Self-managed Component-based Software Architecture for Business Process Management

Bassem Debbabi, Thomas Calmant, Olivier Gattaz
ISANDLATECH
Meylan, France
firstname.lastname@isandlatech.com

Sandra Massonnat, Patrick Emin
M1i
Cran Gevier, France
firstname.lastname@agilium.com

***Abstract*—While the functions of Business Process Management (BPM) tools are already studied and standardized, new challenges regarding the architecture of such type of tools are emerging including the need for more scalability to support increasing demands, and more resilience of the overall solution to detect and avoid third-party code problems, that can causes failure of all the system. In this paper we present the new architecture of Agilium BPM tool that benefits of self-managed distributed architecture provided by Cohorte framework. We discuss the issues of the old architecture and the benefits and results of the introduced autonomic architecture.**

I. Introduction

Business Process Management (BPM) is a system that manages the entire chains of events, activities and decisions that ultimately produce added value for an organization [1]. Agilium is a BPM tool that is currently developed and sold to industrial and commercial customers. It is used in different domains like retail, logistics, production management and/or order processing. Agilium is based on CRISTAL technology [2] outcome from the European Center for Nuclear Research (CERN) in Geneva. CRISTAL was developed for use in the construction of large-scale experiments such as the Complex Muon Solenoid at CERN's Large Hadron Collider (LHC).

Agilium BPM tool consists of one server that includes the CRISTAL kernel, to handle the user processes, and other data management tools and third-party connectors. It also includes a set of user interface applications: Agilium Web component, Agilium Supervisor GUI and Agilium Factory that is used for designing the user business processes. The server component of the product, which is the main part, is designed as one monolithic Java application. Besides the overall modular internal architecture of the server and its possibility to introduce new extensions representing user-specific connectors, the monolithic runtime representation leads to several difficulties like supporting incrementing loads (parallel connected users) and ensuring continuity of service when failures occurs in third-party code. Furthermore, identifying and correcting such failures can take a lot of time due to this monolithic view of the application at runtime.

In this paper, we present the new architecture of Agilium that uses self-managed distributed components handled by Cohorte framework[1] . Cohorte ensures the self-instantiation, self-distribution and self-configuration of the application components across different, separate and remote containers (or isolates). This ensures high availability of the system as new isolates can be created dynamically to support the increasing load, while ensuring autonomic recovery and isolation of faulty components to not affect the other components of the system.

[1]http://cohorte.github.io

II. Cohorte Framework

Cohorte Framework is based on OSGi[2] standard and provides a novel architecture for self-managing distributed components over several containers (isolates). It is based on three main concepts: (1) the instantiation and composition of application's components is handled automatically by a multi-level composers; (2) the handling of heterogeneous service-based component models including Apache Felix iPOJO [3] for Java and IsandlaTech iPOPO [4] for Python; and (3) a connectivity layer that abstracts the remote services between components. This three main concepts are represented in Figure 1 and detailed on the following sub-sections.

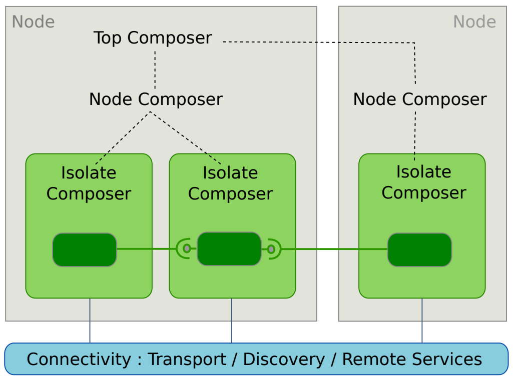

Fig. 1. Cohorte Framework

A. Multi-level Composers

Cohorte framework has different levels of composers which ensures the self-distribution and self-instantiation of components on the different isolates. A Top Composer distributes the components across the nodes; in each node, a Node Composer dynamically creates isolates to host the components; at the isolate level, Isolate Composer instantiates local components using specific component-models and monitors their state.

[2]http://osgi.org

B. *Isolates*

Cohorte Isolates are Service-Oriented containers hosting a set of components of the application. A Cohorte application can have a set of Isolates which are dynamically created by Cohorte Composers to deploy specific components or to react to a runtime failure.

C. *Connectivity and Remote Services*

Cohorte has a common messing and remote-services layer. It abstracts the distributed service calls between the different application components located on different isolates. Local services are published as remote services with no complecated configurations and proxy generation. All this technical, heavy tasks are handled transparently by the framework.

