Software Management

Roland Kammerer

Institute of Computer Engineering
Vienna University of Technology

27. October 2010

Overview

1. Source Code Documentation
2. Source Code Management
3. FLOSS Development

Part I

Source Code Documentation

Why is documentation important?

- Documentation is helpful and not cumbersome
- Helps developer(s) (yourself and others)
- Increases readability
- API documentation for other developers

What should be documented

- Variable/Functionnames should be self-explanatory
- Comments in Source Code (e.g., tricky sections, tricky algorithms)
- Functions (e.g., input parameters, return values $\rightarrow$ API documentation)
- No “last modified by/date”. That is the job of SCMs.
- No redundant documentation (e.g. `.h` and `.c` files). Document `.h` files

How to document source code

- Simple comments in source code
- Tools (e.g., javadoc, doxygen, pydoc,...)

Why doxygen?

- Free Software
- Easy to use
- Multilanguage support (e.g., C, C++, Java, Python, Fortran,...)
- Multiple output formats (e.g., html, latex)

Doxygen example

```
/**
 * \brief Adds two parameters
 *
 *        This function takes two integers as parameters
 *        and returns the sum of them.
 *
 * \param a Fist summand
 * \param b Second summand
 * \return Sum of a and b
 */

int sum(int a, int b)
{
   printf("Parameter a %d and b %d\n", a, b);

   return a +b;
}
```

How to use doxygen

1. Install doxygen
2. `$ doxygen -g <config-file>`
3. Edit the config-file (e.g., output directory, output format)
4. Document your code
5. Generate documentation:
   `$ doxygen <config-file>`
6. Add a rule to your Makefile

Part II

Source Code Management (SCM)

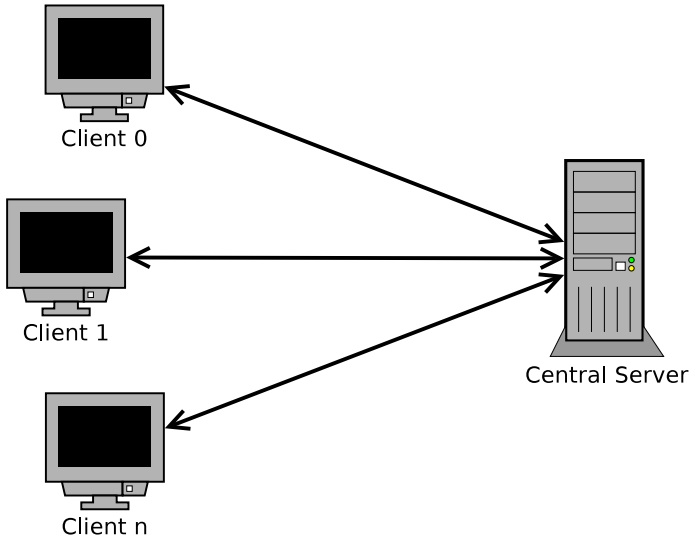
Motivation for Source Code Management

```
$ cp main.c main.c.bak
$ vim main.c
$ make
$ cp main.c main.c.bak.tmp
$ cp main.c.bak main.c
$ vim main.c
$ make
$ cp main.c main.c_tmp_segfault
$ ...
$ tar -czvf proj-20101027.tar.gz proj/
$ #and mail it
```

Motivation for Source Code Management (2)

- No manual `cp`-ing of files
- Revisions: Simple to get to a known/working state
- Sharing: No more sending tar-balls
- Branches: Work on a specific feature, then *merge* it back, or discard it
- Tracking down bugs: Automatic bi-secting between revisions (modern SCMS)
- Integrity: checksums, signed-off-by (again, modern SCMs)

