Towards Robustness and Self-Organization of ESB-based Solutions using Service Life-cycle Management

Paulo Leitão[1,2], José Barbosa[1], Arnaldo Pereira[1]

[1] Polytechnic Institute of Bragança, Campus Sta Apolónia, Apartado 1134, 5301-857 Bragança, Portugal
{pleitao, jbarbosa, arnaldo}@ipb.pt
[2] Artificial Intelligence and Computer Science Laboratory, Rua Campo Alegre 102, 4169-007 Porto, Portugal

Abstract- **Enterprise Service Bus (ESB) is a middleware infra-structure that provides a way to integrate loosely-coupled heterogeneous software applications based on the services principles. The life-cycle management of services in such environments is a critical issue for the component's reuse, maintenance and operation. This paper introduces a service life-cycle management module that extends the traditional functionalities with advanced monitoring and data analytics to contribute for the robustness, reliability and self-organization of networks of clusters based on ESB platforms. The realization of this module was embedded in the JBoss ESB, considering a sniffer mechanism to collect the service messages crossing the bus and a Liferay portal to display relevant information related to the services' health.**

I. INTRODUCTION

The conceptualization of Internet of Things paradigm and the implementation of computational distributed systems reinforce the importance of the integration of heterogeneous software applications across the enterprise IT infrastructures. In fact, according to a prediction report from Gartner on Application Integration [1], by 2016, midsize to large companies will spend 33% more on application integration than in 2013, and by 2018, more than 50% of the cost of implementing 90% of new large systems will be spent on integration.

The advent of service-oriented architecture (SOA) [2] as a software paradigm for distributed systems to integrate the enterprise IT infrastructures brought the concept of Enterprise Service Bus (ESB). An ESB is a software architecture model used for designing and implementing the communication between interacting software applications in a SOA environment. It is based on the idea to have a standard and structured middleware that offers a way to connect and integrate loosely-coupled heterogeneous software components, named services, reducing the complexity of application interfaces. In 2003, Gartner Inc. predicted that the majority of large enterprises will have an ESB running to integrate their IT infrastructures by the end of 2006 [3]. However this prediction has proved to be too much optimistic, since nowadays heterogeneous systems may also use web services interfaces. Several ESB products are available, in the form of commercial and open source products, namely Oracle Service Bus, Mule ESB, Fuse ESB, Talent ESB and JBoss ESB.

The ESB solutions provide a distributed, modular and pluggable architecture, supporting intelligent and dynamic routing and mediation of services' discovery, request and execution. The main benefits of using an ESB platform is the increased flexibility and scalability, the interoperability transparency and the existence of configuration rather integration coding. In fact, large-scale distributed systems can benefit from an ESB middleware acting as a broker between the numerous heterogeneous service providers/requesters, avoiding a potentially huge number of point to point connections. However, the increased overhead and the possible slower communication speed are the main disadvantages.

One of the main functionalities of an ESB is to monitor and control the routing of messages exchanged between services. The life-cycle management of deployed services, e.g. including functions of monitoring and data analytics, is a crucial issue for the component reusing, maintenance and monitoring [4], and in the context of ESBs, contributes to increase the system robustness, reliability and fault-tolerance. In fact, the possibility to monitor the performance of registered services and analyse the evolution of its behaviour, allows to detect in advance possible degradation or risk propagation, being possible to generate warnings to implement proper corrective actions to mitigate the problem.

Currently, and related to the life-cycle management, the ESB platforms only provide basic functions associated to the service registry and completely misses these kind of advanced functionalities, leading to the need to have a life-cycle management functionality embedded in the ESB that provides monitoring and data analytics of the registered services. For this purpose, the objective of this work is to develop a life-cycle management module that will be embedded in the traditional ESBs to provide monitoring and data analytics of the registered and deployed services, contributing to achieve more robust, reliable and fault-tolerant distributed SOA-based systems. Additionally, this functionality will also play an important role in the self-organization of the network of software applications organized as clusters of ESBs, in a dynamic and on-the-fly manner.

The rest of the paper is organized as follows: Section II discusses the architecture of the intelligent enterprise service bus (iESB) developed under the ARUM project, and Section III presents the specification of the life-cycle management module as part of the iESB. Section IV discusses how this module contributes to achieve robustness, reliability and self-organization and Section V presents the technical details related to its implementation. At last, Section VI rounds up the paper with the conclusions.

