Reproducible Research, Open Science
Logging and backing up your work
Git Tips and Tricks, a Scientist Perspective

V. Danjean, A. Legrand, L. Stanisic
University of Grenoble, CNRS, Inria Bordeaux

June 7, 2016 – Reproducible Research Webinar (Episode IV)
Foreword about the organization (1/2)

There is currently a screencast of this seminar:

https://mi2s.imag.fr/pm/direct

The resulting video will be edited and available from GitHub that gathers all the information, slides, and resources:

https://github.com/alegrand/RR_webinars/blob/master/README.org

There is a few seconds delay between what we say and the screencast. We can have almost live interaction with other sites by using pad to comment and ask questions

http://tinyurl.com/RRW-pad4
Foreword about the organization (2/2)

No particular prerequisites: we will use command line and demo with a GUI

- Please install Git and SmartGit by following the instructions given on the RR_webinars GitHub page

1. General introduction plus basic Git usage (≈ an hour) At any time, feel free to ask questions on the pad. Some of these questions may be addressed "silently" by the other attendees, but do not hesitate to ask it out loudly for the remote ones

2. A short break

3. Slightly more advanced Git usage The actual part that is more devoted to practices favoring reproducible research

Many other tutorials on Git but with a different perspective

- Introduction to Git internals by Scott Chacon (GitHub)
- Please. Stop Using Git by Matthew McCullough 😊
- http://try.github.com/
- http://git-scm.com/docs/gittutorial/
Outline

1 Introduction to Git and its Eco-system
2 Git for Writing Articles
3 Git for Coding
4 Git for Experiments/Data/Provenance Tracking
5 Git for Publishing
6 Conclusion and Perspective
Challenges: History and Parallelism

- **Printed doc+handwritten revisions:**
  - Many versions of the file
  - One writer, multiple reviewers

- **Doc+email+contributions:**
  - Many versions of the file
  - Never sure which one is the latest
  - Not aware of others corrections
  - No clear history
  - Merging contributions
  - Problems editing same paragraphs

- **Doc+Dropbox or google-doc:**
  - Limited history through autosaves (30 days on Dropbox)
  - Limited parallelism
  - Coworker deletes everything...

You need a **real version control system**
that allows for a truly **parallel asynchronous edition**
Remember "The Dog Ate my Homework!!"


- 8 ACM conferences (ASPLOS’12, CCS’12, OOPSLA’12, OSDI’12, PLDI’12, SIGMOD’12, SOSP’11, VLDB’12) and 5 journals
- More than 80% of non reproducible work

Common issues: Versioning Problems and Bad Backup Practices

*Thanks for your interest in the implementation of our paper. The good news is that I was able to find some code. I am just hoping that it is a stable working version of the code, and matches the implementation we finally used for the paper. Unfortunately, I have lost some data when my laptop was stolen last year. The bad news is that the code is not commented and/or clean.*

- 8 ACM conferences (ASPLOS’12, CCS’12, OOPSLA’12, OSDI’12, PLDI’12, SIGMOD’12, SOSP’11, VLDB’12) and 5 journals
- More than 80% of non reproducible work

Common issues: Versioning Problems and Bad Backup Practices

*Attached is the ⟨system⟩ source code of our algorithm. I’m not very sure whether it is the final version of the code used in our paper, but it should be at least 99% close. Hope it will help.*

- 8 ACM conferences (ASPLOS’12, CCS’12, OOPSLA’12, OSDI’12, PLDI’12, SIGMOD’12, SOSP’11, VLDB’12) and 5 journals
- More than 80% of non reproducible work

Common issues: Versioning Problems and Bad Backup Practices

Unfortunately, the server in which my implementation was stored had a disk crash in April and three disks crashed simultaneously. While the help desk made significant effort to save the data, my entire implementation for this paper was not found.
History: Centralized Version Control System

- Client/Server interactions
- Local copies of files/remote history

Main issues:

1. **No local history**
   - requires a permanent network connection
   - commits are immediately visible to everyone
   - few (and large) commits, conflicts 😞

2. **Managing Access (Read/Write)**
   - **UNIX** groups and **ssh**. **Forge** makes your life so much easier
     - Web management and easy access to files and history
     - Bug Tracking System, Mailing Lists, Wikis, ...
   - This does not scale (socially)
     - send series of patches 😞
History: Decentralized Version Control System

