Upgrades as a Service

Project defense, 17.2.2012
Eetu Korhonen
http://github.com/Eeko/mediawiki_uaas/

Agenda

1. Introduction
2. The problem
3. The presented solution
    - Demo!
4. Limitations of my solution
5. Alternative approaches
6. Conclusions and discussion

Introduction

Upgrade as a Service?

- Can we use external software and computing resources to provide upgrading services?
- Why would we wan't to do that?
  - **Upgrades without downtime**
  - Fault tolerance, reducing costs, experimenting...
- Can we provide generalized tools for multi-purpose upgrading?
  - Maybe

Introduction

Elastic computing resources?

- Infrastructure as a Service
    - Amazon AWS/EC2
    - Rackspace
    - OpenNebula
    - ...
- No requirement to bind big investments for physical hardware
    - Pay for what you use
    - Temporary resources

The problem described

Problem

Online upgrades

- 24/7 on-demand web-services
- Downtime costs money and time
- Big modifications might demand downtime and limited functionality between versions

Problem

Traditional solutions

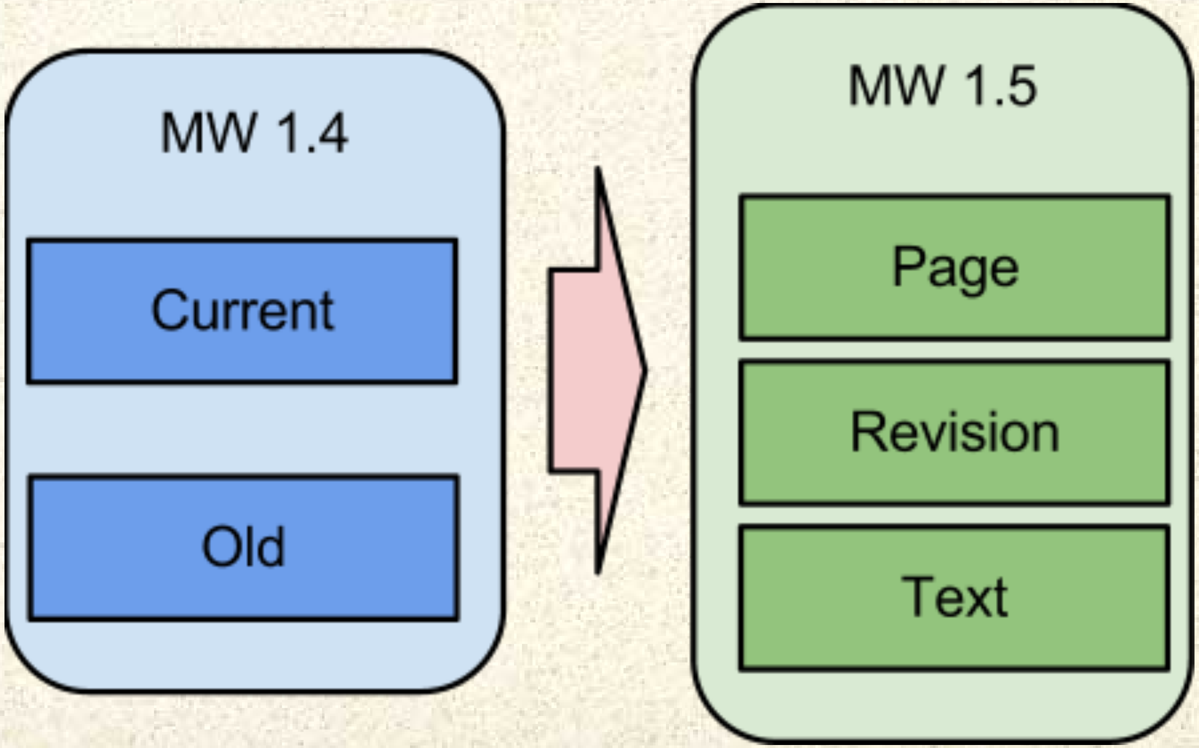
- Switch-over
  - Divide the system in half
  - First upgrade one half and direct traffic to it when it's the turn for the next half
- Rolling wave
  - Update systems in successive order

