Integration and validation testing for PhEDEx, DBS and DAS with the PhEDEx LifeCycle agent

C Boeser[1], T Chwalek[1], M Giffels[2], V Kuznetsov[3] and T Wildish[4]

[1] Institut für Experimentelle Kernphysik, Karlsruhe, Germany
[2] PH-CMG-CO, CERN, CH-1211 Genève 23, Switzerland
[3] Cornell University, Ithaca, NY, USA
[4] Princeton University, Princeton, NJ, USA

E-mail: `a.wildish@princeton.edu`

Abstract. The ever-increasing amount of data handled by the CMS dataflow and workflow management tools poses new challenges for cross-validation among different systems within CMS experiment at LHC. To approach this problem we developed an integration test suite based on the LifeCycle agent, a tool originally conceived for stress-testing new releases of PhEDEx, the CMS data-placement tool. The LifeCycle agent provides a framework for customising the test workflow in arbitrary ways, and can scale to levels of activity well beyond those seen in normal running. This means we can run realistic performance tests at scales not likely to be seen by the experiment for some years, or with custom topologies to examine particular situations that may cause concern some time in the future.

The LifeCycle agent has recently been enhanced to become a general purpose integration and validation testing tool for major CMS services (PhEDEx, DBS, DAS). It allows cross-system integration tests of all three components to be performed in controlled environments, without interfering with production services.

In this paper we discuss the design and implementation of the LifeCycle agent. We describe how it is used for small-scale debugging and validation tests, and how we extend that to large-scale tests of whole groups of sub-systems. We show how the LifeCycle agent can emulate the action of operators, physicists, or software agents external to the system under test, and how it can be scaled to large and complex systems.

1. Introduction

2. The LifeCycle agent

2.1. The LifeCycle Agent

blah blah

3. PhEDEx

3.1. PhEDEx

blah blah

4. DBS

4.1. DBS

blah blah

5. DAS

5.1. Data Aggregation Service

The Data Aggregation Service (DAS) [?] was designed to provide a uniform access to distributed CMS data-services regardless of their security policies, implementation details and data storage solution. DAS provides users ability to query underlying data-services via DAS Query Language (QL). It also aggregates meta-data information from various data-service APIs into common records suitable for end-users. Here we discuss details of DAS architecture and current status of the system.

DAS was build on top of NoSQL document-oriented database (MongoDB [?]). It provides several benefits for DAS use case: schema-less storage, embeded query language and very fast read/write operations.[1] DAS has modular design based on MVC architecture [?] and performs the following workflow upon user query:

- fetch data from underlying data-service via existing APIs
- store unprocessed data into cache database
- process data and perform its transformation to common data representation (we unify differences in naming conventions, units and data-formats, etc.)
- aggregate data on a requested key, e.g. dataset, block, etc.
- store results into merge database
- present results to end-users via and provide set of filters as well as aggregation functions to perform basic operation for slicing and representing the data suitable for user tasks

Such design allowed to developed DAS in data agnostic manner. For instance, data can be fetched from underlying data-services regardless of their data format, e.g. JSON, XML, CSV, etc. The data transformation was done via external set of mapping which was maintaned separately from DAS code development. It also gave us a few benefits which were not foreseen from the design cycle. For example, data aggregation on common entity, e.g. data name, may show discrepancies in underlying services if it exists.

DAS uses in-house Query Language which was based on entity relationship used by physicists, see [?]. It consists of the following structure:

```
<selection keys> <set of conditions> | <filters> | <aggregators>
```

The selection keys were based on well known entities such as dataset, block, file, run. The conditions were formed via key=value pairs. DAS-QL provides limited set of filters such as grep, unique, as well as set of common aggregation function such as sum, min, max. The former supported condition operators and groupping, while later can be extended via custom map-reduce functions to support more complex use cases. Therefore to express a question *Find me all datasets at a given site and show only those who has size more then 50* someone will need to write a DAS query in the following way:

```
dataset site=XYZ | grep dataset.name, dataset.size>50
```

Turns out that such flexibility were not always clear to certain set of users, mostly those who were unfamiliar with DAS-QL syntax. Therefore a further attempt was made to build native support for keyword queries on top of DAS-QL, see [?].

DAS operates as an intelligent cache in front of CMS data-services. It stores results into two caches upon provided query. The raw-cache is used to store results from data-services *as is*, while the merge-cache stores aggregated records. The lifetime of the records is based on

[1] Our benchmark shown that MongoDB can sustain 20k doc/sec for reading and 7k doc/sec for writing operations, respectively.

information provided by data-providers via HTTP headers. The records maintenance is done in a lazy fashion, i.e. upon new user query expired records are wiped out from the cache, while new ones come in. DAS server performs many operation in parallel, e.g. it sends concurrent HTTP requests to underlying data-services, processes and stores data using multiple threads, and runs multiple monitoring and pre-fetching daemons. The server runs on a single 8-core node with 24 GB of memory required for efficient MongoDB operations[2].

The discussed modular design, flexible QL and NoSQL storage, allows DAS to aggregate information from distributed data-services without imposing any requirements on them. DAS is able to deal with different security models, various APIs, data-formats and naming conventions. Right now it uses dozens of CMS data-services. Data are aggregated into JSON records based on common entity keys such that it is possible to see information from multiple data-services in a single record, e.g. run information comes from DBS, Condition DB and RunRegistry and represented as single JSON document listed information from three data-services. Daily load on DAS server is constanly growing and has about 10k queries/day with $\sim O(10M)$ records going in and out of DAS cache.

6. Conclusions

References

[1] *"Egeland R, Wildish T and Metson S 2008 Data transfer infrastructure for CMS data taking"*, XII Advanced Computing and Analysis Techniques in Physics Research (Erice, Italy: Proceedings of Science)

[2] The CMS Data Aggregation System, V. Kuznetsov, D. Evans, S. Metson, doi:10.1016/j.procs.2010.04.172

[3] http://www.mongodb.org/

[4] Model-view-controller architecture, see http://en.wikipedia.org/wiki/Model-view-controller

[5] Keyword Search over Data Service Integration for Accurate Results, V. Zemleris, V. Kuznetsov, to be published in CHEP 2013 proceedings.

[2] MongoDB relies on indexes fitted in RAM to provides its superior speed