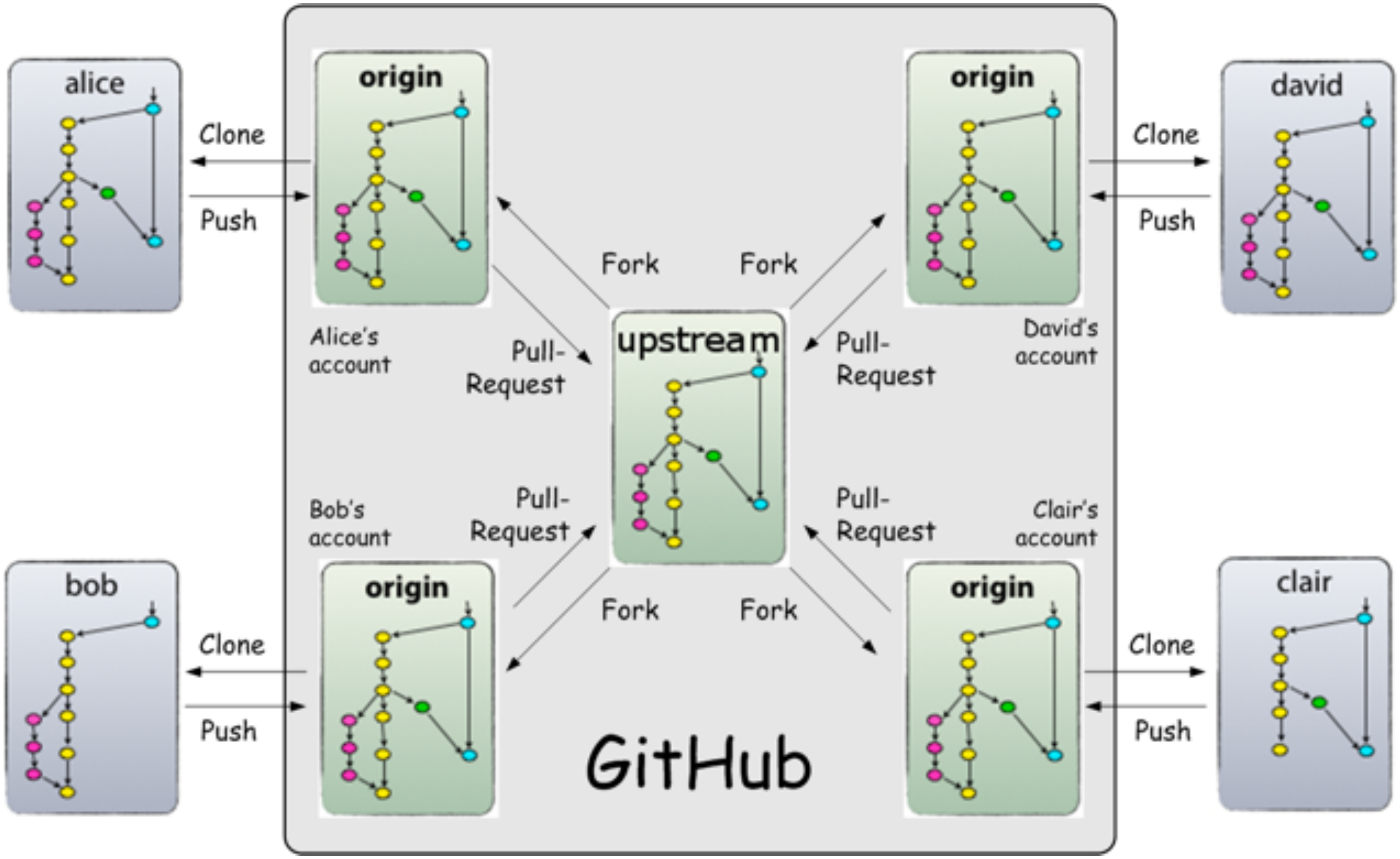
GitHub’s popularity has spawned some interesting development processes

- Forking workflow



GitHub's popularity has spawned some interesting development processes

- Mix-n-match (?!)



These processes have been studied by lots of people, and analysis is ongoing…

- Gitflow:
  http://nvie.com/posts/a-successful-git-branching-model/
  http://danielkummer.github.io/git-flow-cheatsheet/

- Fork-and-branch workflow:
  http://blog.scottlowe.org/2015/01/27/using-fork-branch-git-workflow/

- Comparison of Git workflows:
  https://www.atlassian.com/git/tutorials/comparing-workflows/

- Gitflow considered harmful(!):
  http://endoflineblog.com/gitflow-considered-harmful

Continuous integration (CI): a master branch that always works

- Code changes trigger automated builds/tests on target platforms.
- Builds/tests finish *in a reasonable amount of time*, providing useful feedback when it's most needed.
- Immensely helpful!
- Requires some work, though:
  - A reasonably automated build system
  - An automated test system with significant test coverage
  - A set of systems on which tests will be run, and a controller.

Continuous integration (CI): a master branch that always works

- Has existed for some time
- Adoption has been slow
  - Setting up and maintaining CI systems is difficult, labor-intensive (typically requires a dedicated staff member)
  - *You have to be doing a lot of things right to even consider CI*

Cloud-based CI is available as a service on GitHub.

- Automated builds/tests can be triggered via pull requests.
- Builds/tests run on cloud systems — no server in your closet. *Great use of the cloud!*
- Test results are reported on the pull request page (with links to detailed logs).
- Already being used successfully by scientific computing projects, with noticeable benefits to productivity.
- Not perfect, but *far* better than not doing CI.

Travis CI is a great choice for HPC

- Integrates easily with GitHub
- *Free* for Open Source projects
- Supports environments with C/C++/Fortran compilers (GNU, Clang, Intel[?])
- Linux, Mac platforms available
- *Relatively* simple, *reasonably* flexible configuration file
    - Documentation is sparse, but we now have working examples.

Travis CI

https://github.com/LBL-EESA/alquimia-dev

Wrap-up

- Your software projects need version control **(not debatable among professionals, for reasons discussed)**.
- Distributed Version Control Systems (DVCS) are becoming more popular, because they allow greater flexibility.
- Git seems to be the tool of choice in industry.
  - You don’t need anything more powerful.
  - Lots of documentation, knowledge and experience to draw from.
  - Learning it is an investment, but the payoff is real (but you might want to train up a “Git person”).

Wrap-up

- GitHub and similar sites provide capable, cost-effective development platforms.

- These sites offer useful services that can simplify common processes and improve your engineering practices.

- There are a number of well-described and well-studied software development processes that you can choose from, that incorporate Git and GitHub.

- Continuous Integration (CI) is a very effective practice that improves code quality, and is now within the reach of small teams.

Thanks for Participating!

- **Make sure you get counted. Please visit http://bit.ly/hpcbp-s03**
- **We want to improve this series. Please send feedback to *HPCBestPractices+session03@gmail.com***
- Slides and a recording will be available from the OLCF training web site: https://www.olcf.ornl.gov/training-event/webinar-series-best-practices-for-hpc-software-developers

Next Webinar

Session 4: Testing your Code/Documenting your Code
Date: Wednesday, June 15, 2016
Time: 1:00-2:00 pm ET
Presenter: Alicia Klinvex, Sandia National Laboratories

For updates, please register (if you haven't already)
https://www.olcf.ornl.gov/training-event/webinar-series-best-practices-for-hpc-software-developers