CONCEPT AND PROTOTYPE FOR A DISTRIBUTED ANALYSIS FRAMEWORK FOR THE LHC MACHINE DATA

K. Fuchsberger, J.C. Garnier, A.A. Gorzawski, E. Motesnitsalis, CERN, Geneva, Switzerland

Abstract

The Large Hadron Collider (LHC) at CERN produces more than 50 TB of diagnostic data every year, shared between normal running periods as well as commissioning periods. The data is collected in different systems, such as the LHC Post Mortem System (PM), the LHC Logging Database and different file catalogs. To analyze and correlate data from these systems it is necessary to extract data to a local workspace and to use scripts to obtain and correlate the required information. Since the amount of data can be huge (depending on the task to be achieved) this approach can be very inefficient. To cope with this problem, a new project was launched to bring the analysis closer to the data itself. This paper describes the concepts and the implementation of the first prototype of an extensible framework, which will allow integrating all the existing data sources as well as future extensions, like hadoop clusters or other parallelization frameworks.

MOTIVATION

Next to the physics data, which is collected by the experiments and analyzed by computing centers which are distributed around the world, the LHC also produces a large amount of diagnostics data. This data is used for online diagnosis and systematic performance analysis and thus is critical for the efficient and secure operation of the accelerator. While the amount of this data (in the range of 50 TB per year) is far less than the aforementioned physics data, it is already in a range where naive usage of scripts and simple applications reach their limits due to data I/O limitations and memory consumption. At the time of writing relevant data is mainly stored in two main systems, depending on the nature of the data:

- **LHC Post Mortem System (PM):** This system stores data at a high resolution over short time ranges. The collection mechanism is event based: Triggered by a timing event, different equipment sends data to a central server (PM server). The server collects all the data and groups them into events. The main purpose of the PM system is to collect data after failures (e.g. a beam abort or a power abort of an electrical circuit) and to store and analyze the data for later diagnosis.
- **Common Accelerator Logging Service (CALS):** CALS provides continuous logging of equipment signals all around the LHC. The logging frequency is defined per equipment and/or signal type and is typically much slower than for the PM system. The CALS provides the main source for systematic performance and trend analysis.

Besides these two systems, there are other custom made logging mechanisms which mostly write to files in proprietary formats (E.g.: Orbit data or Tune spectra). While the two main systems (PM and CALS) both provide a Java Application Programming Interface (API) to access the data, this is not the case for most of the dedicated file formats. Even with the available java APIs systematic analysis of data from different sources is complicated, when it comes to correlation and alignment between different datasources, since this responsibility is completely left to the user of the data.

Another limitation is that data extraction is time consuming, since all the data has to be transported over the network. This is very often circumvented by users by extracting the data once and then storing it to (custom) files stored locally on their computers, which then are read for subsequent analysis. This produces a large amount of redundant data in diverse formats, with all the problems of storage space and backups.

These facts, together with the high rate of code duplication resulting from the various analysis implementations, are the main arguments which emphasize the need for a common solution for the analysis of this data.

REQUIREMENTS

The problems outlined in the previous section, led to the following requirements for a common analysis framework:

- **Calculations close to the data:** This avoids transport of huge amounts of data and allows optimizations based on the nature of the data.
- **Horizontal Scalability:** The system should be able to grow with the amount of data and the amount of users.

Out of further considerations, the framework should also handle the following situations in a consistent way:

- **Data incompleteness:** The data might be incomplete or the metadata might change over time: As an example, lets consider a vector of beam positions (orbit) around the LHC ring. This vector consists of one value per beam position monitor (BPM). If we would, for example, take the difference of an orbit at a given time and an orbit from a previous year, it might happen that a BPM was removed or added. Therefore, a blind subtraction of the two vectors (by index) can lead to totally wrong results. Although this problem appears in almost all the data, everyone who has to do some analysis has to take care of these specialties on their own, because no generic solution is in place.

