Git

Paul Bienkowski (2bienkow)

KunterBuntesSeminar

2013-05-29

Gliederung

Wozu Git?

Lasst uns zusammen ein Programm schreiben!

Wozu Git?

Lasst uns zusammen ein Programm schreiben! *(Evolution eines Hackers)*

Wozu Git?

Lasst uns zusammen ein Programm schreiben! *(Evolution eines Hackers)*

1. Wir arbeiten im Etherpad!

Wozu Git?

Lasst uns zusammen ein Programm schreiben! *(Evolution eines Hackers)*

1. Wir arbeiten im Etherpad!
2. Wir arbeiten auf `rzssh1`!

Wozu Git?

Lasst uns zusammen ein Programm schreiben! *(Evolution eines Hackers)*

1. Wir arbeiten im Etherpad!
2. Wir arbeiten auf `rzssh1`!
3. Du schickst mir dann die neue Version per Mail!

Wozu Git?

Lasst uns zusammen ein Programm schreiben! *(Evolution eines Hackers)*

1. Wir arbeiten im Etherpad!
2. Wir arbeiten auf `rzssh1`!
3. Du schickst mir dann die neue Version per Mail!
4. Du schickst mir dann ein Diff per Mail!

Wozu Git?

Lasst uns zusammen ein Programm schreiben! *(Evolution eines Hackers)*

1. Wir arbeiten im Etherpad!
2. Wir arbeiten auf `rzssh1`!
3. Du schickst mir dann die neue Version per Mail!
4. Du schickst mir dann ein Diff per Mail!
5. Wir nehmen Versionskontrolle!

Wozu Git?

Lasst uns zusammen ein Programm schreiben! *(Evolution eines Hackers)*

1. Wir arbeiten im Etherpad!
2. Wir arbeiten auf `rzssh1`!
3. Du schickst mir dann die neue Version per Mail!
4. Du schickst mir dann ein Diff per Mail!
5. Wir nehmen Versionskontrolle!

→ gilt auch für Hausaufgaben in LaTeX

Was ist Versionierung?

- *Snapshots* einzelner Dateiversionen speichern
- History aufbewahren und wiederherstellbar machen
- verschiedene Versionen zusammenführen
- *(optional)* Synchronisation mit entfernten Kopien (Kollaboration)

→ mehr als nur ein Backup

Warum gerade git?

Vorteile:

- verteilt (serverunabhängig, jeder *clone* ist eigenständig/vollständig)
- schnell (lokal, Implementation in C)
- optimal für Quelltext
- FOSS

Nachteile:

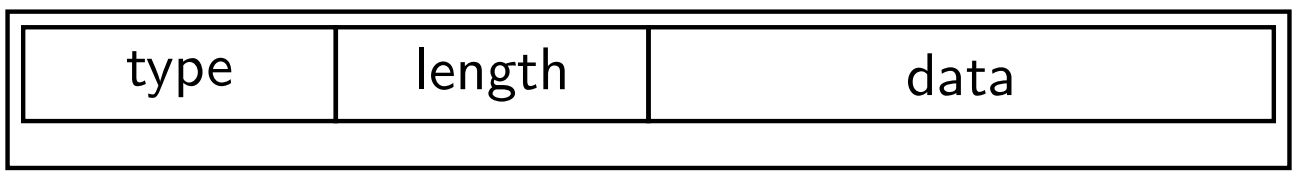
- wenig geeignet für Binärdateien
- gesamte History kann groß werden

Objekte

- Git als map-type storage
- Hashes bilden auf Daten ab
- Speicherung in einzelnen Dateien ("der Kernel macht das")
- jedes Objekt in git ist eine Datei
  - Versions-Snapshot von Dateien (*blob*)
  - Versions-Snapshot von Verzeichnissen (*tree*)
  - Commit-Informationen (*commit*)
- *tree* und *commit* enthalten Informationen in Klartext, *blob* den Dateiinhalt

Objekte

- Git als map-type storage
- Hashes bilden auf Daten ab
- Speicherung in einzelnen Dateien ("der Kernel macht das")
- jedes Objekt in git ist eine Datei
  - Versions-Snapshot von Dateien (*blob*)
  - Versions-Snapshot von Verzeichnissen (*tree*)
  - Commit-Informationen (*commit*)
- *tree* und *commit* enthalten Informationen in Klartext, *blob* den Dateiinhalt



