Git

(Why not CVS? ... because Git.)

Karel Zak
Florian Festi
Bart Trojanowski

December 20, 2007

Source code: http://kzak.fedorapeople.org/git-presentation.git

Agenda

1. **Intro**
   - Development model
   - Commands
2. **Implementation**
   - Internal objects
   - Naming revisions
3. **Getting started**
   - Configuration
   - Visualisation
4. **Branches**
5. **Real life with Git**
   - Changing History
   - Handling Patches
6. **Misc**

Section 1

Intro

What is Git?

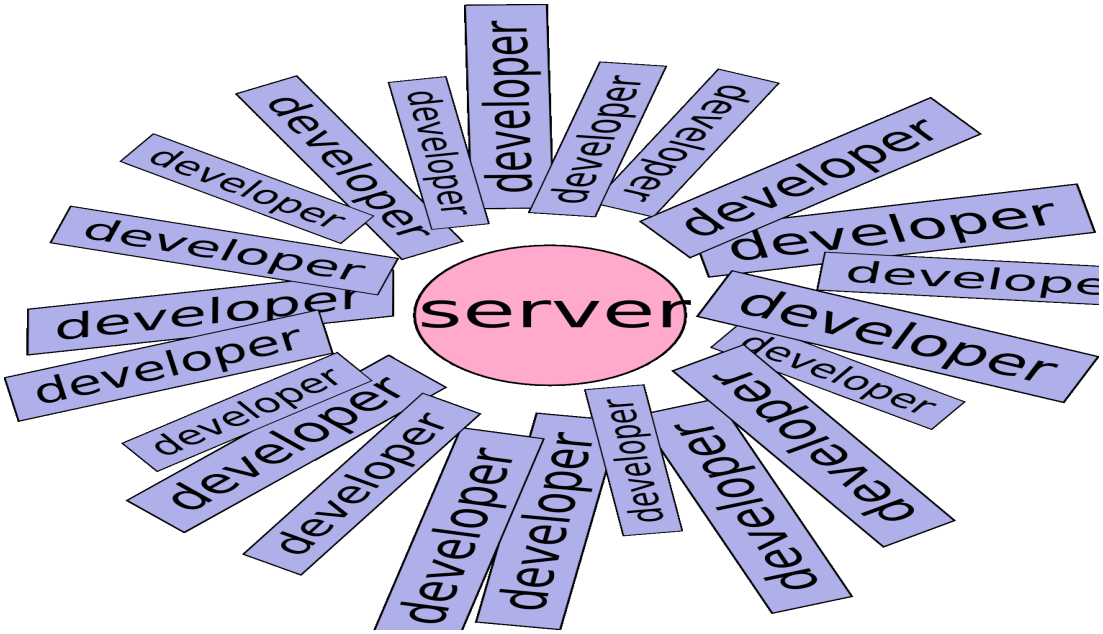
"I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'git'." (Linus Torvalds)

- fast distributed revision control system
- unusually rich command set
- provides both high-level operations and full access to internals
- originally created by Linus Torvalds for kernel development
- design was inspired by BitKeeper and Monotone
- GNU General Public License, version 2