Central Server Solutions



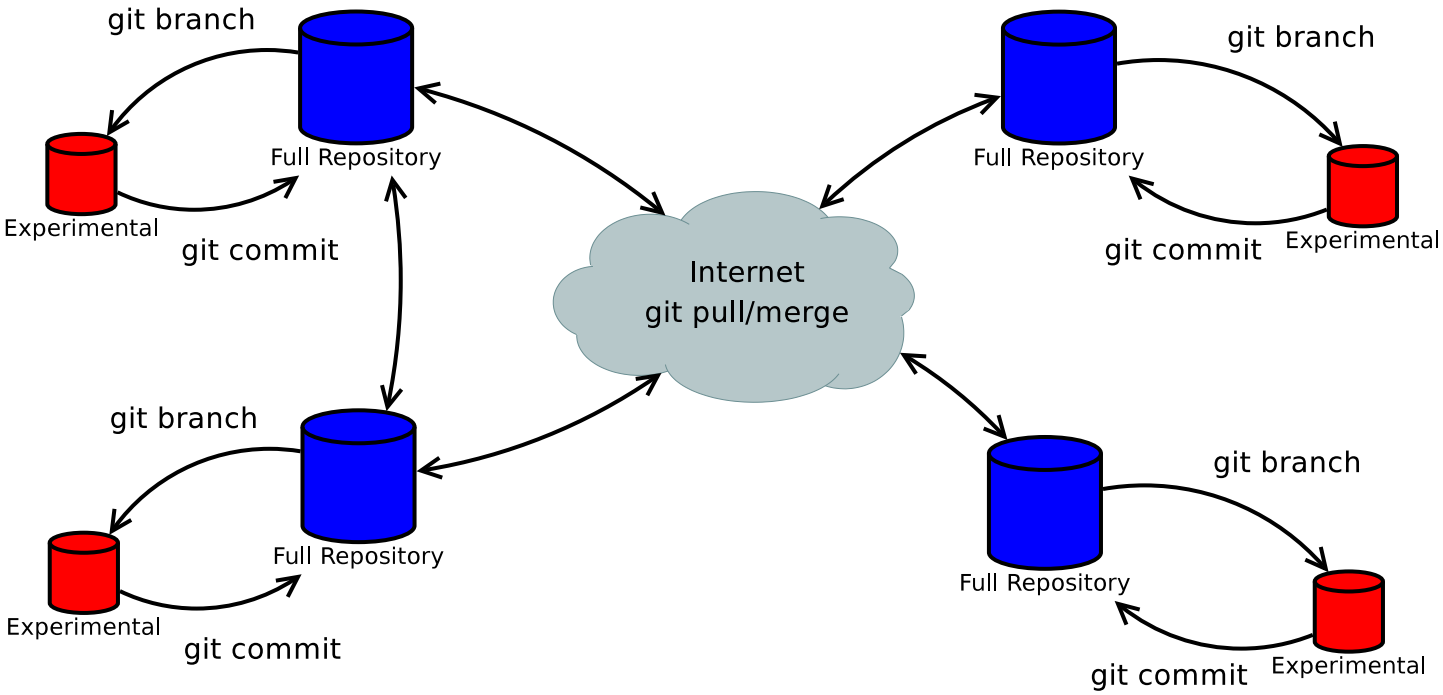
- One central server. Clients checkout/commit code from/to server
- Server is single point of failure
- Commit-access problem
- Old style: CVS cannot rename files, SVN branching is a mess
- “Advantage”: Considered simpler by beginners (e.g. no separate commit/push stage)

Central Server Solutions (2)

- Current Versions System (CVS)
  - Developed since 1989
  - Not maintained any more
- Subversion (SVN)
  - "Modern version of CVS"
  - Developed since 2000
  - Adopted by Apache Foundation $\rightarrow$ Apache Subversion

The slogan of Subversion for a while was "CVS done right", or something like that, and if you start with that kind of slogan, there's nowhere you can go. There is no way to do CVS right. – Linus Torvalds

Distributed SCMs



- No central server
- Every local repository is a *full* copy
- Allows distributed workflow
- Allows central server workflow

Distributed SCMs (2)

- Mercurial (hg)
  - Developed since 2005
  - Very popular (Google, Microsoft, Mozilla, Python,...)
- git
  - Developed since 2005
  - Used by Linux (yes, the kernel not the OS), Perl, Qt, Gnome, Ruby on Rails, Android,...)
- Bazaar (bzr)
  - Developed since 2005-2006
  - Used by Launchpad, Ubuntu, wget,...)
- darcs
  - Developed since 2002-2003
  - Very interesting from an academic standpoint (e.g., patch-theory)

History of git

- Linux (the kernel) used proprietary DVCS: Bitkeeper
- Bitkeeper crisis in 2005 (Hint: this does not happen with FLOSS software)
- Requirements for new SCM:
  - Speed
  - Simple desing
  - Non-linear development (thousands of parallel branches)
  - Fully distributed
  - Support for large projects (speed and data size)

