PLUGIN-BASED ANALYSIS FRAMEWORK FOR LHC POST-MORTEM ANALYSIS

R. Gorbonosov, G. Kruk, M. Zerlauth, V. Baggiolini, CERN, Geneva, Switzerland

Abstract

Plugin-based software architectures [1] are extensible, enforce modularity and allow several teams to work in parallel. But they have certain technical and organizational challenges, which we discuss in this paper.

We gained our experience when developing the Post-Mortem Analysis (PMA) system, which is a mission-critical system for the Large Hadron Collider (LHC). We used a plugin-based architecture with a general-purpose analysis engine, for which physicists and equipment experts code plugins containing the analysis algorithms. We have over 45 analysis plugins developed by a dozen of domain experts.

This paper focuses on the design challenges we faced in order to mitigate the risks of executing third-party code: assurance that even a badly written plugin doesn't perturb the work of the overall application; plugin execution control which allows to detect plugin misbehaviour and react; robust communication mechanism between plugins, diagnostics facilitation in case of plugin failure; testing of the plugins before integration into the application, etc.

INTRODUCTION

The Post-Mortem Analysis (PMA) is a mission-critical system for safe operation of the Large Hadron Collider (LHC). Its main goal is to perform an exhaustive analysis of the behaviour and state of the key LHC components (power converters, quench protection systems, interlock systems, collimators, beam-loss monitors, kickers and many others) in the event of a beam dump and decide if it is safe to continue operation. Detailed domain knowledge about the aforementioned components is necessary to perform the analysis. Because there is no single team possessing sufficient expertise about all the LHC components, we decided to delegate the coding of analysis algorithms to domain experts. In other words, domain experts write software components ("plugins") with analysis algorithms, and the core PMA team provides the general-purpose analysis engine to execute these plugins.

Requirements and Constraints

A plugin-based architecture enforces design and implementation decisions that both mitigate the risks of executing third-party code and simplify the implementation of plugins. In the case of the PMA this is absolutely vital since domain experts providing the analysis plugins are not professional programmers, and are therefore prone to make programming mistakes.

Another requirement which affected the design and implementation of the PMA is that to yield the overall result the plugins need to be executed in the right order and they need to communicate with each other, e.g. a subsequent plugin needs to be able to consume the output of previous plugins.

Workflow

The PMA workflow is shown in Fig. 1. Each box represents an analysis plugin. Typically (but not necessarily) the leftmost plugins focus on a single domain (power converters, collimators, etc.). The main purpose of these plugins is to filter out all the normal data since such data is not interesting for problem detection. The plugins in the middle represent cross-domain analysis. These plugins consume the results of single-domain analysis and perform data correlation in order to find discrepancies. At the right there are one or several plugins producing overall result(s) of the analysis.

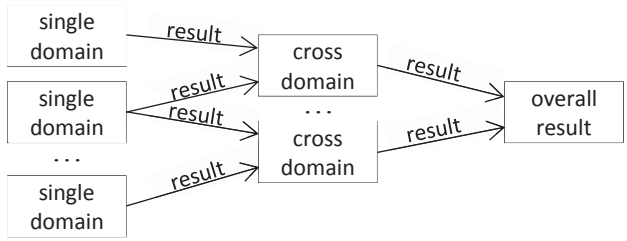

Figure 1: PMA workflow.

PROBLEMS, RISKS AND SOLUTIONS

This section describes problems and risks we faced as well as design and implementation solutions we have put in place to deal with those problems and risks. All the decisions are guided by 2 main principles:

1. lack of domain experts programming experience should not compromise overall system stability and reliability
2. implementation of analysis plugins should be made as simple as possible, domain experts should be able to focus on their business-logic only

Plugins Execution

As described above, the PM analysis plugins are executed in a well-defined sequence where each plugin waits for the relevant data to be ready before starting the execution.

A simplistic approach to implement this behaviour could be to simply link together the plugins using the observer (or any other notification) pattern. In this design, analysis plugins execute in a pretty autonomous manner. They notify each other once they produce data and each plugin decides itself when it has all the required data to start execution (Fig. 2a). Although this approach seems

ISBN 978-3-95450-139-7

quite natural at first sight, it violates both guiding principles. It violates the first one because if (due to a programming error) the first analysis plugin logic fails with an exception, and does not send any notification to the other plug-ins, those plugins do not start and the whole analysis execution gets stuck. It violates the second principle because in addition to writing their analysis code, the domain experts have to write code to keep track of incoming data and to send notifications.

