Rails doesn’t scale!

mumble mumble.. Ready for the Enterprise.. mumble mumble..

Patrick Lenz, limited overload GmbH
Rails Konferenz 2006, Frankfurt/Main

Was heisst eigentlich “Enterprise”?

Definition “Enterprise”

- Enterprise Integration
- Enterprise Speed
- Ich spreche von Letzterem

Was ist überhaupt “Scaling”?

Definition “Scaling”

- Anforderungen an die Applikation
- Passende Kombination aus Soft- und Hardware
- Abhilfe bei “überlasteten” Komponenten

Definition “Scaling”

- Ruby ist langsam
- Alles andere ist viel schneller
- Ist Ruby vielleicht trotzdem schnell genug?

Case Study: eins.de

... eins.de?

- Kundenprojekt
- Deutschlands grösste Infotainment-Community
- Gegründet 1999
- Lokale Präsenz in über 20 deutschen Grossstädten

Was noch?

- Starke Fokussierung auf user-generated Content
- Social Networks
- .. und der ganze andere Web 2.0 Krams

Die nackten Zahlen

- 40 Mio Page Impressions pro Monat
- 4TB Transfervolumen pro Monat
- 25 Mbit/s Peak Traffic
- 4.000 MySQL Queries pro Sekunde

Mehr Zahlen

- 200.000 registrierte User
- 1 Mio. neue Personal Messages pro Monat
- 2.5 Mio. gespeicherte Forenbeiträge
- 3 Mio. gespeicherte Bildkommentare

Damals...

- Aktuell 3. Software Generation
- Erste Version wird totgeschwiegen. Mit Recht.
- 2. Generation basierte auf einem PHP-basierten Redaktionssystem mit angeflanschten Community Features

Und dann?

- Relaunch on Rails im November 2005
- Kompletter Code-Rewrite
- 4 Mannmonate Entwicklungszeit
- Vorher 50.000 Zeilen (gruseliges) PHP
- Nachher 5.000 Zeilen (verdammt gut aussehendes) Ruby