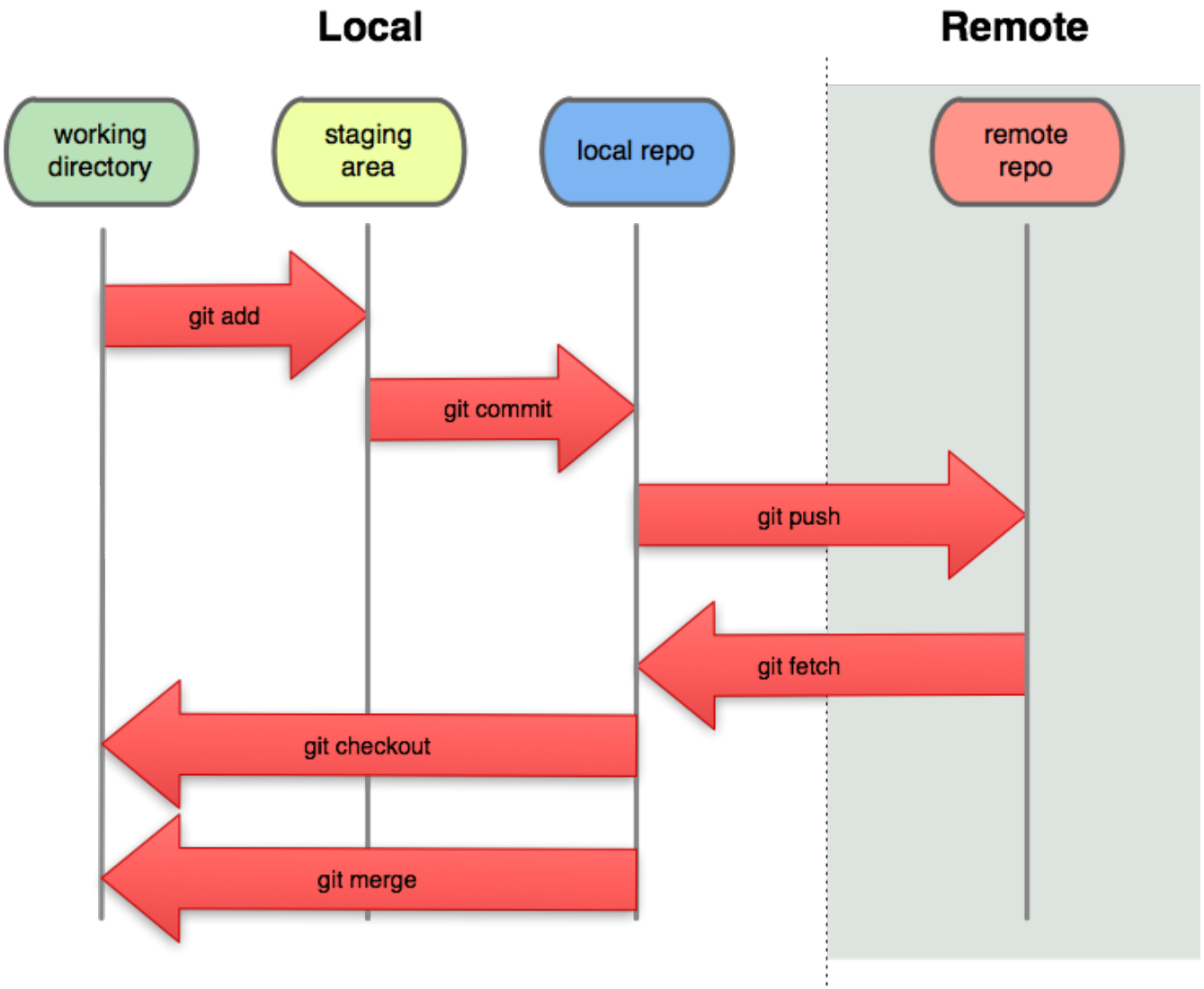
Cheap Local Branching

- Branching/Merging is easy
- In-repo branches
- Use branches for every feature (feature-branch)
- Easy to share branches with other developers

Everything is Local

- In general true for every distributed SCM
- Only fetch/pull/push need communication to outside, therefore actions are fast
- Offline commits (e.g., in trains, airplanes,...)
- No single point of failure because no central server
- Makes git very fast (init, add, status, diff, branching,...)

Everything is Local (2)



Git is Small

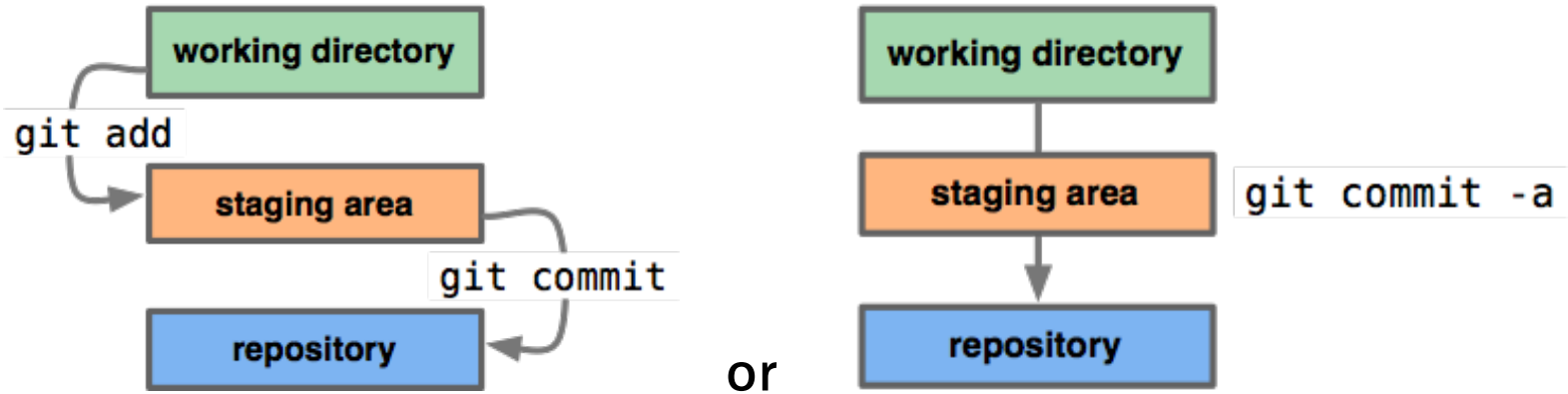
From “Django” project:

	git	hg	bzr	svn
repo alone	24M	34M	45M	
entire dir	43M	53M	64M	61M

- Nice effect: no annoying `.svn` dirs

The Staging Area a.k.a The Index

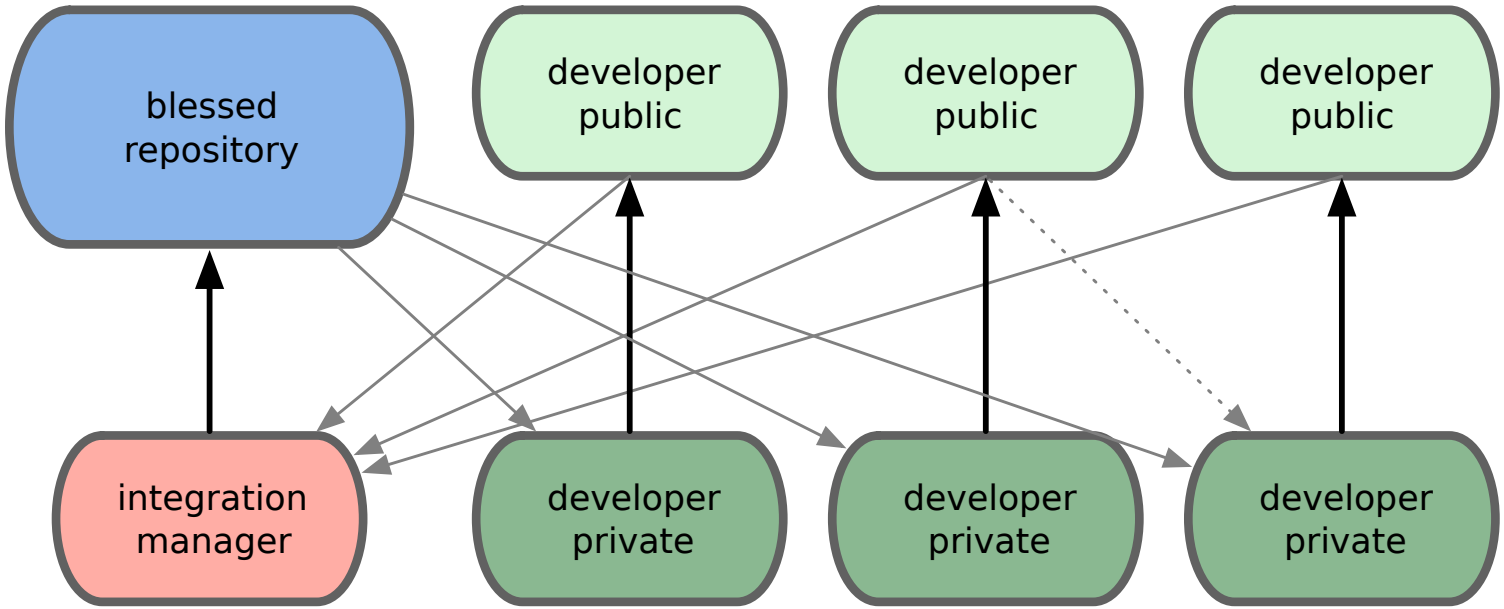
- Area to draft commits
- Seams to be overkill, but is a really nice feature



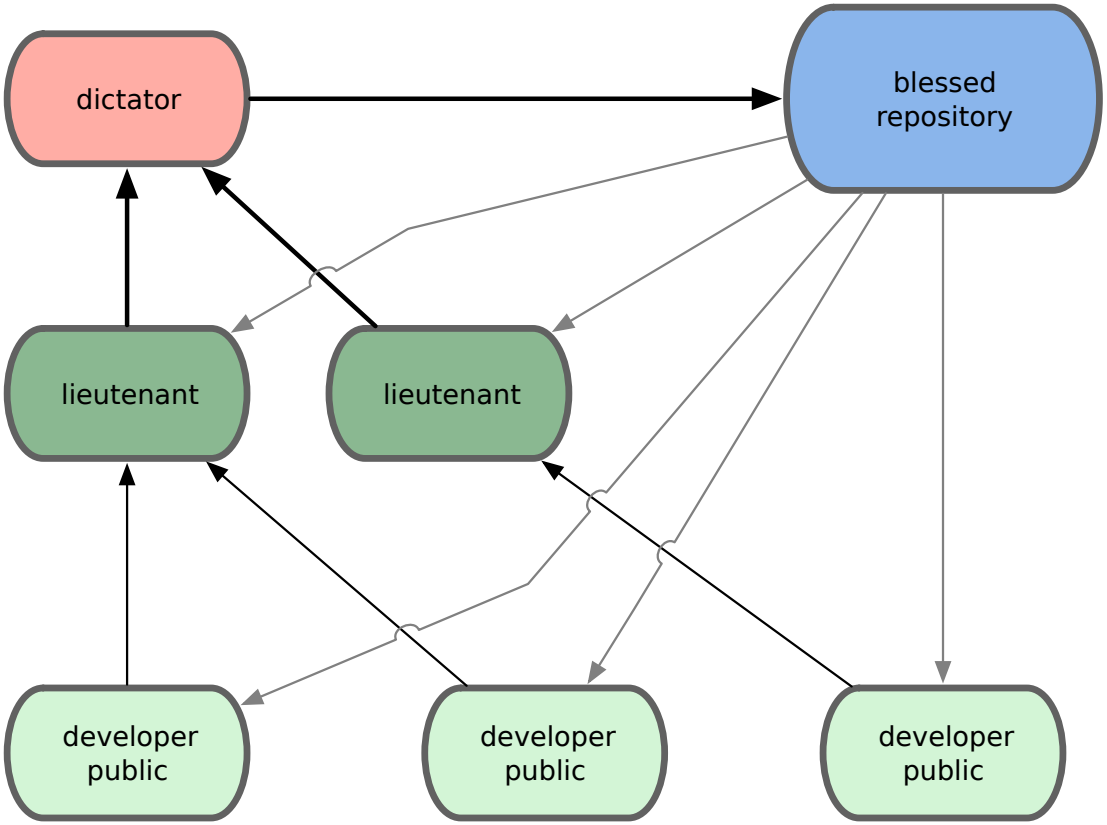
Any Workflow

- Central server workflow
- Integration Manager workflow
- Dictator and Lieutenants workflow