Basic features

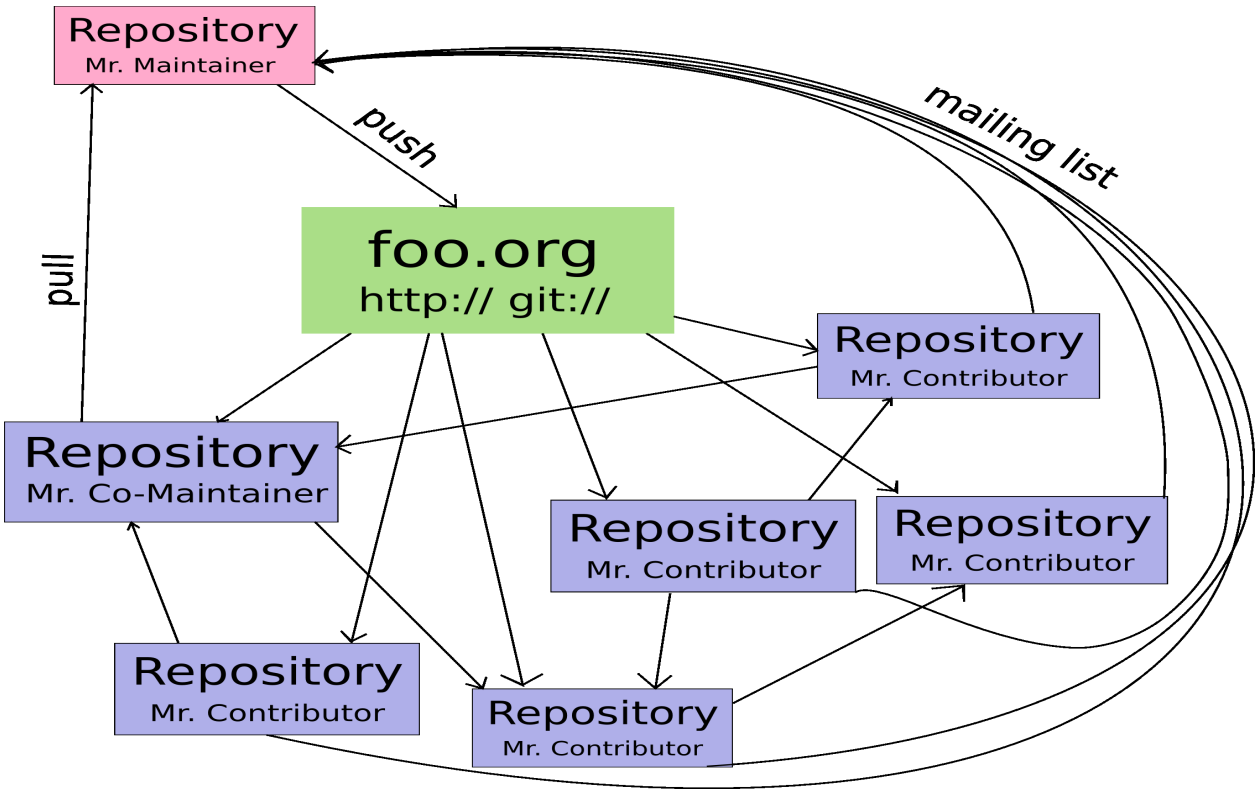
- distributed development model
- support for non-linear development (branching and merging)
- ready for large projects (very good performance)
- repositories can be easily published (git://, ssh://, http://, rsync://, ...)
- cryptographic authentication of history (GPG-signed tags)
- internally objects are addressed by content (SHA-1) – not filenames
- local branches are local only (off-line work)

Centralized model



- extra policy for write access
- SCM is not development tool, but source code archive only
- every change has impact to all developers
- no privat branches
- needs connection to server for most operations

Distributed model



- maintainer has full control over primary (his) repository
- support for non-linear development
- repositories can be easily published (git://, ssh://, http://, ...)

Git improves your work manners and habits

- branching and merging is cheap
  - you can prototype
  - you can collaborate with others developers on incomplete and unstable stuff
  - you can easily (e.g. every day) rebase your changes to new upstream code
  - merge (rebase) often minimizes conflicts between your patches and upstream
- small patch is the best patch (patch per feature/change)
  - reviewers hate huge patches
  - well separated feature or change is easy to revert
  - per item commit messages
- much less dependent on your patches going in upstream
- manage patches - not just store them

Workflow

Changes go through several stages before ending up in their final destination

working dir current checkout - editing happens here

index aka “cache” aka “staging area” - changes that are selected to be commited

commit now packaged up with a commit message and part of the history

master branch move the commits over when the feature is finished

origin get the changes upstream

Syntax

- `git <commandname> [options]`
- `git-<commandname> [options]`
- `man git-<commandname>`
- `git help <commandname>`