II. Intelligent Enterprise Service Bus

The ARUM (Adaptive Production Management) project [5] addresses the development of novel Information Communication Technology (ICT) solutions to handle new challenges in production and ramp-up of complex and highly customized products, such as aircraft and shipbuilding industries. The focus is on the development of mitigation strategies to respond faster to unexpected events and the implementation of systems and tools for the decision support in planning and operation. For this purpose, the ARUM platform comprises an intelligent ESB, which enriches the traditional ESBs with a plethora of advanced modules, and provides a common infrastructure for the integration of heterogeneous planning and scheduling tools (e.g. using the MAS - Multi-agent Systems principles) and legacy systems, as illustrated in Fig. 1. The main modules deployed in the iESB are the Ontology service, Data Transformation Service, Sniffer, Node Management and Life-Cycle Management.

The Ontology service module is responsible to gather the pieces of data from various legacy systems, e.g. MES (Manufacturing Execution System) and ERP (Enterprise Resource Planning), via the data transformation service, aggregate and store it in the local triple store and then provide it on request to other services. Aiming to provide a common and explicitly defined semantics of data, it was developed a set of OWL (Web Ontology Language)-based ontologies for the description of production processes, shop floor topologies, resources and their availability, scheduling strategies, disruption events, etc. [6].

The Data transformation service module is responsible for gathering data from legacy systems. The raw data, received from gateways using the legacy system specific interfaces and communication protocols, is transformed into the ontological format (RDF - Resource Description Framework) using the OWL-based ontologies provided in the Ontology service.

The Sniffer module is responsible for capturing the flow of messages across the ESB and related to the registered services, to support the monitoring and understanding of the overall state of the system especially in a distributed environment with multiple interacting services [7].

The Node Management module supports the distributed management of iESB instances, allowing the inter-connection among several ESBs.

The Life-cycle management module performs the life-cycle monitoring and analysis of the health of the services that are deployed within the iESB, supporting the dynamic, online and on-the-fly actions to mitigate the degradation of their performance. This module will be deeply analysed during the rest of this paper.

The Dashboard acts as a user interface (UI), providing the user with the means for administration and monitoring of the overall ARUM solution (including all deployed tools). It means for example the deployment of services, monitoring their parameters and health, visualizing the message flow and statistics. The dashboard leverages the web portal technology, which is a specially designed web page on which the information is displayed within dedicated user interface components – the portlets.

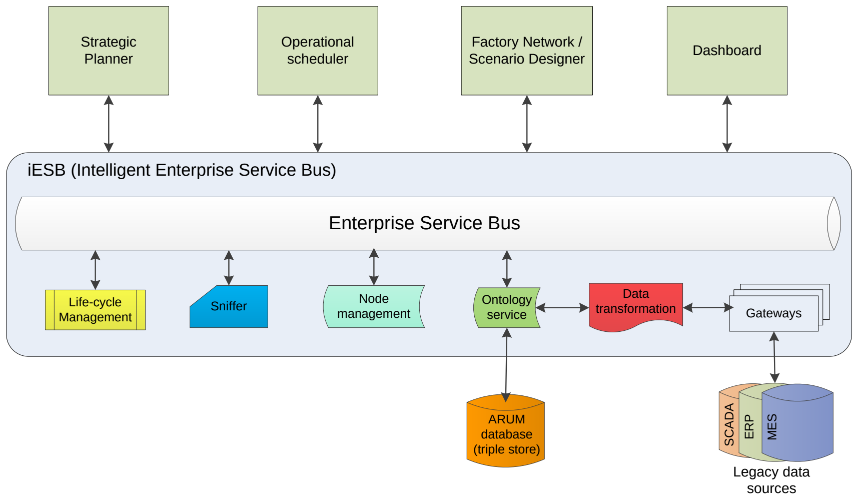


Fig. 1. ARUM platform high level architecture.

III. Life-cycle Management Module

The Life-cycle Management Module (LCMM) performs the continuous monitoring and data analytics of the services that are deployed within the iESB, allowing to dynamically be aware of the current state and health of the services and to perform on-the-fly actions to increase the services' performance. In particular, the main features provided by the LCMM module are:

- Monitoring of the registered services' health, providing on-line information related to different KPI (Key Performance Indicators), such as the response time, the failure rate and the occupancy.
- Detection of the registered services/tools that are not operating properly and analysis of trend and patterns on the services' performance, e.g. the detection of the degradation in the service quality.
- Analysis of the risk propagation in case of service quality degradation.
- Suggestion of actions to maintain the system's robustness and stability.