Integration Manager Workflow

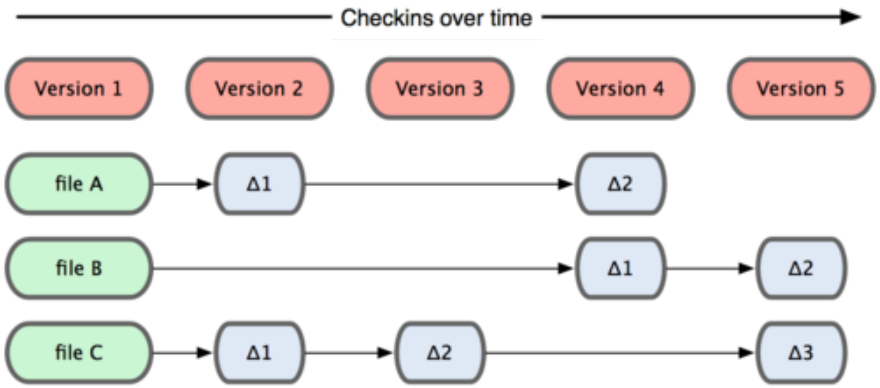


Dictator and Lieutenants Workflow

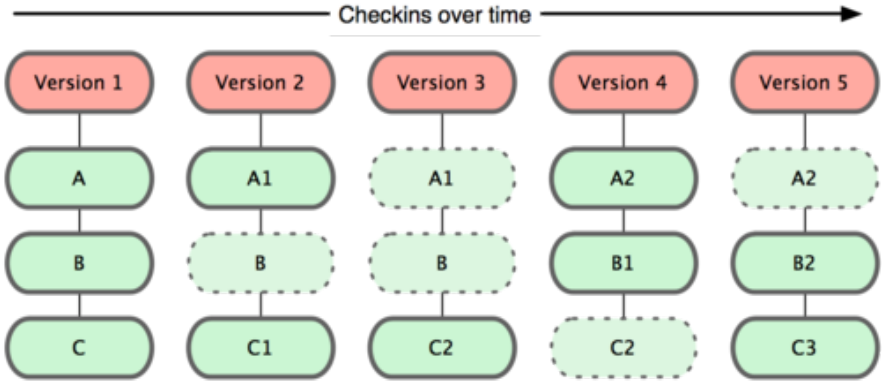


Snapshots, Not Differences

- Traditional SCMS (deltas):



- git (snapshots, “mini file system”):



Installing git

- **GNU/Linux:**

```
$ apt-get install git-core #debian/ubuntu
$ pacman -S git #arch linux
$ yum install git-core #fedora
```

- **Mac:** `http://code.google.com/p/git-osx-installer`
- **Windows:** `http://code.google.com/p/msysgit`

Basic Setup

- **Identity:**

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.net
```

- **Editor:**

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

- **Diff Tool:**

```
$ git config --global merge.tool vimdiff
```

- **Checking the Settings:**

```
$ git config --list
$ git config user.name
```

Getting Help

```
$ git help <verb>
$ git <verb> --help
$ man git-<verb>
```

Getting a Repository

- Initialize a new one

```
$ cd project
$ git init
$ git add *.c
$ git add README
$ git commit -m "initial commit"
```

- Clone an existing one

```
$ git clone git://github.com/schacon/grit.git
$ git clone git://github.com/schacon/grit.git mygrit
```

Adding and Staging Files

- `git add` **is used to add new files *and* to stage files**
- **Use** `git status` **to check the current state**
- **Use** `git diff` **to see a diff between repo and working**

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#new file:   README
#
# Changed but not updated:
#   (use "git add <file>..." to update what
#                            will be committed)
#
#modified:   benchmarks.rb
#
```

Committing Changes

- `git commit` to commit files in staging area (pops up editor)
- `git commit -m "my commit msg"` to commit and specify commit message
- `git commit -a` add files to staging area and commit them (skip explicit staging area)

(Re)Moving Files

- `rm file.txt` removes local file, but this change is not in staging area
- `git rm file.txt` removes local file, and adds this change to staging area → next commit will record the remove
- `git mv foo.txt bar.txt` moves file and stages the move

Viewing Commit History

- `git log` to see commit messages
- `git log -p` to see commit messages and a diff output. Very useful!
- Very flexible (man page for details)
- `gitk` for a graphical version

Undoing Things

- **Change your last commit (e.g. files forgotten):**
  `git commit --amend`
- **Unstage a staged file:** `git reset HEAD <file>`
- **Unmodifying a modified file:**
  `git checkout -- <file>`

Undoing Things (2)

- `git reset`: Reset HEAD pointer. Okay as long as you have not made your changes public
- `git revert`: Revert existing commits (by applying additional changes). The only way if a mistake was pushed.