High level commands: Porcelain

```
$ git commit -a -s -m "cool change"
```

Low level commands: Plumbing

```
$ git rev-list --pretty=oneline v2.13..
```

Basic commands (local)

git init	creates en empty repository at `./.git`
git add	adds file contents to the next commit
git rm	removes a file
git mv	move a file
git status	shows the working tree status
git commit	records changes to the repository
git log	shows commit log
git show	shows commit (or another object)

Basic commands (remote)

git fetch	get new changes from external repository
git pull	fetch + merge
git push	write new changes to external repository
git format-patch	exports a change
git send-email	sends patch(s)
git am	applies a series of patches from a mailbox

Advanced Commands (local)

git branch	create/modify/delete branches
git checkout	switch work dir to another branch/commit
git merge	merge two or more branches
git rebase	changes starting point of a branch
git cherry-pick	copy patch from another branch
git reset	set back a branch HEAD
git bisect	find the breaking patch
git stash	save/restore current work dir changes
git gc	compactify repository and do clean ups

Section 2

Implementation

Internal objects

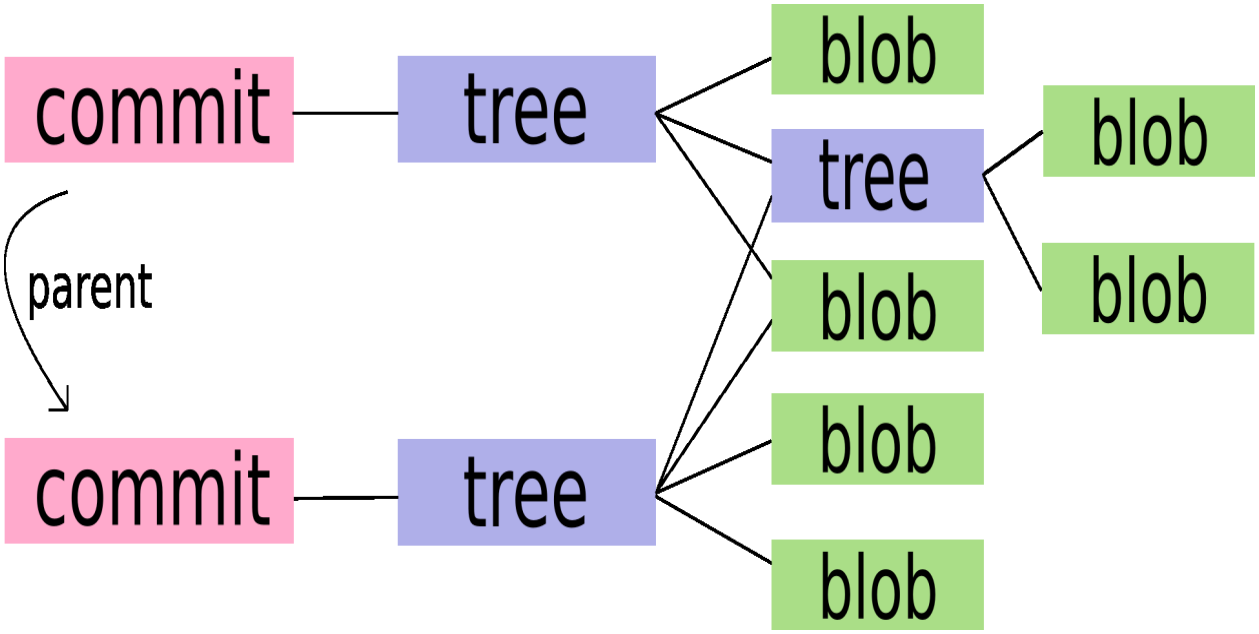
All objects are content-addressable by SHA-1.

commit refers to "tree" and "parent" (connection into the project history) and contains the commit message

tree represents the state of a single directory (list of "blob" objects and subtrees)

blob contains file data without any other structure

Internal objects



commit – connection between “tree” and “parent“

tree – state of a single directory

blob – contain file data

References

- Tag
  - contains SHA-1 sum of a commit
  - may contain an explaining message
  - can be PGP-signed
  - stays fix
  - .git/refs/tags
- Branch
  - SHA-1 sum of a commit
  - “leaf” of the history “tree”
  - follows the commits to that branch
  - .git/refs/heads
- tracked branches - .git/refs/remotes/ origin
- HEAD - the current branch
- ORIG_HEAD - HEAD before the last reset