In our PMA design, it is the framework that controls the analysis execution entirely (Fig. 2b). The framework triggers the execution of a plugin and monitors its progress. Once a plugin has finished executing, successfully or with exception, the framework takes over and calculates which plugin(s) should be triggered next, and so on. This guarantees the execution of all the analysis logic and simplifies the code of plugins.

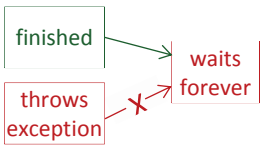

Figure 2a: Simplistic approach.

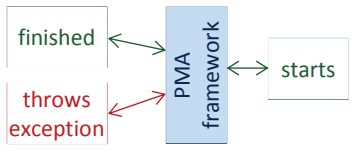

Figure 2b: PMA approach.

Plugins Misbehaviour

So far we have described how the PMA framework deals with a plugin that throws an exception. There are several ways a badly written plugin can fail: it can block, access resources or services (ex. file system, database, etc.) too often or even start producing an enormous number of result data (e.g. if it ends up executing an infinite loop).

A simplistic approach would let analysis code access the resources and services directly (Fig. 3a). Being the simplest first attempt such approach however violates the first guiding principle: if an analysis plugin is stuck in an infinite loop it will never finish, the dependent plugins will never be triggered and the whole analysis execution is compromised. If an analysis plugin overloads services used by other plug-ins it can potentially bring the services down, thus preventing other analysis logic from finishing successfully.

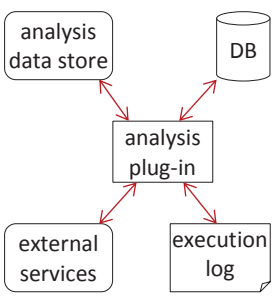

Figure 3a: Simplistic approach.

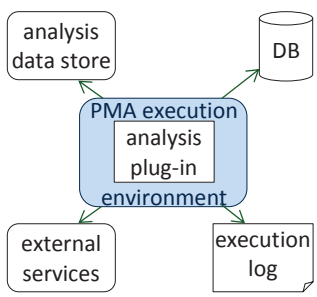

Figure 3b: PMA approach.

In the PMA framework, each analysis plugin is executed in a separate thread – this guarantees that a blocked plugin does not stop the overall analysis execution. Also, the framework does not give analysis plugins direct access to any resources or services (Fig. 3b). Instead plugins are executed in a special environment which provides access to resources and services via proxies. Those proxies can intercept calls of the plugin to external resources and allow the framework to abort the plugin if any misbehaviour is detected.

Inter-Plugin Communication

As mentioned, analysis plugins should be able to consume results of other plugins. In order to make such communication reliable, it is necessary to define a clear data contract between communicating plugins:

- the data container and data format to use: the consuming plugin should know how to fetch the necessary information from the incoming data container (ex. XML, JSON, collection of individual information pieces, hash maps, etc.)
- the data content: which information is expected and how it is represented (ex. strings, numbers, functions, etc.)

The first simplistic approach is to allow for a very loose contract, where each plugin just produces data in its own data format. While this is flexible for each data producer, it is very chaotic and redundant for the overall system: a consumer plugin has to use specific code for each plugin it receives data from.

A better approach is to define a standard data container shared by all plugins: this at least standardizes the way the information is accessed. The standard container should be flexible enough to accommodate all possible sizes and types of information (currents, beam-losses, images, etc.) produced by different plugins. It is clear that usually standardization and flexibility contradict each other. In PMA we've chosen maps with key-value pairs as data containers exchanged between plugins.

However, a standard flexible data container still gives no guarantee that the content of the data container is valid and complete. This can produce quite misleading results. For example, Fig. 4 shows two plugins communicating with each other. The plugin at the left has finished successfully but it has produced incomplete data. The data is consumed by the plugin at the right which fails because of this missing information. The problem is detected in the right-hand (consumer) plugin but in reality the root cause of the problem lies in the left-hand (producer) plugin. This example breaks the first guiding principle of PMA development mentioned above.

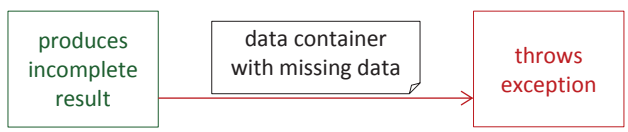

Figure 4: Exchange of incomplete data.