Tagging

Tags give a state/commit history a human readable name

- **List tags:** `git tag`
- **Add tag:** `git tag -a v1.0 -m 'version 1.0'`
- **Tags can be signed** $\rightarrow$ **integrity**

Basics for Remotes

- Remotes specify remote repositories
- Used to get new code (pull), or to store local repository to a remote place (push)
- Extremely powerful! You decide to which external repositories you want to forward code (or to get code from)
- If you clone a repo, there is the default remote `origin` which points to the location you cloned from

```
$ git clone git://foo.net/bar.git
$ cd bar
$ git remote -v
# origin git://foo.net/bar.git
$ git remote add myserver git://myserver.net/bar.git
$ git remote -v
# origin git://foo.net/bar.git
# myserver git://myserver.net/bar.git
```

Interacting with Remotes

- **Getting data from remotes:** `git fetch <remote>`
- **Forwarding data to remotes:**
  `git push <remote> <branch>`
- **Renaming:** `git remote rename foo bar`
- **Removing:** `git remote rm bar`

Branches

- Branches are lightweight
- Creating/Switching/Merging is easy
- One of the “killer features” of git

Creating and Switching to Branches

- **Listing:** `git branch`
- **Creating:** `git branch testing`
- **Switching:** `git checkout testing`
- **Create and switch:** `git checkout -b testing`

Merging Branches

- **Merging:** `git merge <branch>`

```
$ git checkout -b hotfix
$ #fixing a bug in main.c
$ git add main.c #stage file
$ git status #ok, everything is okay
$ git commit -m 'fixed bug 123'
$ git checkout master #switch to master branch
$ git merge hotfix #merges and commits
$ git branch -d hotfix #delte the old branch
```

Remote Branches

- Whenever code shall be shared, remotes/remote branches are necessary
- Default remote branch that exists: origin/master (master branch on the remote origin)
- Fetching data from remote: `git fetch origin`
- Merge fetched data to local repository: `git merge origin/master`
- Fetch and merge: `git pull`. Fetches origin/master and merges changes
- Pushing to a remote: `git push origin hotfix`
- To push to origin/master: `git push`

Example Workflow

```
$ cd bar
$ git pull #any news?
$ git checkout -b newfeature
$ # modify e.g., main.c
$ git add main.c
$ git commit -m 'fixed it'
$ git checkout master
$ git merge newfeature
$ git push #update to server
```

Appetizer: Finding Regressions

```
$ git bisect start
$ git bisect good v2.6.18
$ git bisect bad master
Bisecting: 3537 revisions left to test after this
[65934a9a028b88e83e2b0f8b36618fe503349f8e] BLOCK...

$ git bisect bad
Bisecting: 1769 revisions left to test after this
[7eff82c8b1511017ae605f0c99ac275a7e21b867] i2c...
$ ...
git bisect reset
```

Appetizer: Imap Send

```
[imap]
        folder = "[Gmail]/Drafts"
        host = imaps://imap.gmail.com
        user = [hidden email]
        pass = p4ssw0rd
        port = 993
        sslverify = false

$ git format-patch -M --stdout origin/master |
  git imap-send
```

- Generates patches between your version and origin/master, attaches this patches to a mail in your Gmail drafts folder, uses the short commit msg as mail subject and the long as mail body.

Git and the ESE LU

- You have to use git for the ESE LU
- At a minimal and basic level
- You submit a tarball which contains your project (and the `.git` directory)
- Your repo has to contain the tag `abgabe`
- This tag will be checked out and will be used for grading
- Details follow on the LVA homepage

Part III

FLOSS (Free/Libre/Open Source Software) - Development

What is FLOSS?

- Liberally licensed software (right to use, study, change, and improve)
- Licenses: GPL, BSD, MIT, ...
- In most cases distributed development
- Great for students to get real-world experience. Use this great opportunity

What is FLOSS (2)?

- ***F***ree Software
- ***L***ibre Software
- ***OS*** Open Source
- ***S***oftware