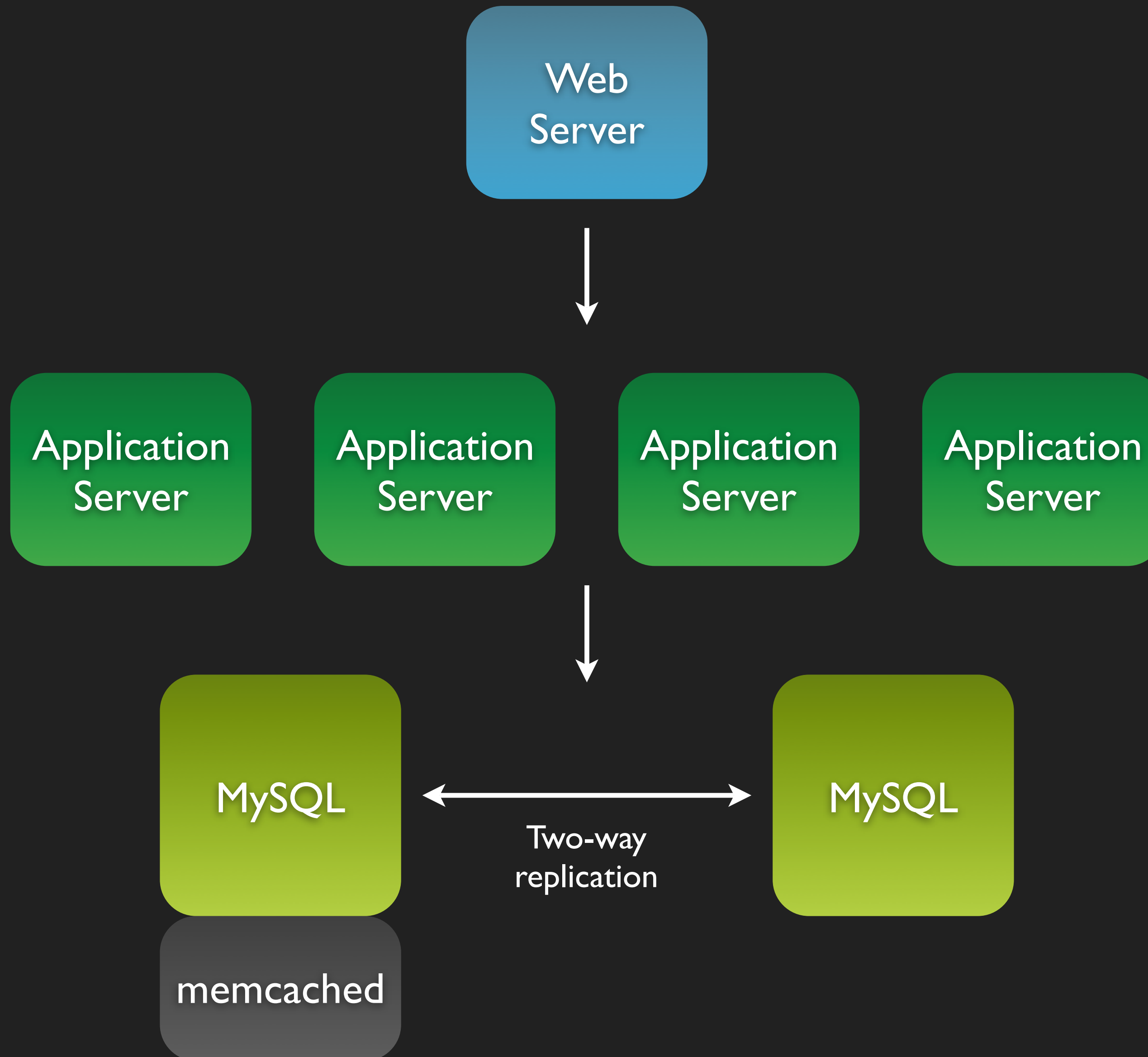
Bestandsaufnahme

Die Software

- Debian Linux
- lighttpd
- FastCGI
- MySQL
- Ruby on Rails

Die Hardware

- 1 Webserver
- 4 Application Server
- 2 Datenbankserver


Web
Server
Application
Server
Application
Server
Application
Server
Application
Server
MySQL
Two-way
replication
MySQL
memcached

Dann der Relaunch.

Ziele

- 1 Mio. Page Impressions pro Tag vor dem Relaunch
- 1.4 Mio. Page Impressions waren das Ziel.
- Bei 750.000 Page Impressions sind alle kollabiert
- Autsch.

Symptome

- Server völlig überlastet
- Furchtbar lange Ladezeiten
- Unzufriedener Kunde
- Unzufriedene User
  (gut, sind sie eigentlich immer, speziell nach einem Relaunch)

Warum langsam?

- Wusste keiner
- Wollte keiner drüber reden
- Hatte noch keiner gemacht

Hilfe!

Analyse

Irgendwas ist langsam

- Ruby
- Rails
- MySQL
- Alles.

Wie “läuft” eine Rails Anwendung?

Dedizierte Request Dispatcher

- “Persistente” Ruby Prozesse
- Kein Starten des Ruby Interpreters pro Request
- Zugriff per FastCGI, SGI, oder HTTP

Dispatcher Theorie

- Dispatcher kümmern sich nur um Rails Requests
- Statische Files (Bilder, CSS, Javascript) werden durch den Webserver ausgeliefert

Dispatcher Theorie

- Ein Dispatcher pro Request
- Mehrere Dispatcher pro Applikationsserver
- Kein Durchsatz mehr, wenn alle Dispatcher beschäftigt sind

Dispatcher warten ungern

- Request “hängt”
- Weitere Requests laufen auf
- Ein Backlog entsteht
- Alles wird langsam

Nur worauf wartet das Ding?