These solutions do not work if an upgrade creates backwards incompatibilities :(

Problem

Mediawiki 1.4 to 1.5 upgrade

- A major schema change requiring re-writes of the entire article-base.
- No backwards compatibility



How I solved it

My solution

A toolset to leverage Amazon EC2 computing resources for online-upgrade

1. "Parallel universe" to apply the upgrade
2. After upgrade is complete, do a "catch-up" for the entries made during the upgrade
3. When both systems in synchronous state, direct traffic to newer version

My solution

Implementation

- Good ol' LAMP
  - Amazon Linux + Apache 2 + MySQL 5.1 + PHP 5.2
- Deployed from slightly customized Amazon AMI
- All services running on the same instance
  - Not representative of a real system, where the 3-tiers are distributed

My solution

Implementation

- Set of shell scripts leveraging standard GNU-tools and Amazon EC2-tools
    - To automate the replication procedure
    - Maintain communication between nodes
- Set of Python scripts
    - Parsing query logs
    - Translating relevant entries into new schema
    - Hooking into the database and making the necessary entries

DEMO!

a screencast available at http://youtu.be/xwqOBv4cOn0

Limitations of my solution

Limitations of solution

Non-generalizable

- The translations replicate much of the application logic
- Other implementations require similar effort for modeling the functionality of the app
- E.g. in MediaWiki, new entries get added "twice"
    - First they are inserted as the current article
    - Then they are added to the archive of old-articles
    - We only need one record in the new schema

Limitations of solution

Query-logs are not the best source of data

- Shows only the entries to the database
    - Not how they are interpreted within the database
    - What will we write with NULL-entries?
    - Faults?
- Hard to parse
    - MySQL-logs by default are not even valid SQL, but more "human readable" logs
        - timestamps, extra formatting and non-sanitized nor terminated input-lines
- Better to go with binary logs?
    - Hard to program

Limitations of solution

Inefficient and unstable

- Individual transaction parsing with a high-level scripting language
    - And modern web-architectures can treat millions of transactions per second
- Can we assume a distributed system will create equivalent mappings?

Limitations of solution

Virtualization technology

- Not entirely without downtime
    - We need to shut down the server to create our replica-image
    - Could be substituted with regular db-replication, though