ISBN 978-3-95450-139-7

Another problem of using simple key-value maps appears while developing a consuming plugin: the developer has no clear idea which information is included in the data container and in which form. He has to talk to the other plugin developer (if possible) or examine the code of the producer plugin, which is cumbersome and error-prone. This complicates the task of the plugin developer a lot, thus breaking the second guiding principle.

In order to ensure a clear contract between plugins in PMA, map data containers are wrapped into a data-specific Java class known as a "Java bean" [2] (Fig. 5).

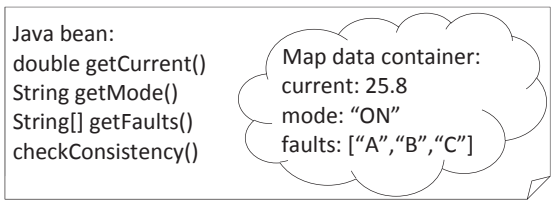

Figure 5: Java bean wrapper over map data container.

The Java beans capture the data contract: they have a getter/setter for every piece of information in the data container as well as generic logic checking the data correspondence to the contract: presence of all the information pieces and their correct representation (strings, numbers, arrays, etc.). With tools provided as part of the PM framework, the beans are generated automatically based on a sample data – so they don't require any routine coding.

With Java beans, the PMA framework can at runtime check the consistency of the produced data thus eliminate the confusion shown in Fig. 4. In addition, the developer of a consuming plugin gets the full power of compilation check and IDE code-completion.

Versioning of Data Contracts

Over time, the information produced by plugins can evolve, which means that the data contract has to evolve too. At the same time it must be still possible to process the data form the past. For example, we have the requirement to be able to re-analyse the beam dumps from several years ago, typically with improved analysis logic or after fixing bugs.

A simplistic approach to deal with such requirement is to force plugin developers to make their logic capable of dealing with different versions of data contracts. This approach however definitely makes the task of plugin developers more difficult thus violating the second guiding principle.

To simplify the work of plugin developers, the PMA framework uses only the latest data contracts. But the data producers are required to provide converters every time they change their data format. These converters transform old data into the latest data contract, thus making it again usable by the latest version of the PMA framework.

Plugin Testing

After a plugin is developed and unit-tested it is necessary to test it in a real runtime environment before it can be deployed into the production environment.

The first approach could be just to deploy it into a development version of the application and run some tests there. However, it is extremely difficult (especially for a non-professional programmer) to debug problems in a remotely running application. Also, every re-deployment, even to a development version of the application, usually requires an intervention of the PMA team, leading to an additional overhead.

To deal with this problem the PMA framework provides plugin developers with a test environment which should be used before deploying plugins into a development version of the application. The test environment simulates the real application and provides read-only access to all the resources and services available in the real application. It also allows using fake data to simulate different test-cases. The plugin developers can use the test environment locally profiting from all the debugging options provided by their IDE.

CONCLUSIONS AND OUTLOOK

The PMA framework has been used operationally for several years and proved to be very extensible, flexible and reliable. At CERN there are currently 4 mission-critical LHC applications based on the PMA framework: Global PMA [3], Injection Quality Check [4], External Post-Operational Check of LHC beam-dump system [5] and Powering Event Analysis. In total there are over 45 analysis plugins developed by a dozen of domain experts. Such a broad adoption would have never been possible without a plugin-oriented architecture and the design decisions described in this article.

A new area of work is to put in place a mechanism to hot-swap new versions of analysis modules without restarting the whole analysis application. Yet another functionality to work on is the implementation of automated tests in the real application for each module based on a set of incoming data representing possible use-cases.

REFERENCES

[1] http://en.wikipedia.org/wiki/Plugins
[2] http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html
[3] M. Zerlauth et al, "The LHC Post Mortem Analysis Framework", TUP021, ICALEPCS 2009, Kobe, Japan.
[4] L. N. Drosdal et al, "Automatic Injection Quality Checks for the LHC", WEPMU011, ICALECS 2011, Grenoble, France.
[5] N. Magnin et al, "External Post-Operational Checks for the LHC Beam Dumping System", WEPMU023, ICALECS 2011, Grenoble, France.

ISBN 978-3-95450-139-7