- Peer-to-peer and/or Client/Server
- Full local history

Git birth:
- BitKeeper was used by the Linux kernel dev. community
- BK. withdrawn gratis use of the product for reverse-engineering reasons
- Torvalds needed performance. He coded Git in 2 weeks and used it for releasing the next kernel 2 weeks after 😊

Pros:
- Allows local commit without any access on the server
- Allows large scale collaboration (everyone pulls from Linus)
- Many web/cloud deployments: Gitorious, GitHub, BitBucket, GitLab
Setting up Git

**Tools**

- **git** originates from Linux so it is **command line** oriented
- There are many convenient **GUIs**. E.g., **SmartGit** or even **magit 😊**
- See the **instructions** given on the RR_webinars GitHub page

**Accessing servers**

setup your SSH public/private key to avoid typing passwords all the time (see **How do I setup ssh (even on windows?)**)

**Provide your name**

All this will go in your `~/.gitconfig`

```
1  git config --global user.name "Your Name Comes Here"
2  git config --global user.email you@yourdomain.example.com
```

**Customize even more**

```
1  [color]
2     diff = auto
3     status = auto
4     branch = auto
5  [alias]
6     wdiff = diff --color-words
7     hist = log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short
8  [branch]
9     autosetuprebase = always # you may not want this...
10  [push]
11     default = simple      # you may not want this...
```
Basic usage (demo)

You have a complete stand-alone local history!
- `git clone` / `git init` (initialize the DB)
- `git pull` sync from the remote repos
  - (actually `git pull` = `git fetch` ; `git rebase` in simple contexts)
- `git push` sync to the remote repos
- `git add` and `git commit` (or `git commit -a` if you really don’t care)

This is the basic linear history, very similar to an incremental backup.
- If you put such commands in a crontab `-e`, you will get a time machine/bacuppc/...

But git is much more than this
Actually there are branches... (demo)

- `git merge`
- `gitk` or equivalent
  - `git checkout` allows to switch from a branch to another
  - You can easily create a (local) branch to mess around. You're safe! 😊
- `git pull = git fetch; git rebase` or `git fetch; git merge`
  - Question: What is the difference? 😊
Actually there are branches... (demo)

- **git merge**
- **gitk** or equivalent
  - `git checkout` allows to switch from a branch to another
  - You can easily create a (local) branch to mess around. You're safe! 😊
- **git pull** = `git fetch; git rebase` or `git fetch; git merge`
  - Question: What is the difference? 😊

It's is a matter of taste... 😊
Collaborative Writing using Git

- Multiple collaborators editing the same file simultaneously offline
- Everyone commits locally, occasionally merging to remote
- If editing different paragraphs $\leadsto$ automatic merge
- If editing same paragraphs $\leadsto$ need to handle conflicts

Everyone has a clean history of the whole article writing process, with all intermediary versions
Working with Textual Files

- Necessary so Git can make difference between commits
- Common solutions: \textsc{LaTeX}, markdown, Org-mode, etc.
- Nowadays many user-friendly environments for \textsc{LaTeX}
- No dependencies on (proprietary) software tools

Advanced usage

Alternatively convert binaries into text files:

- \texttt{git oodiff} (based on odt2txt conversion) for OpenOffice files (.odt, .odp, etc.)
- \texttt{rakali} (based on pandoc conversion) for Office files (.doc, .docx, etc.)
Typical Workflow

- **Basic commands:**
  1. Setup repository (git config, git clone)
  2. Update to the latest version (git pull, git fetch)
  3. Contribute with your changes (git commit -a, git push)

- **Commands for handling conflicts:**
  1. Compare two versions (git diff)
  2. Use conflict solvers
  3. Merge or rebase (git merge, git rebase)

- **Commands for finding the previous version:**
  1. Going for a particular version (git checkout SHA1)
Useful Tips and Tricks

- Use a **descriptive commit message**

```
1 git commit -m "'curl -s http://whatthecommit.com/index.txt'"
```