Trust

- everything is content-addressed and based on SHA-1
- two trees are same when HEAD SHA-1 are same
- SHA-1 summ are checked to asure data integrity
- content, history and commit messages can be signed by only GPG-signing one tag

```
$ git tag -v v2.13
object 49ef7acdf77066ed05a6c828c261d332c4f54644
type commit
tag v2.13
tagger Karel Zak <kzak@redhat.com> Tue Aug 28 01:01:35 2007 +0200

stable release v2.13
gpg: Signature made Tue 28 Aug 2007 01:01:35 AM CEST using DSA key ID DC06D885
gpg: Good signature from "Karel Zak <kzak@redhat.com>"
```

Object reference

SHA-1	40-hexdigit object name
tag	human readable name for commit
`commit^n`	N-th parent
`commit~n`	N-th generation grand-parent of the named commit object, following only the first parent.
ref@{date}	specify the value of the ref at a prior point in time
:/text	commit whose commit message starts with the specified text
HEAD	refers to the head of the current branch

```
rev~3   =   rev^^^   =   rev^1^1^1
```

```
$ git reset HEAD^
```

Ranges

r1..r2 commits reachable from r2 but exclude the ones reachable from r1

r1...r2 set of commits that are reachable from either one of r1 or r2 but not from both

```
$ git log v2.13..v2.14
```

"tree-ish"

Lots of commands take a tree as an argument. A tree can be referred to in many different ways, by:

- name for that tree
- name of a commit that refers to the tree
- name of a branch whose head refers to that tree

Section 3

Getting started

Configuration

global configuration is in ~/.gitconfig

```
$ git config --global --list
   user.name=Florian Festi
   user.email=ffesti@redhat.com
   diff.color=auto
```

repository configuration is in repo/.git/config

```
$ git config --list
```

changing settings

```
$ git config --global user.name "Florian Festi"
$ git config --global user.email ffesti@redhat.com
```

.gitconfig

simple sample config

```
[user]
        name= Florian Festi
        email = ffesti@redhat.com
[diff]
        color = auto
```

see `man git-config` for all config options

Create a repository

- create a new repository

```
$ mkdir project
$ cd project
$ git init
```

- clone an existing remote repository ("origin" repository)

```
$ git clone http://foo.com/project
```

- add a next remote repository

```
$ git config remote.bar.url git://bar.com/project
$ git config remote.bar.fetch master:refs/remotes/bar/master
$ git fetch bar
```

Repository config file

```
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
[remote "origin"]
        url = ssh://login.linux.duke.edu/.../yum.git
        fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
        remote = origin
        merge = refs/heads/master
```

Visualisation

- visualization helps when working with branches
- http://git.or.cz/gitwiki/InterfacesFrontendsAndTools

History viewer:

gitk Tcl/Tk History Viewer

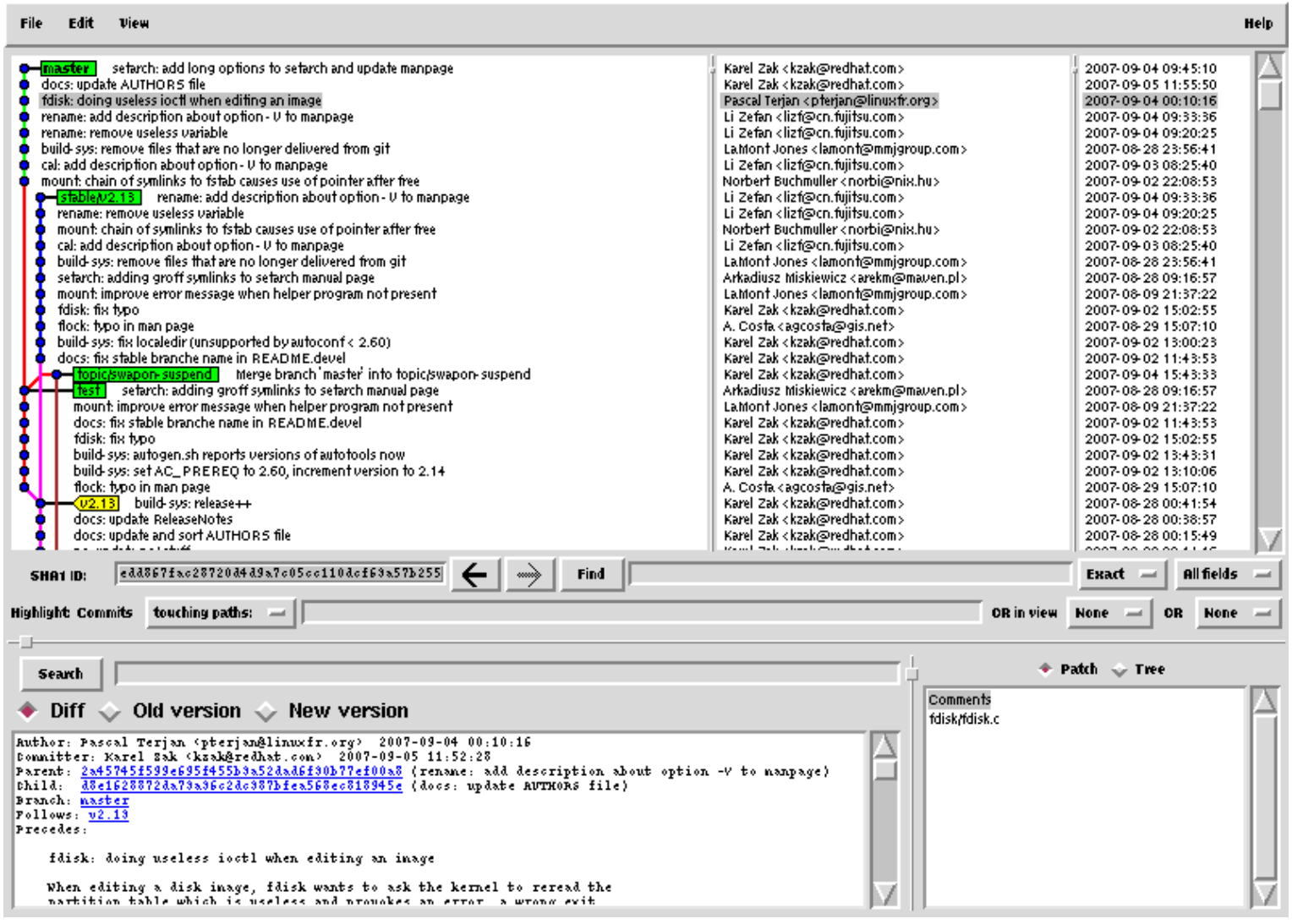
qgit Qt History Viewer, patch import/export

gitweb Web front end (CGI, mod_perl)