The LCMM module interacts with the Sniffer module to get data related to the exchanged messages and the UI Dashboard to support the interaction with the user and particularly to display the monitored info related to the health of registered services according to pre-defined KPIs, as illustrated in Fig. 2. Internally, the module comprises the Event Monitoring, Data Analysis and local database.

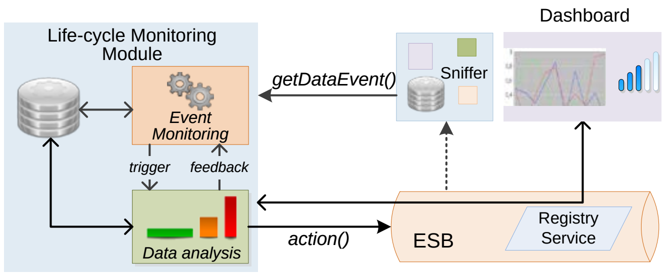


Fig. 2. Architecture of the life-cycle management module.

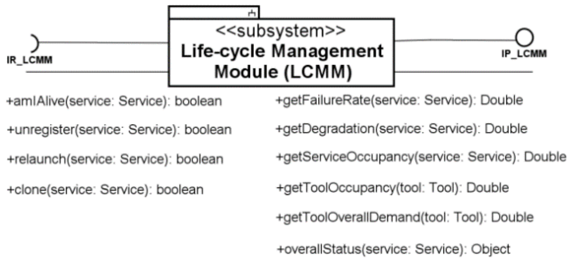
The interaction between the Event Monitoring and Data Analysis components allows to trigger a more detailed data analytics and also to provide feedback regarding the adjustment of the pooling rate for a specific service.

A. *Event Monitoring Component*

The Event Monitoring component performs mainly the collection of the data related to the exchanged messages across the bus and the monitoring of the services' health. Since the Sniffer module is continuously sniffing the messages crossing the ESB and feeding its database with the gathered information, the Event Monitoring component can request this data using a proper and dynamic polling mechanism that is parameterized according to the service frequency and priority. In fact, the polling time is adjusted according to the service usage frequency, i.e. short polling time if the service is usually requested or larger time if rarely requested. Also, an event-driven mechanism can be used to collect the data from the Sniffer module, but this alternative can only be used if the Sniffer module provides the subscription functionally.

The reasoning engine, embedded in this component, processes the gathered and historical information in order to support the health monitoring of registered services by calculating several pre-defined KPIs, namely in terms of performance and status, that will be exposed as monitoring services to the user, namely through the UI dashboard. Examples are the detection if the registered services are not alive by identifying not answered messages and behaviours that not follows the service patterns.

Considering that $T = \{tool_i : i = 1, \ldots, M\}$ is the set of tools connected to the ESB and each tool offers a set of services $S_i = \{service_{ij} : j = 1, \ldots N_i\}$, the LCMM module provides a plethora of services aiming to monitor several KPIs, as detailed as follows (also illustrated in Fig. 3):


<<subsystem>>
Life-cycle Management Module (LCMM)
IR_LCMM
IP_LCMM
+amIAlive(service: Service): boolean
+unregister(service: Service): boolean
+relaunch(service: Service): boolean
+clone(service: Service): boolean
+getFailureRate(service: Service): Double
+getDegradation(service: Service): Double
+getServiceOccupancy(service: Service): Double
+getToolOccupancy(tool: Tool): Double
+getToolOverallDemand(tool: Tool): Double
+overallStatus(service: Service): Object
Fig. 3. Services provided by LCMM and also the requested services.