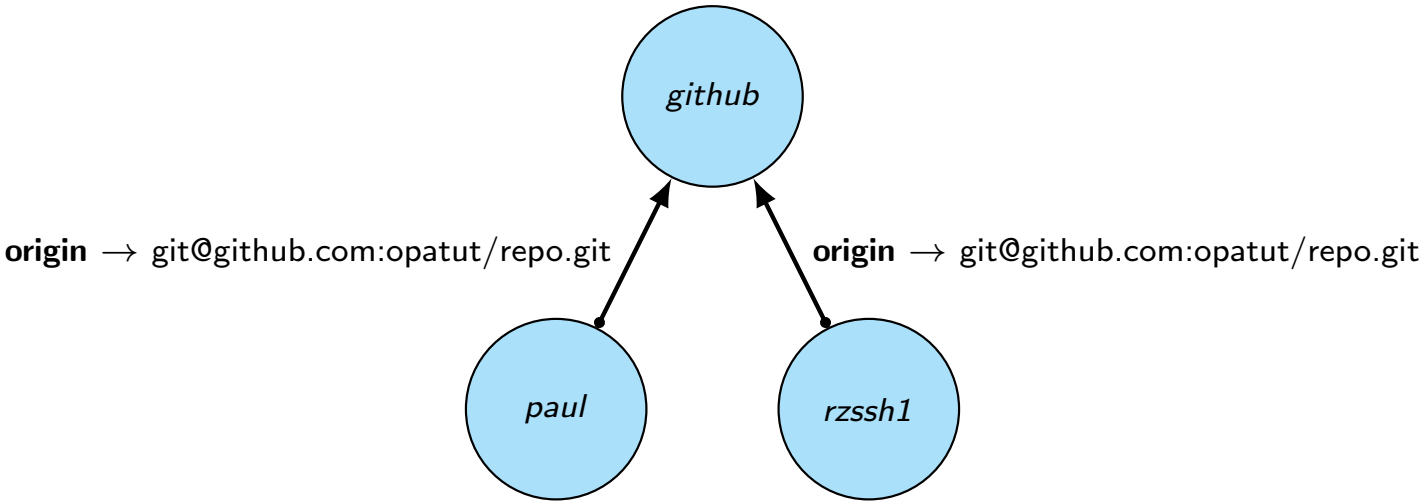
zlib compress

Clones

- in einem Repository entstehen nur neue Objekte (Dateien), sie werden nicht verändert
- jede Datei hat einen eindeutigen Namen (SHA kollidiert *praktisch* nicht)
- $\rightarrow$ Synchronisation ohne zentrale Verwaltung möglich

Einzige Ausnahme: Branch-Referenzen verändern sich.

Remotes


github
origin → git@github.com:opatut/repo.git
origin → git@github.com:opatut/repo.git
paul
rzssh1

Working copy

- In `.git/` liegen alle Versionen aller Dateien als *blob* vor
- Wie soll man damit arbeiten?
- → aktuelle Version (*HEAD*) liegt im Hauptverzeichnis
- verständliche Dateinamen (statt Hashes)
- wechseln der Version per CHECKOUT

```
git checkout 4b5c8e2f95a4407c4d0c596565b367eaca07af57
```

→ nicht möglich, wenn unversionierte Änderungen vorliegen

Branching

- Commits liegen ungeordnet vor
- $\rightarrow$ Welche ist die "aktuelle" Version?
- Ein **Branch** kann auf einen Commit zeigen ("Pointer")
- Branches haben **Namen** (z.B. *master* oder *my-feature*)
- Man kann zwischen Branches wechseln wie zwischen Commits (CHECKOUT)
- Nach dem committen zeigt der aktuelle Branch auf den neuen Commit

Branching

```
git status
```

– tikz picture here, finaltrue to show –

Branching

```
git checkout -b foobar
```

– tikz picture here, finaltrue to show –

Branching

```
git commit -m "D"
```

– tikz picture here, finaltrue to show –

Branching

```
git checkout master
```

– tikz picture here, finaltrue to show –

Branching

```
git checkout -b hotfix
```

– tikz picture here, finaltrue to show –

Branching

```
git commit -m "E"
```

– tikz picture here, finaltrue to show –

Branching

```
git checkout master
```

– tikz picture here, finaltrue to show –

Branching

```
git merge hotfix
```

– tikz picture here, finaltrue to show –

Branching

```
git branch -d hotfix
```

– tikz picture here, finaltrue to show –

Branching

```
git checkout foobar
```

– tikz picture here, finaltrue to show –

Branching

```
git commit -m "F"
```

– tikz picture here, finaltrue to show –

Branching

```
git commit -m "F"
```

– tikz picture here, finaltrue to show –

Branching

```
git checkout master
```

– tikz picture here, finaltrue to show –

Branching

```
git merge foobar
```

– tikz picture here, finaltrue to show –

Branching

```
git branch -d foobar
```

– tikz picture here, finaltrue to show –

Merging

Was passiert?

- gemeinsamen Vorgänger finden
- Änderungen ermitteln
- beide Änderungssätze auf gemeinsamen Vorgänger anwenden
- neuen Commit erstellen (automatische *message*)

Konflikte

- bei Änderung gleicher Zeile kann git nicht entscheiden, welche Änderung übernommen werden soll
- ```
  Auto-merging <filename>
  CONFLICT (content): Merge conflict in <filename>
  Automatic merge failed; fix conflicts and then commit the result.
  ```
  ```

Merging - Konflikte

```
vor dem Konflikt
<<<<<<< HEAD
meine Änderungen
=======
deine Änderungen
>>>>>>> fremder-branch
nach dem Konflikt
```