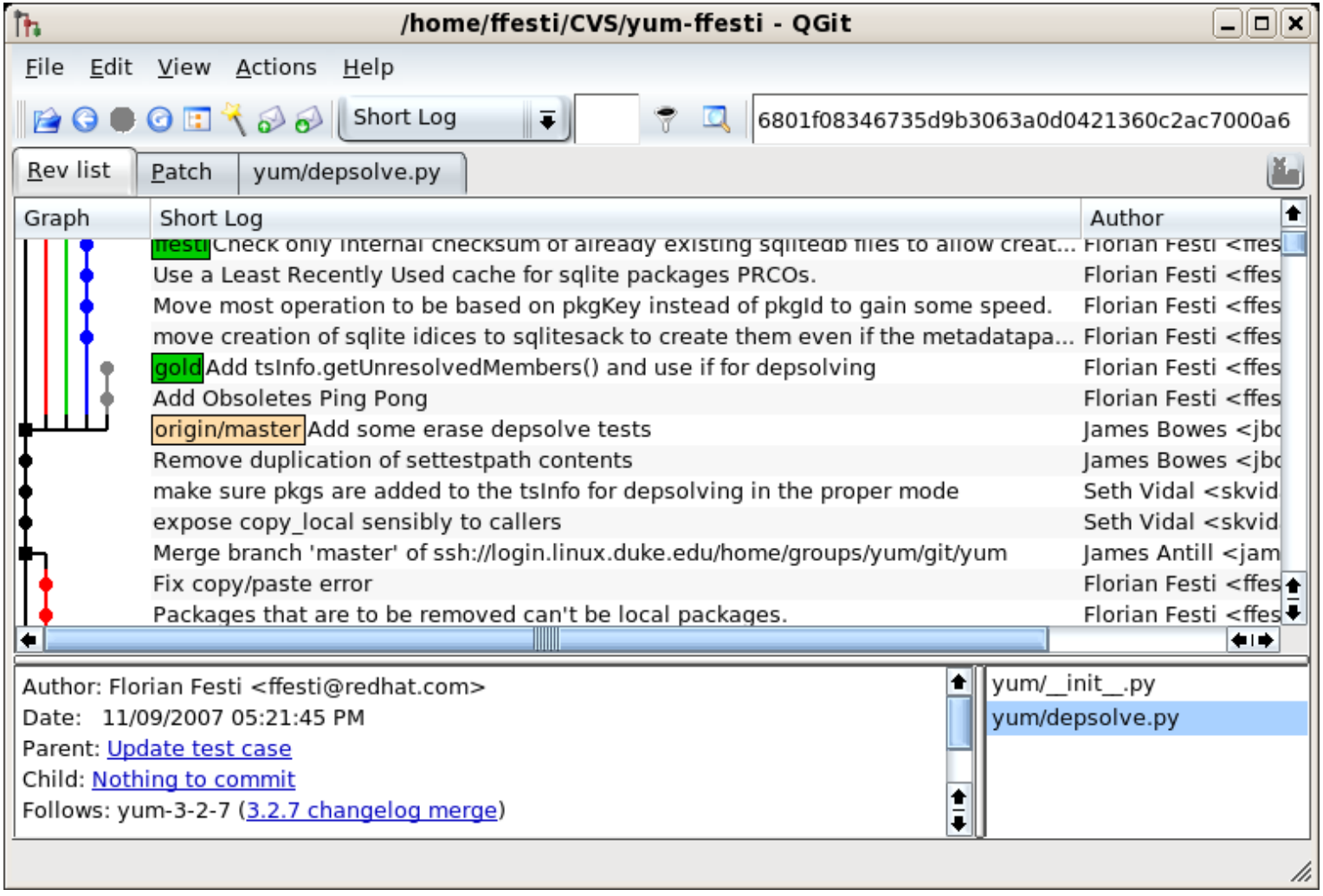
Commit tools

git gui Tcl/Tk, builtin

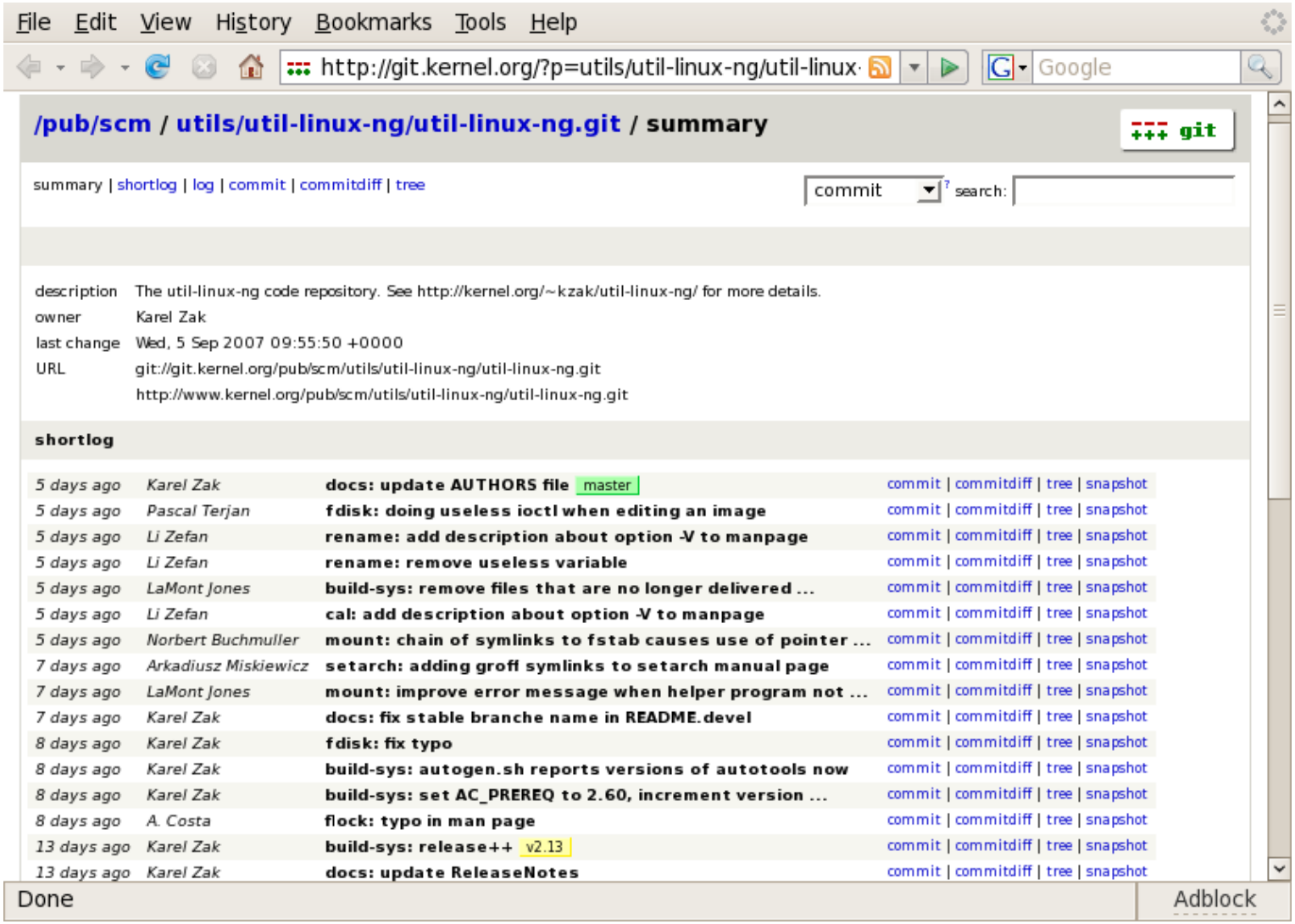
Visualisation: gitk



Visualisation: qgit



Visualisation: Gitweb



Browsing changes

git log	shows commit logs
git show	shows one or more objects (blobs, trees, tags and commits)
git blame	shows what revision and author last modified each line of a file
git whatchanged	shows logs with difference each commit introduces

```
$ git log v2.5..                        # commits since v2.5
$ git log test..master                  # commits reachable from master
                                        #          but not test
$ git log --since="2 weeks ago"         # commits from the last 2 weeks
$ git log Makefile                      # commits which modify Makefile
$ git log --pretty=format:"%h [%an]" # commit log in format
                                        #         "sha-1 [Author Name]"
$ git blame -L 10,15 foo.c              # who modified code between lines
                                          10 and 15
$ git show c1a47c171b                   # shows selected object (commit)
```

Section 4

Branches

Branches

```
          o--o--o   <-- Branch A
         /
 o--o--o--o--o     <-- master
         \
          o--o--o   <-- Branch B
```

branch is line of development

branch head is a reference to the most recent commit on a branch

- branches become remote branches when cloning a repository
- use `git branch -a` (all) or `-r` (remote) to see the remote branches