- *getFailureRate*: provides the failure rate of a service, calculated as follows, where f_{ij} is the number of failures of the $service_{ij}$ with reference to the last n requests of this service:

$$FR_{ij} = \frac{f_{ij}}{n} \qquad (1)$$

- *getDegradation*: provides the information related to the degradation of the response time of a service j of the tool i (δ_{ij}). The degradation is the comparison of the response time of the last two events.

$$D_{ij} = \frac{\delta_{ijt} - \delta_{ij(t-1)}}{\delta_{ij(t-1)}} \qquad (2)$$

- *getServiceOccupancy*: provides the information related to the occupancy of a service. The Service Occupancy (SO_{ij}) of a service j running in the tool i is defined as the ratio of the overall time t_{ij} that the service is being used with the overall time Δ_i of the software tool deployed on the system:

$$SO_{ij} = \frac{t_{ij}}{\Delta_i} \qquad (3)$$

- *getToolOccupancy*: provides the information related to the occupancy of a tool. The Tool Occupancy (TO_i) is defined has the ratio of the time t_i that a given tool i is being used (independent of the overlapping of services in the tool) with the overall time Δ_i of the tool deployment on the system, as illustrated as follows:

$$TO_i = \frac{t_i}{\Delta_i} \tag{4}$$

- *getToolOverallDemand*: provides the information related to the load of a tool within the overall ESB load. This load is the ratio of the number r_i of requests to the services running in tool i and the total number of requests to all tools, represented as follows:

$$TOD_i = \frac{r_i}{\sum_{k=1}^{M} r_k} \tag{5}$$

- *overallStatus*: provides the overall service status considering all evaluation parameters, namely the failure, degradation and occupancy, weighted according to pre-defined values. This will be defined as a health scale, where 0 means "good", 1 means a potential "risk" or "problem".

This component can also implement a pre risk analysis allowing to determine potential situations of service/tool failure. In this way, when a set of conditions are met, such as the presence of historical problematic tools or the warnings coming from the evolution of service KPIs, the component can signalize the critical service(s) and take more pro-active measures, such as changing the warning threshold values. Beside this action enabling the early signalling of potential hazardous situations, it additionally allows an anticipated taking of known actions that permit to overcome the potential situation.

B. Data Analysis Component

The Data Analysis component aims to perform advanced reasoning, and particularly data analytics, over the historical and current collected information related to the deployed services. In fact, the functions provided by this component include:

- Analysis of trends to detect deviations or patterns in the quality and performance of the service.
- Analysis of correlation among the execution of different services (also including the correlation considering services deployed in other ESBs belonging to the same network).
- Analysis of the impact and risk propagation related to the degradation of a service.

The implementation of these functionalities may consider the use of data mining techniques [8], namely clustering. Clustering is a technic used to find, in an automatized way, hidden patterns in big quantities of data. Based on the *k*-means clustering algorithm presented in [9], Fig. 4 illustrates a strategy integrated in LCMM to perform data analytics to discover the set of services that presents more risk of abnormal behaviour.

A periodic or trigger event causes the Data Analysis component to start reading the values collected by the Event Monitoring component in local database. The algorithm prescribes choosing *k* arbitrary readings as the cluster centres. All readings are assigned to the most similar centre based on the calculation of Euclidean distance between the read values. A next step is related to the calculation of the most central point for each cluster (not necessarily equal to the initial choice for the centre). Then, it is performed a new reassigning of all values in conformity with the new centres. Finally, the services in the cluster with the worst centre are signalized as services with possible degraded performance, and in this way will require a close monitoring.

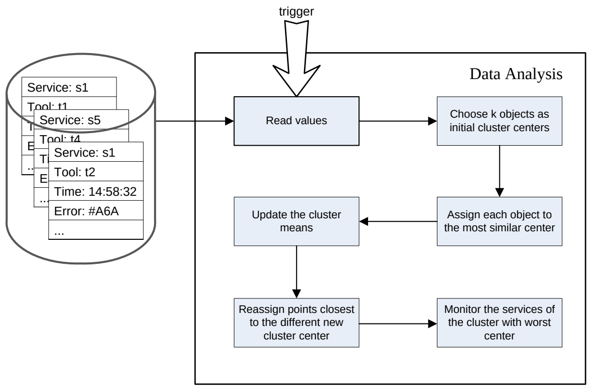

trigger
Data Analysis
Service: s1
Tool: t1
Service: s5
Tool: t4
Service: s1
Tool: t2
Time: 14:58:32
Error: #A6A
...
Read values
Choose k objects as initial cluster centers
Assign each object to the most similar center
Update the cluster means
Reassign points closest to the different new cluster center
Monitor the services of the cluster with worst center

Fig. 4. Algorithm for detecting patterns in the data analysis.

