Upgrades as a Service

Eetu Korhonen, eeko@iki.fi

January 22, 2012

http://eeko.iki.fi/
http://github.com/Eeko/mediawiki_uaas/

Abstract

**Eh, maybe writing an abstract etc. is a bit too irrelevant in this scope.

1 Introduction

This document is the end report for an independent research project performed for EURECOM[1] semester project done between July 2011 - January 2012. The purpose of the project was to research and demonstrate possibilities to leverage flexible cloud-infrastructure to provide online service updates with very little or no downtime for the end-user. The project was directed by Dr. Tudor Dumitras from Symantec Research Labs and supervised by Prof. Marc Dacier from EURECOM.

2 Problem

The high level issue under research is the possibility of performing system-updates in scale without affecting the availability of service to the end-user. With system updates changing the functionality of software, the usual case requires some unavailability period while modifications to the system are made. For this study, we reviewed the Wikipedia upgrade 1.5 from 2002, which required a 22 hour write lock due to a significant database-schema change requiring a re-write of the entire article database.[2]

There are two conventional methods to avoid availability breaks in a distributed system. One is to perform the upgrade as a switch-over, where the system is split in two halves. First one part of the system gets updated upgraded while the other part serves the clients. When the update is completed, the updated system is switched to be the client-serving end and the other part applies the update in turn. The second way is to perform the upgrade as a "rolling

[1] http://www.eurecom.fr/
[2] http://meta.wikimedia.org/wiki/MediaWiki_1.5_upgrade

wave". Here the upgrade is applied to individual nodes of the distributed system in successive order. This allows for a greater accuracy in failure localization and reduces risks of failures as the entire system (or significant parts of it) do not get compromised for upgrade-errors.

However, neither of these approaches allow a downtimeless upgrade if the upgrade causes backwards incompatibilities. Any updates into the non-updated systems should reflect into the updated system as well. In this study, such incompatibility appears with the significant database-schema change of MediaWiki 1.5.

2.1 Leveraging elastic computing resources for updates

The examined method of avoiding the incompatibility issues with downtimeless upgrade is to use external computing resources to flexibly clone the existing service into a "parallel universe", where the upgrade can be applied without touching the existing system providing service to the clients. When the upgrade is successfully applied to a machine cloned from a corresponding existing resource, the system performs some kind of catch-up with the changes inserted into the original client-serving machine and starts routing the client requests into itself. [1]

Modern cloud computing infrastructures provide us with a flexible platform for creating and utilizing external resources as needed. Applications running within Infrastructure as a Service (IaaS) providers such as Amazon EC2, Rackspace Cloud Servers or OpenNebula are by default running on virtualized hardware and are thus very easily replicable without large and permanent investments in big hardware.

2.2 Mediawiki 1.4 to 1.5 upgrade

**Here be, what kind of schema changes happen. Details about the database-queries made by application. What can be seen.

3 Solution

We built a small prototype of a software-stack capable of reading the MySQL query logs in real time from the system providing service to the end-user. Whenever it detects an update to the article-tables under update, it would create a translation of those queries compatible with the new schema. After the standard, non-modified 1.4 to 1.5 update is applied to the parallel universe clone of the system, we use an external program hooking into the new database. This program uses the recorder modifications to mimic inserting article updates to the updated database. When the system under update has reached a synchronous state with the live-system, we can shut down the old systemand the upgrade programs and route all traffic to the updated system.

3.1 Implementation details

The original test-system under study is a small-sized[3] Amazon EC2-instance running a software stack[4] capable of running MediaWiki 1.4 with a custom test-database for a set of test articles.

3.1.1 System replication tools

To automate the system replication procedure, we developed a series of bash-scripts leveraging the Amazon EC2-tools and knowledge of the details of the system under upgrade. Mainly we require the Amazon instance running details (instance number, hostname) and the application information (database name, host, username and password) for running the replication stack. The system is designed in a way, that we can use an external node with ssh- and EC2-tools access to the Amazon Instances to dowload the necessary programs from repository and start performing the upgrade process centrally, without touching the running instance providing service for the clients.