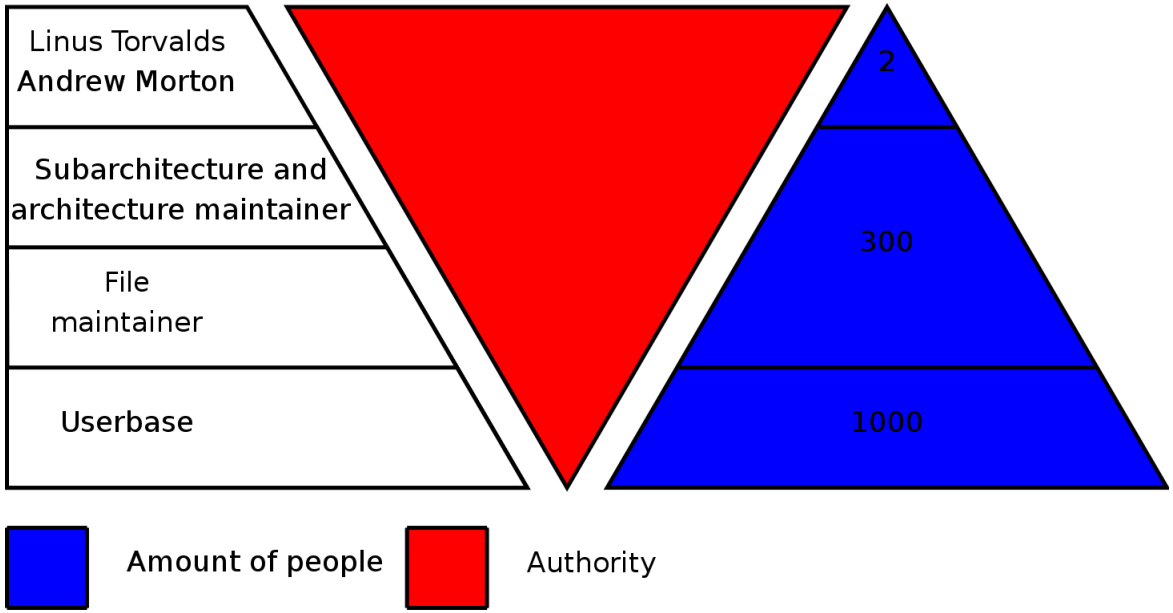
A short history

- GNU: GNU (GNU's not Unix) started to create a full OS (compilers, editors, ...). Kernel was missing.
- Minix: Nice OS kernel for teaching purposes (Andrew Tanenbaum)
- Linus Torvalds did not like the license of Minix, and GNU did not have a kernel $\rightarrow$ He started to write one

Linux Kernel Development

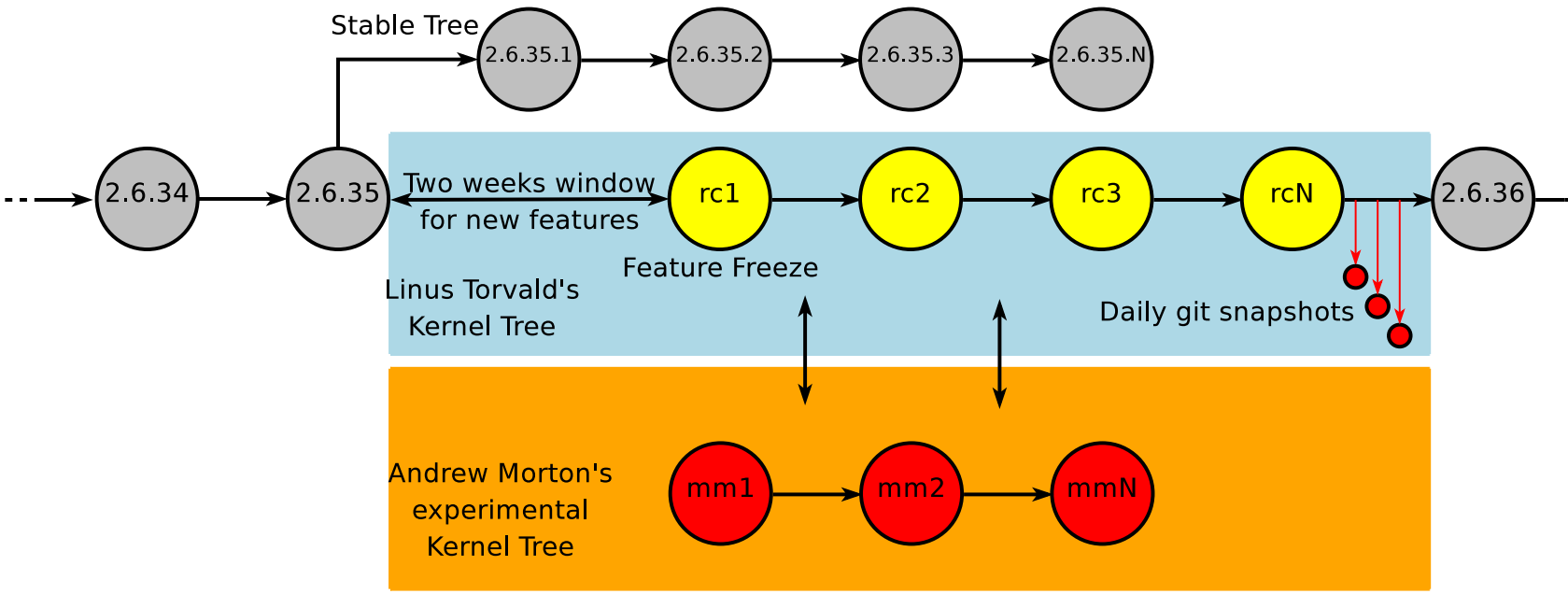
- Linux: FLOSS (gpl v2) operating system kernel (not the whole OS itself)
- Kernel of the GNU operating system $\rightarrow$ GNU/Linux
- Widely used OS kernel (Servers, Desktops, Mobile phones, Routers,...)
- BDFL (Benevolent Dictatorship For Life): Linus Torvalds
- Distributed development from the beginning (mailing-lists (LKML), distributed SCMs)

Development Model



- Distributed
- Hierarchical
- Maintainers for every sub-/architecture
- Every file has a maintainer (`/usr/src/linux/MAINTAINERS`)
- Chain of trust (signed-off-by messages)

Release Model



How git Helps

- Supports distributed development model
- Signed-off-by messages
- `git blame`
- Integrated integrity
- Rich set of tools (`git bisect`, `git imap-send`, `git archive`,...)