Alternate approaches to the problem

Alternate approaches

Use an existing database-replication tool

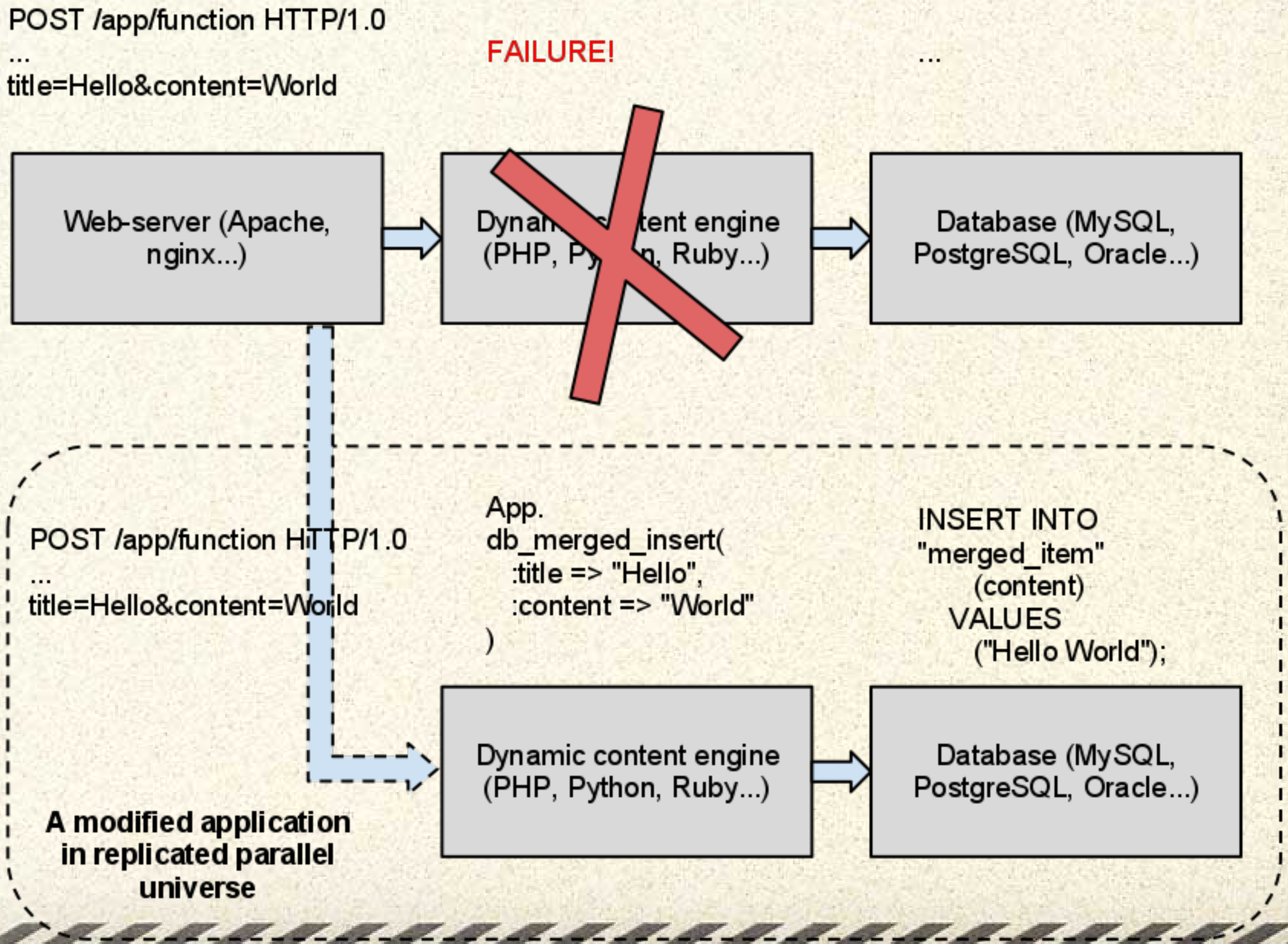
- The initial approach was in utilizing GORDA Open Replication of DAtabases toolkit
    - Proved to be too much work in this scope and skill-level
    - However, could solve the problems of limits of visibility and programmability

Alternate approaches

Do the translation for the application calls

- What if one just captures the Apache-logs and transfers nearly identical http-requests for the upgraded system?
  - Not all upgrades manage to leave the frontend untouched

Alternate approaches



```
POST /app/function HTTP/1.0
...
title=Hello&content=World
```

FAILURE!

...

Web-server (Apache, nginx...)

~~Dynamic content engine (PHP, Python, Ruby...)~~

Database (MySQL, PostgreSQL, Oracle...)

```
POST /app/function HTTP/1.0
...
title=Hello&content=World
```

```
App.
db_merged_insert(
    :title => "Hello",
    :content => "World"
)
```

```
INSERT INTO
"merged_item"
      (content)
   VALUES
      ("Hello World");
```

Dynamic content engine (PHP, Python, Ruby...)

Database (MySQL, PostgreSQL, Oracle...)

A modified application in replicated parallel universe

Alternate approaches

Leveraging the existing upgrade script incrementally

- Can we just do the inserts and restructuring again for a smaller subset (divided by timestamps) after the initial restructuring is done?
  - Maybe?
  - There are issues with equivalence
    - Some (badly programmed?) applications might depend on tables to be ordered similarly
    - FETCH FIRST N is in SQL-standard...
- But needs more research

Future work and conclusions

Conclusions

Why the initial approach would have been so nice?

- Building a similar architecture over the binary logs and standard replication protocols might be worth exploring
    - Though that's much what GORDA is supposed to deliver
    - Would there be other programmable monitoring and modification tools intercepting the replication procedure?

Conclusions

Leverage what we know about the upgrade, instead of the database?

- Can we derive an application logic from database schema?
    - And from the schema-changing scripts?
    - We've shown an consistency issue with incremental upgrade versus a singular batch upgrade
        - Does it matter?

Conclusions

Taking upgrades into account

- Can we form good practices to support better upgradeability within software development process?
    - Splitting upgrades into pieces by their downtime requirements?
    - Substituting SQL-upgrading with a suitable modeling language?
        - Can generate SQL and support an external upgrading tool?

It's hard! :(

Thank you for your time