ISBN 978-3-95450-139-7

- **Data invalidity:** Even if there were values recorded for certain signals, it might be that some of these values are meaningless (If we consider again e.g. orbit data, it could well be that BPMs deliver some values, although there is no beam in the machine). To diagnose and handle such specialties, usually considerable domain knowledge is required. And again, the treatment of such cases in subsequent calculations is left to each user, which easily leads to code duplications.
- **Mathematical operations:** Most of the calculations that have to be performed on data, are similar to each other and are mostly of a statistical nature (E.g. average over time, subtraction, scaling). Again, users of the data are re-implementing such operations over and over which leads to a lot of code duplication and is an error prone process. Furthermore some of these operations produce only a few values as a result, while they require a huge amount of data to be extracted from the storage system, which is a time consuming operation as mentioned before.
- **Physical units:** For many different reasons (efficiency, signal resolution), signal values of different systems are stored in many different units on the storage layer (PM, CALS). When using the data in analysis, the user has to be very careful to perform the correct conversions at the right moment, when e.g. correlating data from one system with data from another system.
- **Error propagation:** Error estimation of results from a chain of numerical operations is a nontrivial task which is currently left to the user. This results in inconsistent treatment of errors and makes it hard to compare different analysis, although a standard strategy could be provided at least for the most common operations.

The vision for such an analysis framework emerged from the development of a system for automatic analysis of test data resulting from LHC hardware commissioning tests [1, 2]. Although the development was done with this framework in mind, the design decisions always took further usage and extensibility into account.

OVERVIEW

The simplified architecture of the analysis framework is depicted in Fig. 1. Users can interact with the framework by the use of a Java API which is implemented in the form of an embedded domain specific language (eDSL) [3], which limits the functionality provided to the user to the actually available functionality.

This language produces a tree of expressions which is then sent to the execution framework. This approach completely de-couples the description of the analysis from the execution part and provides the necessary flexibility to change implementation parts later, if required. The execution framework itself can be run on a dedicated server or alternatively embedded in another server or application and will be the main focus of the subsequent sections.

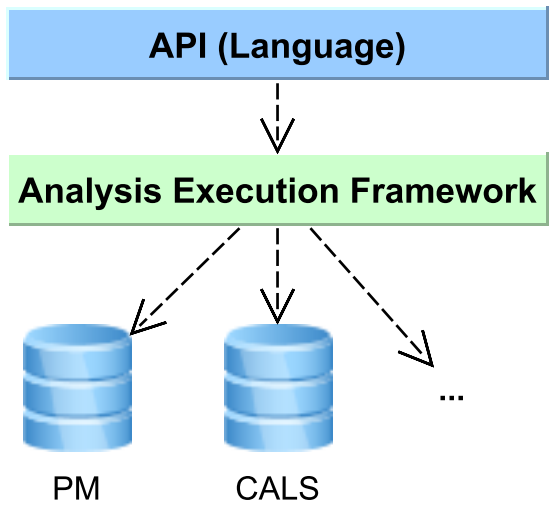

Figure 1: Overview of the Framework Architecture.

EXECUTION FRAMEWORK

Due to the complete disentanglement of the analysis language from the execution layer, as described in the previous section, it is possible to provide a very flexible and modular layer of execution. This was a deliberate choice in our design since it provides us with the possibility to start off with a very simple, but not optimized solution, and later slowly optimize the execution without any impact on the language level. Further, the chosen concept allows plugging different data sources and/or processing engines into the system.

To illustrate the basic concepts in some detail, we will consider the (simplified) example as shown in listing 1.

Listing 1: Simple Assertion

```
assertThat(I_MEAS).isLessThan(55.0,
    AMPERE).at(PM_EVENT_TRIGGER);
```

The code in this example will instruct the analysis framework to check, if the signal (`I_MEAS`) is less than 55.0 amperes, at a certain point in time (denoted by `PM_EVENT_TRIGGER`) and to return the result. The code is translated into a tree of expression objects, similar to the one shown in Fig. 2.

As illustrated in this figure, some of the nodes of the produced tree already contain all of the necessary information (Nodes D, E and F). We call those nodes 'resolved nodes'. For other nodes (which we call 'unresolved nodes') the result still has to be calculated (Nodes A, B and C). The goal for the evaluation of an expression is to resolve all nodes of such a tree until the root node (the Assertion node in the example) is resolved. In the simplest possible version of an

ISBN 978-3-95450-139-7

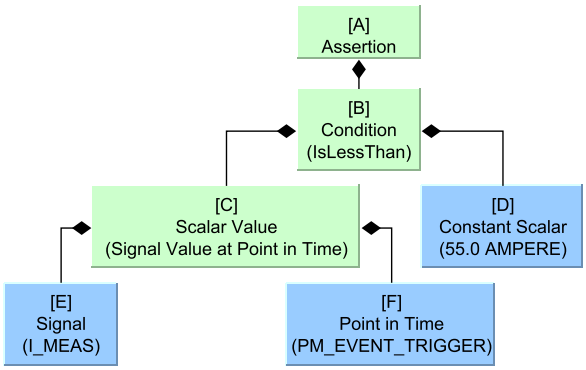

Figure 2: Expression tree created from listing 1. Characters in square brackets ([]) are labels for the nodes; blue nodes are already resolved ones (values known); green ones are unresolved (to be calculated).