The main scripts to initiate the upgrading process are as follows:

- configs.conf – A sourcable configuration file to set the required environmental variables in the bash-scripts. Requires manual modifications to point to the EC2-node to be replicated and for the necessary database knowledge.
- prepare_for_cloning.sh[5] – Intended for installing a necessary stack of software to the node to be replicated. Such as Python 2.7 and mysql-python required by the updater software. Should also ensure that the required program-versions are available for the updating scripts at the locations specified in them.
- create_aws_replica.sh – Initiates the cloning process by copying the targeted node disk-image into a Amazon AMI (Requiring a brief shutdown of the said instance.) and starting a new identical instance with the said image. Creates a modified. configs.replica -file to include the necessary instance details of the replicated instance needed by the rest of the scripts.
- setup_replica.sh – Copies the necessary scripts and programs into the new instance.
- start_update.sh – Makes some necessary database-access modifications into the query-translating programs and launches an SSH-pipe into the original MediaWiki 1.4 node to stream the query log into a file to be readable by the local transaction-catchup programs. This is to be run in the new, replicated instance.

[3] `http://aws.amazon.com/ec2/instance-types/`
[4] Amazon Linux 2011.2 with PHP 5.2, Apache 2 and MySQL 5.1
[5] Not implemented yet as of January 22, 2012.

- std_update_mwiki14-15.sh – This script contains tools to download the newer MediaWiki 1.5 version and for running the standard upgrade-procedure to create a new copy of MediaWiki 1.5 running on top of the restructured database. This is to be run in the new, replicated instance and requires superuser access for the necessary Apache configuration and reboots.

3.1.2 Query parser, translator and mapper

The rest of the software is a series of python-programs used by update_mediawiki.py. Update_mediawiki.py requires the path to the file where the original systems query-log has been streamed as an argument and eventually writes all updates detected for existing articles into the new database schema within the parallel universe.

The program components are as follows:

- update_mediawiki.py – The entry-point of the program. Contains a main() method executing the translator-program from translator.py.
- translator.py – Contains the logic needed to use the query-log parser program (parser.py), how to interpret its returns and to translate them into SQL-queries writable by the database-hookup component. (mysql_connect.py)
- parser.py – Contains the logic required to detect and parse relevant INSERT and UPDATE queries from MySQL query-logs. The lines we wish to detect are the ones making article-updating modifications into an original MediaWiki 1.4 database.
- mysql_connect.py – Helper methods used to interact with a MySQL database.

3.2 Problems with the implementation

Our query-detector and translator approach mainly requires us to understand and re-implement large portions of the application logic within an external framework. This requires a significant amount of manual labor and would intuitively be more suitable to be integrated directly into the standard upgrading mechanisms instead of providing an external framework. More notably, individual upgrade-instructions are not recyclable for other upgrades nor can we leverage the existing SQL-upgrade instructions to automate the logic-programming.

Another issue is the limits of the data extractable from the query logs. Much of the details ending up to the database can be programmed and computed to be performed by the database itself without necessarily revealing them in the query logs E.g. generating entries via database-triggers and the auto-incrementations of id's are usually done within the database and can't be read from default query-logs. In the lower levels the database itself may perform optimizations or transaction aborts not necessarily visible to the logs or which can be insufficiently hard to predict and react for in large scale parsing. For adequate

understanding of the workings of the database, the visible plain-text query-logs are likely inadequate. The approach would be more suited to be done by using the existing database replication infrastructures and binary-logs, which reliably reveal the internal database-actions in detail. Though due to the elasticity of computing resources and low cost of upgrade-failures, failed upgrades can be tried again as often as needed.

The implementation looks into the updates as individual transaction one at a time, which is necessary as the program simulates a working application performing similar actions in live use scenario. This is hardly efficient for larger data sets and is somewhat error prone should there be unexpected modifications (such as manual inserts) to the database not detectable by the developed application. Another way would be to use the query-logs to create records of data requiring action after a stage of upgrade has been completed. For example, two updates to the same article could be marked as a single entry to a table of "touched" article-id's. Then we make a external query to the original database to stream the necessary changes into the parallel universe. This does not free us from implementing some application logic, as actions such as deleting rows or modifying their unique id's would have to be represented in the tracking logic of tainted-entries. Neither is it granted that the query-logs available present us with enough data to identify the tainted items. For example, an INSERT-query might enter their unique id as NULL and auto-increment it in the database or application-logic. Such incrementation based on the MAX(ID)-value of the new flattened text-table of MediaWiki 1.5 was required in our implementation.

** Practically this is re-implementing the application functionality in another layer. And we can't just offer an update as an automated service for an arbitary program.

** Provides functional correspondance, but the databases might not be identical and replicable

** would be better to be implemented in the database-replication layer.

4 Alternate approaches

During the course of the study, several other methods of performing the online-upgrade were speculated of and experimented with.