Manipulating branches

- **git branch** lists, creates, or deletes branches
- **git checkout** `<branch>` makes the current branch `<branch>`, updating the working directory
- **git checkout -b** `<branch>` creates a new branch `<branch>`check it out
- **git show-branch** shows branches and their commits
- **git diff** `<branch>..<branch>` diffs between branches

Merge branch

Before

```
                A---B---C topic
               /
          D---E---F---G master
```

Command

```
$ git merge topic
```

After

```
                A---B---C topic
               /         \
          D---E---F---G---H master
```

Rebase branch

Before

```
                  A---B---C topic
                 /
        D---E---F---G master
```

Command

```
$ git rebase master topic
```

After

```
                          A---B---C topic
                         /
        D---E---F---G master
```

Alternative

```
$ git rebase -i topic
```

Merge vs Rebase

- Merge
  - does an 3-way merge (for simple cases)
  - leads to non linear history
  - merging several branches with each other looks messy
  - keeps separate branches visible
  - Use in public repositories!
- Rebase
  - reapplies the patches on top
  - alters history - new patches with new SHA-1 sums
  - breaks work based on that branch
  - therefore not suited for published work
  - allows creating “the perfect patch” against upstream
  - Use for private work!
- Read the man pages for details!

Resolve Conflicts

- Read the messages!
- resolved stuff gets added to the index
- conflicts are applied to the work dir only
- resolve and add to index
- merge: `commit`
- rebase: `--continue`, `--abort` or `--skip`

Section 5

Real life with Git

Edit 3rd commit from the top

1. Working on branch master

```
A--B--C--D--E(master)
```

2. realized you made a mistake in commit 'B'

```
$ git checkout HEAD~3
$ git commit --amend
          .B'(HEAD)
         /
A--B--C--D--E(master)
```

3. bring back the other commits

```
$ git rebase HEAD master
          A--B'--C--D--E(master)
```

Changes in project history

- the very last patch – "`git commit --amend`" to add changes to last commit
- the latest patches – "`git reset`" to remove the last commits from the history
- organize your own branch
  - "`git cherry-pick`" patch per patch into a new branch
  - "`git rebase -i`" to freely reorder patches
- deep in project history
  - "`git rebase`" to move around large part of the history
  - "`git revert`" to add a reversed patch on top

Send a patch

Basic rules:

- **one patch per e-mail**
- don't use stupid e-mail clients (e.g. Outlook)
- **don't use attachments**
- export patches by `git format-patch`
- send patches by `git send-email`
- well formatted patch is possible to apply by `git am`
- don't forget to **keep correct authorship** (e.g when you are not author of the patch)
- **use commit messages** – a patch without comment is incomplete crap

Export patches to files

```
git format-patch [options] <since|range>
```

- creates one file per patch
- created patches are usable by **git am**

```
$ git format-patch -o ~/ HEAD~5
/home/kzak/0001-setterm-opened-file-leaving-unclosed.patch
/home/kzak/0002-sfdisk-opened-files-leaving-unclosed.patch
/home/kzak/0003-blockdev-fix-opened-file-leaving-unclosed.patch
/home/kzak/0004-tailf-opened-file-leaving-unclosed.patch
/home/kzak/0005-tests-use-losetup-s.patch
```

Patch description

Send patches by e-mail

```
git send-email [options] <file|dir>
```

Takes the patches given on the command line and emails them out.

- no attachments
- no broken patch format
- correct subject line

```
$ git send-email --to "God <father@heaven.com>" \
                 ~/0001-make-this-world-better.patch
```

Section 6

Misc

References

Git User's Manual

`http://www.kernel.org/pub/software/scm/git/docs/user-manual.html`

A tutorial introduction to git

`http://www.kernel.org/pub/software/scm/git/docs/tutorial.html`

The perfect patch

`http://www.zip.com.au/~akpm/linux/patches/stuff/tpp.txt`

The end.

Thanks for listening.

`http://kzak.fedorapeople.org/git-presentation.pdf`