execution framework, each unresolved node could be resolved as soon as all of its children are resolved. In the initial state of the example as depicted in Fig. 2, node C could be resolved by this method, while node B could not (yet). In the current implementation of the execution framework the resolving mechanism is a little bit more complicated, as described in the following:

The relations between the main components (Java objects) of the framework that are involved in this resolving-process are shown in Fig. 3.

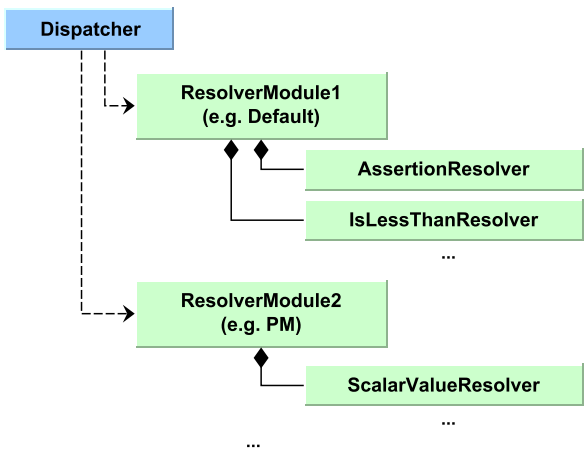

Figure 3: Relations of components involved in the resolving process (simplified).

The component which orchestrates the resolving process is the so-called `Dispatcher`. It uses a set of `ResolverModules`. Each of them provides a set of `Resolvers`. A resolver is a class which implements a simple interface: it has one `invoke(..)` method which takes an unresolved node as an argument and returns a resolved one. Optionally, it can have a `canInvoke(..)` method with the same argument. The algorithm, which is implemented in the dispatcher is basically the following (characters A to F refer to the nodes of the example):

1. Starting from the root node (A), the dispatcher queries each of the resolvers if it can resolve the given node (with the current state of the tree). A resolver can resolve a given node,
   - if the signature of the `invoke(..)` method matches the node to resolve
   - and if the `canInvoke(..)` method returns `true` for the node (if such a method exists).
2. If a resolver can resolve the given node, then the dispatcher remembers this resolver in a list for each node as a potential candidate and continues checking all the other resolvers.
3. If none of the resolvers can resolve the node, then the same procedure is recursively invoked with all the children of the node (e.g. node B in the example).
4. Else (if at least one potential candidate resolver is found for the node), the dispatcher stops the iteration through the tree in this branch and does not try to resolve the child nodes anymore (in the initial state of the example, this would happen for node C).
5. As soon as the iterations through all branches are finished (no more resolvers can be found, which could resolve nodes from the current tree state), the dispatcher selects one of the candidate resolvers per node (in the simplest implementation the first one), invokes it and rebuilds the tree with the resolved node.
6. After all resolvers return, the whole algorithm is looped starting at item (1), until all nodes (including the root node) are resolved.

The main advantage of this algorithm, compared to a simple bottom-up resolving, is that it is easy to add new resolvers. For example, a resolver could be added, that resolves a node which still contains some unresolved children (i.e. resolves a whole sub-tree without the need to resolve the bottom nodes). In our example this could be a resolver which performs the less-than operation for certain signals directly in the database without the need to extract the signal values (which might be much faster, since no data has to be extracted at all). The algorithm would automatically select the new resolver for those signals, as soon as it would be plugged into the system.

This current implementation is purely serial and has a lot of potential for improvements:

- The invocation of the resolvers (5) could be easily done in parallel.
- The selection of the candidate resolver for a node, that in the end will be invoked, could be fine-tuned. For example, some machine-learning algorithm could be put in place, which selects a fast candidate based on experience from invocations on 'similar' nodes.

ISBN 978-3-95450-139-7

- Currently, no caching is in place. Thus, if the same node appears in different expressions, it will be re-evaluated each time. This could be easily avoided by some caching mechanism.