Daten, meistens

- Daten aus Ruby Strukturen
- Daten aus MySQL
- Daten aus memcached
- Daten aus dem Filesystem

Schneller werden

Keine Option

- Internes Rails Caching
- 30 neue Server kaufen
- Zurück zu PHP

Unverändert

- Hardware
- Software
- Personal

1. Linux

Linux

- Upgrade von Linux 2.4 auf Linux 2.6
- Besseres Memory- und Prozess-Management
- Signifikant gesunkene Last auf Applikationsservern

2. memcached

memcached

- Aufwändige Datenbank-Kalkulationen gehören in memcached
- Ruby-MemCache ist komisch
- memcache-client (Robot Coop) ist viel sympathischer

memcached sessions

- Session Store gehört nach memcached
- Weniger Write Requests für MySQL
- Performant und schlank
- Sessions ohnehin temporär

3. Dispatcher

Dispatcher

- Dispatcher sind wie Lemminge
- Sporadisch stirbt einer, meist aber nur halb
- Monitoring und erzwungener Restart

http://poocs.net/2006/2/14/killing-me-softly-keeping-dispatchers-alive

4. Ruby

Ruby

- Ruby self-compiled, mit Intel C compiler
- Anzahl Dispatcher an erwartete Requests pro Sekunde anpassen

```
1M requests / 14 hours = 20 req/sec
```

5. MySQL

MySQL

- Version 5
- Offizielle Binaries statt self-compiled/Debian binaries
- Kein Round-Robin zwischen beiden Servern aus Latenzgründen

MySQL/InnoDB

- Row-level locking
- Transactions
- Caching von tatsächlichen Daten anstatt nur Indizes

Volltextsuche

- Trennung von MyISAM und InnoDB Tables auf separate Server
- Routing von FULLTEXT Search Queries zu spezifischem Datenbankserver
- Alle InnoDB Benefits auf primärem Server

```ruby
def prepare_database_connection
  return unless RAILS_ENV == 'production' and
    ActiveRecord::Base.configurations['search']

  current_config = ActiveRecord::Base.connection.instance_variable_get('@config')

  new_config = ActiveRecord::Base.configurations.fetch(
    use_search_db? ? 'search' : RAILS_ENV).symbolize_keys

  ActiveRecord::Base.establish_connection new_config unless
    new_config == current_config
end

def use_search_db?
  true if self.class == SearchController or am_i_at?(:profile, :search)
end
```

MySQL

- “Know your database”
- Identifizierung von Slow Queries (`EXPLAIN`)
- Index-Optimierung

6. Code

Code

- gzip Compression per `after_filter`
- Weg von "Components"

Code

- Keine AJAX “live previews” verwenden
- Ein Request pro Preview (alle x Sekunden)
- Besser: Preview auf Anfrage, per Klick

Code

- Minimierung der Write Requests pro Page Impression
- Write Request muss repliziert werden
- Jeder Write Request blockiert potentielle Read Requests

7. GetText

Gettext

- Internationalization/Localization
- Mächtiger Overhead
- Speziell ab Ruby-GetText 1.5.0 langsam

7 Zeilen für ein Hallelujah

```ruby
module GetText
  @@__bound_targets = {}

  alias :uncached_bound_targets :bound_targets
  def bound_targets(klass)
    @@__bound_targets[klass] ||= uncached_bound_targets(klass)
  end
end
```

(Schneller als GetText 1.4.0)

http://poocs.net/2006/11/3/speeding-up-ruby-gettext

Heute

Heute

- Load auf allen Servern entsprechend der CPU Power
- Application Server immer noch Nadelöhr
- Weitere Skalierung nur über weitere Application Server
- Datenbank immernoch “mostly idle”

Ausblick

Mongrel?

- Mongrel ist einfach
- Mongrel benutzt HTTP
- Aber: Mongrel ist momentan noch langsamer als FastCGI

Stefan Kaes

- Template Optimizer
- query_builder
- Garbage Collector

Alternative Suche

- acts_as_searchable (HyperEstraier)
- acts_as_ferret (Ferret)
- acts_as_solr (Lucene)

Was sonst?

- Neuer Routing Code in Rails 1.2
- Neues mod_proxy_core Modul in lighttpd 1.5
- Was bringt Ruby 2.0?
- Bekommen wir Byte Code?

Zusammenfassung

Zusammenfassung

- Rails skaliert
- Eventuell Abstriche im Komfort im Austausch gegen bessere Performance
- Umdenken in der Architektur

Danke für's Zuhören.

patrick@limited-overload.de