GNU/Linux Distributions

- Glue together software projects (Linux kernel, browser, desktop environments, boot manager,...)
- Provide packages and packages management
- Provide updates and security updates
- Different software release life cycles (stable releases vs. rolling release)
- Enormous number of distributions (advantage and disadvantage!)

Debian Release Model

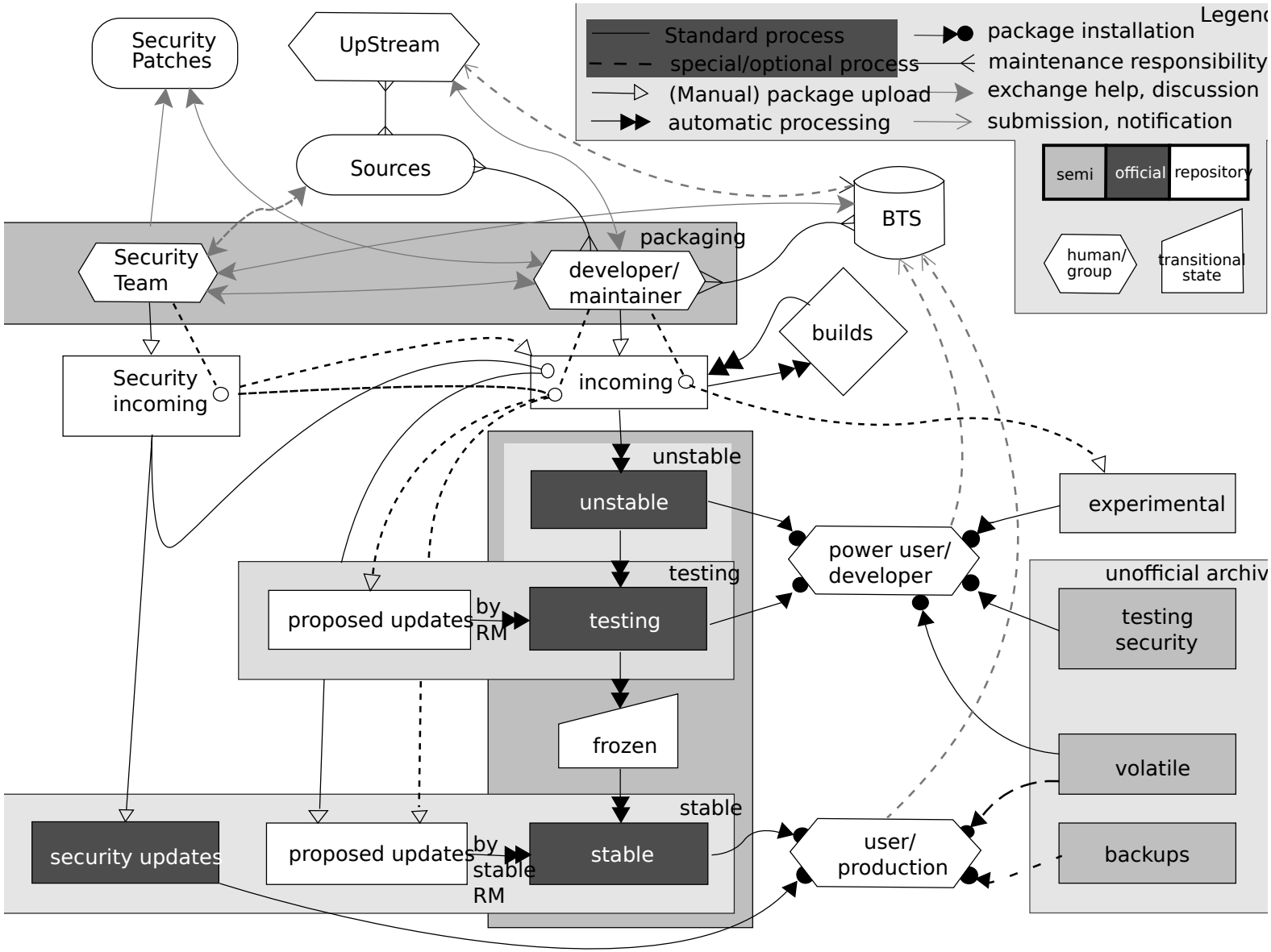


Figure: From http://en.wikipedia.org/wiki/Debian

Conclusion

- Source code documentation is important. It helps you and other developers that read the code. Tools exist
- State of the art source code management is distributed and has impact on code quality of real world projects
- FLOSS is an opportunity for students and still an emerging field (e.g., Linux in safety-critical applications?)
- FLOSS development is professional (clear development/release models, hierarchy and assignment of responsibilities, software testing, quality management,...)

References

- **Pro Git:** `http://www.progit.org`
- **Why Git is better than X:** `http://whygitisbetterthanx.com`