PARALLELIZATION

As mentioned in the previous section, one obvious way to optimize the analysis is parallelization. For the longer term, an approach has to be found which scales horizontally, as soon as the needs and the load on the analysis framework would grow. At a first glance, a map-reduce approach as provided e.g. by hadoop [4] looked promising. Nevertheless, due to the strong focus on data stored in files this approach turned out less effective as expected because of the following reasons:

- None of the currently available data is available in a useful file format for hadoop: PM stores its data in a custom file format and CALS keeps the data in a relational database.
- The data volume for our intermediate results (results of individual nodes) are relatively small and the I/O to files would be an overhead. This argues for a more flexible solution, where data is kept in memory.

The usage of a hadoop cluster is not at all excluded for the future (nothing argues against adding resolvers that are based on hadoop map-reduce at a later stage, if we see an advantage). This could be useful in the near future, as the serialization format of the PM system will be migrated soon to Apache Avro, which is a splittable, hadoop compatible format.

Having the dispatching mechanism in mind, as described in the previous section, another parallelization approach appeared to be a better match: Akka [5], which is an actor based system for Java and Scala. The idea was, that if we would implement the dispatching mechanism in the akka way, the process could be easily scaled to a cluster, if necessary. To test this theory, a small student project was launched: The goal was to port an already existing algorithm (purely Java) for the search of so-called 'Unidentified Falling Objects' (UFOs) to a version using akka [6]. The result was very promising. Within a few days it was possible to create a version of the algorithm, which scales natively to as many cluster nodes as desired and proved to give a performance increase by a factor of 20, although all cluster node instances were still run on the same machine, as illustrated in Fig. 4. This enormous gain was only possible because the CALS started supporting parallel data extraction. After this result, the next steps will be to verify the results on a small cluster using multiple nodes and later to replace the current dispatcher with a version using akka and evaluate the limits of the parallel data extraction.

CONCLUSION AND OUTLOOK

We outlined the main motivation for a central analysis framework for accelerator machine data at CERN and proposed solutions for different aspects of it. All these proposals were backed by experience from working prototypes produced as parts of other applications in the previous years. It looks very promising to finally combine all the mentioned concepts to establish a first version of such an analysis framework with additional added values like domain objects and a strongly typed analysis language with parallelized execution. Still, our minds have to be kept open for better solutions. However, such changes will be easy to integrate if necessary, due to the strongly layered and modular design of the proposed approach.

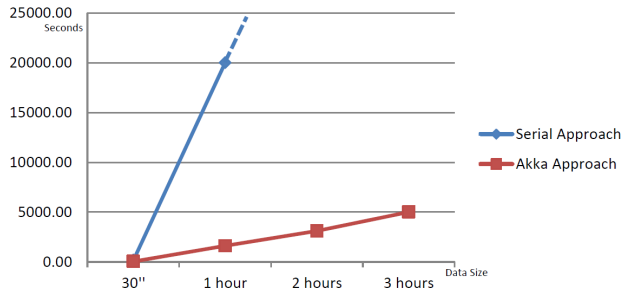

Figure 4: Execution times of the parallelized algorithm (using akka) for UFO search compared to the completely serial approach. The x-axis represents the amount of data analyzed in hours of logged data. The vertical axis represents execution times of the search algorithm in seconds (Courtesy of E. Motesnitsalis).

ACKNOWLEDGEMENTS

The authors want to thank the members of the TE-MPE-MS software team for their marvelous work and the members of the BE-CO-DA section for all their help and cooperation. Special thanks goes to L. Burdzanowski, C. Roderick and M. Sobieszek for their work on CALS (especially for the parallel data extraction feature) and V. Baggiolini for his continuous feedback and support.

REFERENCES

[1] K. Fuchsberger et al., "Automated Execution and Tracking of the LHC Commissioning Tests", proc. of IPAC12, New Orleans, LA, USA.

[2] D. Anderson et al., "The AccTesting Framework: An Extensible Framework for Accelerator Commissioning and Systematic Testing", THPPC078, proc. of ICALEPCS 2013.

[3] D. Andersen et al., "Using a Java Embedded DSL for LHC Test Analysis", THPPC079, proc. of ICALEPCS 2013.

[4] http://hadoop.apache.org

[5] http://akka.io

[6] E. Motesnitsalis, "Using Akka Platform in Unidentified Falling Object Detection on the LHC", CERN TE Notes, CERN, Geneva, 2013.

ISBN 978-3-95450-139-7