Inspired by Michael Ernst

---

Seriously... don’t fake it 😊

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.
Useful Tips and Tricks

- Use a **descriptive commit message**
- Make each commit a **logical unit**:
  - `git add file1 file2 ; git commit`
- Avoid large indiscriminate commits (e.g., with `git commit -a`)
  - `git status`, `git diff` and `git wdiff`

Inspired by Michael Ernst
Useful Tips and Tricks

- Use a **descriptive commit** message
- Make each commit a **logical unit**: `git add file1 file2 ; git commit`
- Avoid large indiscriminate commits (e.g., with `git commit -a`)
  - `git status`, `git diff` and `git wdiff`
- Do frequent **local commits** (it’s safe 😊)
- Incorporate others’ changes frequently (`git pull`)
- Share your changes frequently (`git push`)

Inspired by Michael Ernst
Useful Tips and Tricks

- Use a **descriptive commit** message
- Make each commit a **logical unit**: `git add file1 file2 ; git commit`
- Avoid large indiscriminate commits (e.g., with `git commit -a`)
  - `git status`, `git diff` and `git wdiff`
- Do **frequent local commits** (it’s safe 😊)
- Incorporate others’ changes frequently (`git pull`)
- Share your changes frequently (`git push`)
- Remember that the tools are **line-based**
  - Never refill/rejustify paragraphs
  - Do not write excessively loooooong lines (>80 characters)
Useful Tips and Tricks

- Use a **descriptive commit** message
- Make each commit a **logical unit**: `git add file1 file2 ; git commit`
- Avoid large indiscriminate commits (e.g., with `git commit -a`)
  - `git status`, `git diff` and `git wdiff`
- Do **frequent local commits** (it’s safe 😊)
- Incorporate others’ changes frequently (**git pull**)
- Share your changes frequently (**git push**)
- Remember that the tools are **line-based**
  - Never refill/rejustify paragraphs
  - Do not write excessively loooooong lines (>80 characters)
- Don’t commit generated files
  - Don’t version temporary \LaTeX files (.aux, .toc, etc.), use `.gitignore`
  - Don’t version the result `.pdf` (unless your collaborators have problems generating it). Add only the final version of the `.pdf`, possibly with a unique name

**Advanced usage**

If you collaborate with SVN users, you may enjoy **git svn**
Outline

1. Introduction to Git and its Eco-system
2. Git for Writing Articles
3. Git for Coding
4. Git for Experiments/Data/Provenance Tracking
5. Git for Publishing
6. Conclusion and Perspective
• Remember: commit often. In this context, the `git stash` command can be useful.
  • Allows you to code/test/... with no fear
  • `git reset` / `git revert`

• Region based committing (`git commit --patch` although nobody does that through the CLI)
  • Working at a fine granularity allows the next ones to better understand what was done and decreases the risks of conflicts

• Locally rewriting your history (`git rebase -i bc23b0f`) before publishing it with `git push`

• Such history can then be exploited:
  • `git log` (git hist), `git blame`, `git bisect`

```bash
1  git config --global alias.hist \
2   'log --pretty=format:"%h %ad | %s%d [%an]" --graph --date=short'
```
Rescue!

*Git is really pretty simple, just think of branches as homeomorphic endofunctors mapping submanifolds of a Hilbert space 😊*

- Don’t force it and don’t panic... Call a friend.
  - Everything is stored in Git backend and identified by a SHA1
  - Anything is saved as soon as it is in the backend (`git add`)
- Several ways to recover objects/files states that cannot be reached by the classical history:
  - `git fsck` to find unreachable files/commits/...
  - `git reflog` (for emergency) generally more useful than `git fsck` for recovering a previous repository state (i.e., a previous commit)
- A few dangerous commands:
  - `git gc` removes unreachable old objects (>2 weeks)
  - `git prune` removes unreachable objects...
  - `rm -rf` without pushing (or cloning elsewhere) before
Merging Branches

- **Why?** Long term work to keep in sync with the rest of the world (e.g., translating some code for internationalization)
- **Merging** ("public" branch but where you’re the only developer) vs. re-basing (cleaner final set of patches, but the backup is more complex to set up)
- Depends on the meaning of the history wished within the project. Re-basing can be better for code review (sometimes, some part of the history are useless).
Merging Branches

- **Why?** Long term work to keep in sync with the rest of the world (e.g., translating some code for internationalization)
- **Merging** ("public" branch but where you’re the only developer) vs. **rebasin**g (cleaner final set of patches, but the backup is more complex to set up)

![Diagram showing loopy and nice history](image)

- Depends on the meaning of the history wished within the project. Rebasing can be better for code review (sometimes, some part of the history are useless).