The output of these functions is the generation of warnings to the user, e.g. providing useful information about the state and risk of a specific service and also suggesting the execution of proper actions, such as unregistering the service (e.g. when the service is not being used), re-starting of a tool (e.g. when the service is degraded or not responding) or creating one clone (e.g., when the service/tool is too busy). These actions can also be performed automatically, under well controlled conditions. In this case, as also illustrated in Fig. 3, and aiming to support the operation of the LCMM module, several services provided by other modules in the iESB may be requested, namely *amIAlive* (to verify if the service is alive), *unregister* (to unregister services by accessing the ESB Register Service), *relaunch* (to restart the service provided by the tool) and *clone* (to clone a service/tool, e.g. when a service/tool is too busy). Note that the use of these services may require some kind of privilege access to external tools.

Learning is an important piece of the LCMM module, supporting the discovery in advance of potential problems and the definition of the actions to be implemented when a risk is detected (as well as in the adaptation of the warning threshold values). The LCMM module should also consider self-monitoring and self-analysis in order to avoid its chaotic behaviour, e.g. acting as a "cancer" deploying very rapidly services/tools and consequently overloading the system.

IV. Robustness and Self-Organization Provided by LCMM Module

The implementation of the LCMM functionalities is a step forward to achieve intelligence in the ESB platform and in this way to achieve an iESB. More concretely, this module may contribute to achieve robustness, reliability, fault-tolerance and self-organization in this kind of distributed systems, i.e. those based on the ESB middleware.

Robustness can be defined as the capability of a control system to remain working correctly and relatively stable, even in presence of disturbances. Additionally, an important issue is the system fault-tolerance, i.e. the capability to detect and tolerate internal failures, in order to continue performing their operations without the need for an immediate intervention. Being more tolerant, the downtime is reduced, and being able to detect and diagnosis, the repair process is speed-up, increasing the robustness and productivity of manufacturing systems [10].

In such kind of distributed systems, based on offering and requesting services, the inexistence of central nodes makes these systems more robust than the traditional centralized systems, by eliminating the single point of failure problem. In fact, more decentralization provides additional reliability due to the implicit redundancy and diversity and the non-dependency of central control nodes [11]. However, the existence of a middleware infra-structure to integrate the IT software applications based on services can somehow restrict the robustness and reliability of such systems. Note that reliability is the ability of a system or component to perform its required functions under stated conditions for a specified period of time. In this way, the LCMM module ensures the increase of robustness and reliability by permitting an automatic discovery of problematic services, e.g. the ones that may be failing, not responding properly or overloaded, and take/suggest appropriate actions, such as launching a parallel service of the one that is identified to be near of failure, to mitigate the possible problems.

Additionally, the LCCM module can greatly contribute as an underlying mechanism to support self-organization at two levels: at service level or at ESB level. On the first case, the LCCM module can act as a referee, issuing warning signals for the deployed services, preventing erratic behaviour (e.g. when a tool is sending over the limit service requests). In this case, and if the appropriate behaviour actions are implemented in the affected tool, the tool can change its internal behaviour accordingly. A second example can be found in a tool that has reduced utilization. In this case, and if a redundant tool is present, the LCMM can advise the less used tool to change into low profile mode or to, at the limit, un-plug itself from the system, as seen in Fig. 5 (hexagonal service).

At the ESB level, the information mined by the LCMM can be used by the self-organization mechanism as the way to internally (re)arrange structurally the ESB, by adding, modifying or removing services (ellipse service in Fig. 5), or (re)arranging the relations and constitution of clusters in an inter-ESB perspective. This structural self-organization level allows the dynamic clustering of ESBs, arranging themselves accordingly, aiming a uniform service performance distribution where the performance of each individual ESB is increased by the decrease of individual service/tool overload and failure rate.

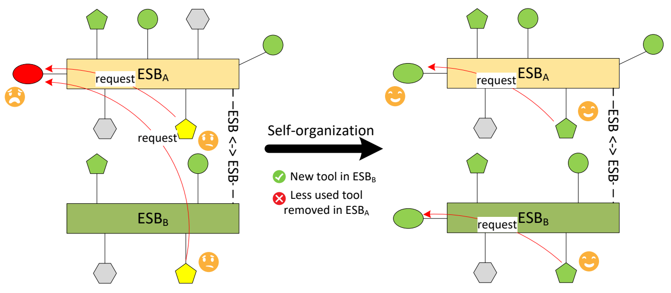


Fig. 5. Structural self-organization in a network of ESB clusters.