4.1 Using existing database-replication

** Here be a scetch of a proof why plainly leveraging an SQL-script of update does not necessarily work. Reference Gorda. [2]

4.2 Using similarity in application calls

** A short view of could we just feed apache- or network traffic logs to the parallel universe from the old system. In a pure database-schema-changing update this would work, since the server-application layer stays untouched.

4.3 Using the existing upgrade-tools

One of the more intriguing approaches would be to create a framework to be able to read the existing schema-upgrade scripts available and deduct the upgrade logic and resulting table from those. Since the target tables are expanding leading to increasing time-requirements of re-applying the updates to complete tables, we need to be able to divide the work into smaller subsets as new items and updates get inserted during the online-upgrade.

An intuitive way for such division would be to split the dataset under update by their timestamps, so that we only re-run the standard upgrade script for new items inserted after the last known item in the databases under upgrade was received. However, under some upgrades this will provide and incompatible and possibly broken result due to the unpredictability of the live-system updates.

In an example, we have a database of two tables representing a list of current states and saved histories of those states:

state_id	state_content
1	1_Fourth_state
2	2_Second_state

Table 1: State-table

history_id	history_content	history_link_to_state
1	1_First_state	1
2	1_Second_state	1
3	2_First_state	2
4	1_Third_state	1

Table 2: History-table

Suppose, that a schema upgrade would flatten these said tables into one table containing both the current state of the items and the given history of said items. An upgrade would be done with the following SQL-code:

```
-- Note, that the primary id's of the table are auto-incremented.
-- This is similar to how Mediawiki 1.4 to 1.5 upgrade handles the
-- database-flatten operation.
INSERT INTO "history" (history_content, history_link_to_state)
        SELECT (state_content, state_id)
        FROM "state";
```

After the update, the new table would look like this:

history_id	history_content	history_link_to_state
1	1_First_state	1
2	1_Second_state	1
3	2_First_state	2
4	1_Third_state	1
5	1_Fourth_state	1
6	2_Second_state	2

Table 3: Merged history-table

However, suppose that we receive a third state to the original-system during the time taken by the upgrade of the system and it receives several updates for it. We have sufficient translation logic in place to only apply the INSERT-queries for items entered to the database after we begun merging our previous entries. The code would work something like this:

```
-- We first apply the modifications from history-table
INSERT INTO "history" (history_content, history_link_to_state)
        SELECT (history_content, state_id)
        FROM "olddb.history"
        WHERE timestamp > last_update_time;

-- Then we flatten the items from the table of current states
INSERT INTO "history" (history_content, history_link_to_state)
        SELECT (state_content, history_link_to_state)
        FROM "olddb.state"
        WHERE timestamp > last_update_time;
```

Then when we would receive the following rows into the database:

history_id	history_content	history_link_to_state
5	3_First_state	3

Table 4: New row in history-table inserted during the update

state_id	state_content
3	3_Second_state

Table 5: New row in state-table inserted during the update

We would end up with a merged table looking like the following:

history_id	history_content	history_link_to_state
1	1_First_state	1
2	1_Second_state	1
3	2_First_state	2
4	1_Third_state	1
5	1_Fourth_state	1
6	2_Second_state	2
7	3_First_state	3
8	3_Second_state	3

Table 6: Merged history-table after incremental upgrade

If the upgrade would have been done with write-locks and the two tables would be merged after all 8 commits were received in the same sequential order as in our online-example, the resulting table would look like this:

history_id	history_content	history_link_to_state
1	1_First_state	1
2	1_Second_state	1
3	2_First_state	2
4	1_Third_state	1
5	3_First_state	3
6	1_Fourth_state	1
7	2_Second_state	2
8	3_Second_state	3

Table 7: Merged-history table without online-upgrading

The id's for items in the different upgrade-approaches are not equivalent. If it is used as an external-id in somewhere else in database-logic without the necessary modifications, we will encounter in faulty behaviour. To fix this, we need to implement some amount of application- or database-logic to upgrade corresponding tables with history_id references.

5 Conclusions

5.1 Future work

References

[1] Tudor Dumitras and Priya Narasimhan. Toward upgrades-as-a-service in distributed systems. In Fred Douglis, editor, *Middleware (Companion)*, page 29. ACM, 2009.

[2] Alfrânio Correia Jr., José Pereira, Luís Rodrigues, Nuno Carvalho, Ricardo Vilaça, Rui Carlos Oliveira, Susana Guedes, and Susana Guedes. Gorda: An open architecture for database replication. In *NCA*, pages 287–290, 2007.