Verlässliche Echtzeitsysteme

Übungen zur Vorlesung

Isabella Stilkerich, Florian Franzmann, Martin Hoffmann

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
www4.informatik.uni-erlangen.de

2. Mai 2013

Überblick

Gerrit

- Infrastruktur für Softwareprojekte
- verwaltet `git`-Repositories
- unterstützt Code-Reviews
- für die Anmeldung notwendig:
  - OpenID eines beliebigen Anbieters, z. B.
    - RRZE ⇝ `http://openid.fau.de/<login>`
    - Facebook
    - Google

    ⇝ `http://openid.net/get-an-openid/`
  - öffentlicher SSH-Schlüssel

Anmeldung an Gerrit I

1. mit OpenID anmelden
   `http://vamos.cs.fau.de/gerrit`
   ⇝ als Benutzernamen den CIP-Login angeben
2. eine Mail mit den Namen der Gruppenmitglieder an `i4ezs@lists.cs.fau.de` schreiben
3. ggf. SSH-Schlüssel erstellen
   `% ssh-keygen -t rsa -f i4gerrit`
4. und verwenden
   - Datei `i4gerrit` nach `~/.ssh` verschieben
   - In Gerrit in *Settings* → *SSH Public Keys*: öffentlichen Schlüssel hinzufügen ⇝ `i4gerrit.pub`
   - auf dem Arbeitsplatzrechner Datei `~/.ssh/config` anpassen:

Anmeldung an Gerrit II

```
Host i4gerrit
  HostName vamos.cs.fau.de
  Port 29418
  User <login>
  IdentityFile ~/.ssh/i4gerrit
  ForwardAgent no
  ForwardX11 no
```

Anmeldung an Gerrit III

5. Rechte anpassen

Anmeldung an Gerrit IV

6. Vorgabe-Repository herunterladen:
   `% git clone ssh://i4gerrit/vezs_ss13/vorgabe`
7. Vorgabe ins Gerrit hochladen:
   - `master`-Branch ins Gerrit hochladen
     `% git push ssh://i4gerrit/vezs_ss13/gruppe<X> master:master`
   - `remote origin` in `.git/config` konfigurieren:
     ```
     [remote "origin"]
       fetch = +refs/heads/*:refs/remotes/origin/*
       url = ssh://i4gerrit/vezs_ss13/gruppe<X>
     ```
8. Gerrit kommt sehr schlecht mit `merge` zurecht
   ⇝ `rebase` als Voreinstellung für `pull` in `.git/config`:

Anmeldung an Gerrit V

```
[pull]
  rebase = true
```

9. wir werden in Zukunft mit zwei Remote-Repositories arbeiten
   `upstream` das Vorgabe-Repository
   `origin` das Gruppen-Repository
   - `remote upstream` konfigurieren
     ```
     % git remote add upstream \
           ssh://i4gerrit:/vezs_ss13/vorgabe
     ```
   - Alias für *pull von upstream* in `.git/config` konfigurieren:

Anmeldung an Gerrit VI

```
% git config alias.pu '!git fetch origin -v; \
                       git fetch upstream -v; \
                       git rebase upstream/master'
```

⇝ `git pu` zieht von `origin` und `upstream` und macht `rebase`

Table of Contents

Gerrit für Fortgeschrittene

- Gerrit kann mehr als nur `git`-Repositories verwalten

⇝ Unterstützung bei Code-Review

Code-Review: gegenseitiges

- Korrekturlesen
- Testen

von geschriebenem Code

Ablauf der Gerrit-Code-Review

1. *Autor* schiebt Patch in Code-Review
2. *Gerrit* benachrichtigt Projekteigentümer
3. *Approver*
   - inspiziert Code des Patches
   - genehmigt ihn oder fordert Verbesserungen
4. *Verifier*
   - baut und testet Code in lokaler Arbeitskopie
   - genehmigt ihn bei Erfolg
5. falls Approver und Verifier zustimmen
   ⇝ Gerrit wendet Patch auf öffentliche Codebasis an
6. falls dabei Konflikte auftreten, muss nachgebessert werden
   ⇝ weiter bei 1.

`git push` mit Code-Review

`git push origin HEAD:refs/for/master`

- legt einen Pseudo-Branch für diesen einen Push an
- Gerrit-Server antwortet mit URL zur Code-Review
- Weitere Patches für diesen Branch können durch Angabe der Change-Id hinzugefügt werden
- im Webinterface: Korrekturlesen/Genehmigung des Patches
- erst nach der Genehmigung landet der Patch in `master`

Literatur

- `http://gerrit-documentation.googlecode.com/svn/Documentation/2.2.2/index.html`

Debugger: gdb

- Ein Debugger dient zum Suchen und Finden von Fehlern in Programmen
- Im Debugger kann man u.a.
  - das Programm schrittweise abarbeiten
  - Variablen- und Speicherinhalte ansehen und modifizieren
  - core dumps (Speicherabbilder beim Programmabsturz) analysieren
    - Erlauben von core dumps (in der laufenden Shell): z. B. `limit coredumpsize 1024k` oder `limit coredumpsize unlimited`
- Programm sollte Debug-Symbole enthalten
- Aufruf des Debuggers mit `gdb <Programmname>`

Beispiel

```
void initArray(int *array, unsigned int size) {
  int i;
  for ( i=0; i<=size; i++ ) {
    array[i] = 0;
  }
}

int main(int argc, char *argv[]) {
  int *array;
  int buf[8];
  array = buf;

  initArray(buf,8);

  while ( array != buf+8 ) {
    printf("%d\n", *array);
    array++;
  }

  exit(EXIT_SUCCESS);
}
```

Befehlsübersicht

- Programmausführung beeinflussen
  - Breakpoints setzen:
    - `b [<Dateiname>:]<Funktionsname>`
    - `b <Dateiname>:<Zeilennummer>`
  - Starten des Programms mit `run` (+ evtl. Befehlszeilenparameter)
  - Fortsetzen der Ausführung bis zum nächsten Stop mit `c` (continue)
  - schrittweise Abarbeitung auf Ebene der Quellsprache mit
    - `s` (step: läuft in Funktionen hinein)
    - `n` (next: behandelt Funktionsaufrufe als einzelne Anweisung)
  - Breakpoints anzeigen: `info breakpoints`
  - Breakpoint löschen: `delete breakpoint#`

Befehlsübersicht

- Variableninhalte anzeigen/modifizieren
  - Anzeigen von Variablen mit: `p expr`
    - `expr` ist ein C-Ausdruck, im einfachsten Fall der Name einer Variable
  - Automatische Anzeige von Variablen bei jedem Programmstopp (Breakpoint, Step, ...): `display expr`
  - Setzen von Variablenwerten mit `set <variablenname>=<wert>`
- Ausgabe des Funktionsaufruf-Stacks (backtrace): `bt`
- Quellcode an aktueller Position anzeigen: `list`
- Watchpoints: Stoppt Ausführung bei Zugriff auf eine bestimmte Variable
  - `watch expr`: Stoppt, wenn sich der Wert des C-Ausdrucks `expr` ändert
  - `rwatch expr`: Stoppt, wenn `expr` gelesen wird
  - `awatch expr`: Stopp bei jedem Zugriff (kombiniert `watch` und `rwatch`)
  - Anzeigen und Löschen analog zu den Breakpoints

Fragen?