The aforementioned insights are drawn from the ADACOR[2] control architecture. In the proposed architecture, the individual behaviour of the entities [12] are dynamically changed, aiming a smooth evolution of the system, while a more drastic evolution is achieved through the change of the entities relations [13]. Similarly to what is achieved in the ADACOR[2] approach, by combining these two self-organization levels, the LCMM module will enable the achievement of a self-organized and evolvable ESB system, once the overall system is able to adapt itself internally and structurally to system demand fluctuations, internal services disruption or to ESB node change.

Additionally, and also as indicated in the ADACOR[2] control architecture, the LCMM must undergo with a nervousness controller in order to avoid entering in a chaotic process when taking decisions. This stabilization mechanism will prevent intermittent service/tool stop or launch as also the constant (re)arrangement of the ESB clustering.

V. Implementation and Operation of LCMM Module

The proposed LCMM was developed and deployed as a JBoss ESB service, encapsulating its business logic into a set of Java classes. JBoss ESB [14] is an ESB solution maintained under the umbrella of JBoss Community [15] and intends to provide an open source option for the construction of systems based on SOA principles.

The main constitutive part of the LCMM service is a chain of “Actions”. Basically, in the JBoss ESB framework, an “Action” is a Java class that allows the ESB services to carry out their tasks. These tasks are realized after the processing of the data referring to the exchange of messages between the registered services in the ESB. To accomplish that, it was implemented a connection to the Sniffer’s database, which is implemented using a MySQL database.

The user interface, supporting the visualization of the data resulting from the processing operation of the LCMM service, was developed as a web-based application that can be accessed via web browser. This web-based application was built on the Liferay Portal [16] [17] as a portlet. One of the central pieces used to construct the portlet was the Highcharts 3.0 [18], which is a charting library written in HTML5 and JavaScript, allowing, among others, to build dynamic charts.

The communication between the LCMM service and the

Liferay portlet is achieved by using the Java Message Service (JMS). The JMS specification describes the exchange of messages between Java programs, in particular for the use in publish-subscribe solutions. Going into details, it is used a JMS topic, allowing the delivery of messages to multiple subscribers.

Fig. 6 illustrates a screenshot showing the evolution of two KPIs related to the evolution of two services, namely the failure rate and the degradation.

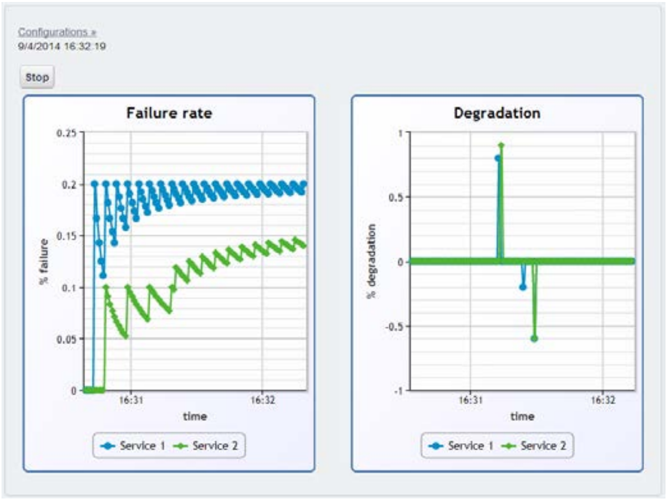


Fig. 6. Screenshot of the LCMM user interface

As showed in the screenshot, the failure rate of Service 1 converges to 20% stabilizing around this value. Observing the evolution of the failure rate curve of Service 2, it is possible to verify that, between 16:31 and 16:32, the failure rate is increased from 10% to 15%. In parallel (seeing the chart on the right), it is possible to observe a degradation of the response time of the Service 1 and Service 2 after 16:31, which may be explained by a peak on the demand using the bus. However, both services recover after a while as shown by the negative values in the chart. The continuous monitoring of these KPIs allows to detect these problems and to trigger warnings for the implementation of proper actions that will mitigate their negative impact.