III. Agilium Implementation

A. *Old Architecture*

The Agilium server is a classic Java application that runs on one JVM. It has internal modules like the CRISTAL kernel, its client, a set of technical modules (TimeOut manager, HttpSdk), and third-party connectors to external systems (Figure 2).

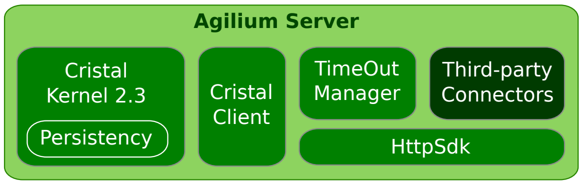

Fig. 2. Old Agilium Monolothic Architecture

This architecture has the following known runtime issues:

- **Libraries hell:** as developers, we had a serious complexity when deploying multiple versions of an internal libraries needed by other ones. The dependencies and classpaths should be handled rigorously. Actually, the server has 130 jar files.
- **Third-party code safety:** when third-party code crashes, all the server goes down. Developers should identify the source of the problem by analyzing the different log files while trying, as quickly as possible, to relaunch the server with new versions of the crashed modules.

Agilium contributors have identified several requirements to limit the complexity of the Agilium Server by (1) avoiding circular JAR dependencies; (2) protecting the other subsystems of the crash of one of them; and (3) allowing the starting of several instances of a sub-system to accept more load.

B. *Agilium's new Autonomic Architecture*

Following the Cohorte approach, the new Agilium (Agilium NG) has been built using several isolates for the different kind of its internal components.

The new architecture is composed of initial five isolates (Figure 3): *Server*, *Runner*, *Events*, *REST* and *Web UI*. At runtime, the Service-Oriented Components of the application are automatically dispatched on the different isolates by the multi-level composer according to the constraints defined in the composition specification.

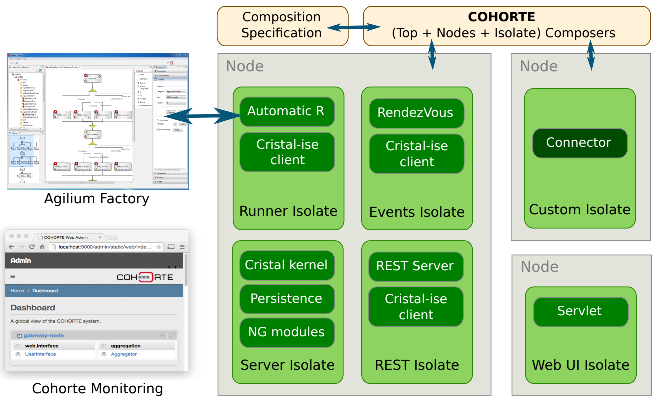

Fig. 3. New Agilium Autonomic Architecture

The main benefits using this autonomic architecture are resumed as follows:

- **Global crash prevention:** the failure of a third-party component deployed in a custom isolate has no impact on the availability of the Agilium application.
- **Load adaptation:** A new instance of an automatic runner component can be deployed in a new remote isolate to accept dynamically an increase of the load.
- **Easy enhancement and deployment:** the distribution of components in many isolates is an effective way to brake the limits of the management of the libraries dependencies (cf. The java class-loading hell).

IV. Conclusion and Future Work

From the Agilium new requirements study and our experience doing this initial work implementation, we can conclude that a self-management distributed components solution provides major benefits to software systems such as BPM tools. In addition to simplifying the development of such system's components as all the instantiation, deployment and configuration tasks are handled by the autonomic system, such architectural solution provides self-healing features that detects, isolates, and by then helps developers to replace the faulty components while ensuring a continuity of application's service. As future work, we are investigating the use of DevOps techniques to continuously and autonomously deploy new components on the different isolates or rolling-back to old versions if failure occurs to ensure continuity of components service.

References

[1] DUMAS, Marlon, LA ROSA, Marcello, MENDLING, Jan, et al. *Fundamentals of business process management*. Berlin : Springer, 2013.

[2] J. Shamdasani, et al. *CRISTAL-iSE - Provenance Applied in Industry*. ICEIS (3)) 2014: 453-458

[3] Escoffier, C., et al. *iPOJO: An extensible service-oriented component framework*. SCC 2007 IEEE.

[4] Thomas Calmant, et al. *A dynamic and service-oriented component model for python long-lived applications*. CBSE 2012.