Developers who do not understand this mechanism quickly end up with a huge plate of spaghetti 😞
- **git workflow** (supported by SmartGit)
http://nvie.com/posts/a-successful-git-branching-model/
Access Rights

Forge the UNIX groups philosophy still works

Linus Torvalds integrates from a few trusted developers who themselves turn integrate from various developers.

Every one pulls from Linus Torvalds

GitHub organization (similar to UNIX groups) are possible but encourages fork and pull requests done through their interface.
Outline

1. Introduction to Git and its Eco-system
2. Git for Writing Articles
3. Git for Coding
4. Git for Experiments/Data/Provenance Tracking
5. Git for Publishing
6. Conclusion and Perspective
Git as a Complement to a Laboratory Notebook

- Versioning all scripts for running experiments, pre-processing raw data, analysis and presentation
- Backing up read-only experiment results
- Commit often and separate different types of commits
- Git history helps in understanding and reproducing experiment results

**Major Challenges**

1. May depend on other projects
2. Large files and thus repositories
3. Linear history hard to explore and thus exploit
Integrating Multiple Git Sources

- Pulling and pushing changes to both projects
- Several solutions (git submodule, git subrepo)
- Work well for simple Git inside Git, but not for advanced use cases (SVN, complex branches, etc.)
Handling Large Files

- Cloning takes forever 😞
- Occupies a lot of disk space (especially for multiple projects)
- Several solutions ([git lfs](https://git-lfs.github.com/), [git annex](https://git-annex.branchable.com/))
- Probably soon becoming part of the standard (similar to Mercurial)
Improving Git History: git-labbook (demo)

- Get the data you are interested in
- Track provenance (can also be done in labbook)
- Explore and reset experiment conditions
- Expand to write an article

https://github.com/Git-labbook/git-labbook
• Get the data you are interested in
• Track provenance (can also be done in labbook)
• Explore and reset experiment conditions
• Expand to write an article


• Keeps data and code that generated it together, yet isolated
• Keeps data and code that generated it together, yet isolated
• Facilitates experiment reproduction and analysis replication
- Keeps data and code that generated it together, yet isolated
- Facilitates experiment reproduction and analysis replication
- Unites all phases of typical research study
Outline

1 Introduction to Git and its Eco-system

2 Git for Writing Articles

3 Git for Coding

4 Git for Experiments/Data/Provenance Tracking

5 Git for Publishing

6 Conclusion and Perspective
• External researchers can improve or build upon your work

• Exploring Git history allows for deeper understanding of the project evolution, not only the final results $\sim$ improved reproducibility

• Comes as a natural step if the initial study was performed with a clean methodology (example)

• Some may have valid reasons for not doing so (copyright, company policy, implementation parts that the authors do not wish to disclose now, etc.)
**Citable: GitHub + Zenodo**

- Providing DOI to the release of your GitHub

  DOI: 10.5281/zenodo.51269

- Your code becomes "citable" (although not necessarily easier to access 😔)

- Easy to use: follow the instructions

- GitHub+figshare: similar approach for figshare

**Durable: Software Heritage**

- None of this archiving is really durable (Google Code, Gitorious, Code Spaces, ...)

- Impressive ongoing work lead by Roberto Di Cosmo

  https://wg.softwareheritage.org/

  Full development history of Debian, GitHub, GNU, Gitorious, ...
Outline

1. Introduction to Git and its Eco-system
2. Git for Writing Articles
3. Git for Coding
4. Git for Experiments/Data/Provenance Tracking
5. Git for Publishing
6. Conclusion and Perspective
Where do we stand now?

We did one such webinar per month. We will stop during summer and resume in mid September with other topics (workflows, data and software archiving, evaluation challenges, . . .).

- I need volunteers! 😊
- Announcement on recherche-reproducible@listes.univ-orleans.fr and a few others but do not hesitate to crosspost.

Next webinars: New season in September!

https://github.com/alegrand/RR_webinars