VI. Conclusions

The use of ESB middleware allows to implement distributed systems that integrate loosely-coupled heterogeneous IT infra-structures. ESB provides several functionalities, namely the monitor and control of the routing of messages exchanged between software applications that expose their functionalities using services. Related to the life-cycle management of services, the ESB usually only provides basic functions associated to the service registry and completely misses advanced functionalities regarding e.g. data analytics.

This paper describes a service life-cycle management module that is embedded in the traditional ESB to provide advanced monitoring capabilities and data analytics to the registered services, contributing to achieve more robust, reliable and self-organized SOA-based systems.

The proposed module was implemented as a JBoss ESB service, using Java, and the user interface was developed as a web-based application built on the Liferay Portal. Several functions were implemented allowing to monitor the health of services according to pre-defined KPIs and also to detect trend and patterns in the service performance. The experimental implementation allowed a proof the concept and the future work is related to the implementation of data mining techniques supporting the data analytics, and also its installation in real IT infra-structures.

Acknowledgment

The research leading to these results has received funding from the European Union Seventh Framework Programme FP7 ARUM project, under grant agreement n° 314056.

References

[1] Gartner Inc., “Predicts 2013: Application Integration”, 14 November, 2012.

[2] T. Erl, *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*, Prentice-Hall, 2005.

[3] Gartner, Inc., “Predicts 2003: Enterprise Service Buses Emerge”, 9 December, 2002.

[4] B. Wang, X. Zhou, G. Yang, Y. Lou, “Service Lifecycle Management in Distributed JBI Environment”, Web Information Systems and Mining, Lecture Notes in Computer Science Volume 7529, pp 431-438, 2012.

[5] ARUM - Adaptive Production Management, http://www.arum-project.eu/ (accessed on 8th April 2014).

[6] U. Inden, N. Mehandjiev, L. Mönch, P. Vrba, “ Towards an Ontology for Small Series Production”, Mařík, V., Martinez Lastra, J. L., Skobelev P. (eds): Industrial Applications of Holonic and Multi-Agent Systems, Springer Verlag Berlin-Heidelberg, LNCS 8062, pp. 128-139, 2013.

[7] P. Vrba, P. Kadera, M. Myslík, M. Klíma, “JBoss ESB Sniffer - Message Flow Visualization for Enterprise Service Bus”, Proceedings of the IEEE International Symposium on Industrial Electronics (ISIE’14), 2014.

[8] H. Witten, Eibe Frank, and Mark A. Hall, *Data Mining: Practical Machine Learning Tools and Techniques* (3rd ed.), Morgan Kaufmann Publishers, San Francisco, 2011.

[9] T. Kanungo, D.M. Mount, N.S. Netanyahu, C.D. Piatko, R. Silverman, A.Y. Wu, “An Efficient k-means Clustering Algorithm: Analysis and Implementation”, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol.24, n.7, pp. 881-892, 2002.

[10] P. Leitão, “A Holonic Disturbance Management Architecture for Flexible Manufacturing Systems”, International Journal of Production Research, vol. 49, n.5, pp 1269-1284, 2011.

[11] A. Pereira, N. Rodrigues, J. Barbosa, P. Leitão, “Trust and Risk Management Towards Resilient Large-scale Cyber-Physical Systems”, Proceedings of the IEEE International Symposium on Industrial Electronics (ISIE’13), May 28-31, Taipei, Taiwan, 2013.

[12] J. Barbosa, P. Leitão, E. Adam, D. Trentesaux, “Self-Organized Holonic Multi-agent Manufacturing System: The Behavioural Perspective”, Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics (SMC’13), pp.3829-3834, 2013.

[13] J. Barbosa, P. Leitão, E. Adam, D. Trentesaux, “Structural Self-organized Holonic Multi-Agent Manufacturing Systems”, Industrial Applications of Holonic and Multi-Agent Systems (HoloMAS’13), Lecture Notes in Computer Science, vol. 8062, Springer pp. 59-70, 2013.

[14] L. DiMaggio, K. Conner, M.B. Kumar and T. Cunningham, “*JBoss ESB Beginner's Guide”*, Packt Publishing, 2012.

[15] JBoss Community, https://www.jboss.org/ (accessed on 8th April 2014).

[16] P. Sarang, *Practical Liferay: Java-based Portal Applications Development*, Apress, 2009.

[17] Liferay, http://www.liferay.com/ (accessed on 8th April 2014).

[18] Highcharts JS, http://www.highcharts.com/ (accessed on 8th April 2014).