ARMISCOM: Autonomic Reflective MIddleware for management Service COMposition

Juan Vizcarrondo[1], José Aguilar[2], Ernesto Exposito[3, 4] and Audine Subias[3,4]

***Abstract*—**In services composition the failure of a single service generates an error propagation in the others services involved, and therefore the failure of the system. Such failures often cannot be detected and corrected locally (single service), so it is necessary to develop architectures to enable diagnosis and correction of faults, both at individual (service) as global (composition levels). The middlewares, and particularly reflective middlewares, have been used as a powerful tool to cope with inherent heterogeneous nature of distributed systems in order to give them greater adaptability capacities. In this paper we propose a middleware architecture for the diagnosis of fully distributed service compositions called ARMISCOM, which is not coordinated by any global diagnoser. The diagnosis of faults is performed through the interaction of the diagnoser present in each service composition, and the repair strategies are developed through consensus of each repairer distributed equally in each service composition.

***Keywords*—** Reflective middleware, Web service composition, Web service fault tolerance, Autonomic Computing Architecture.

I. Introduction

SOA (Service Oriented Architecture) is a software development model in which an application is broken down into small units, logical or functional, called services. SOA allows the deployment of distributed applications very flexible, with loose coupling among software components, which operate in heterogeneous distributed environments [15] [17]. Examples of software components to be integrated are the web services. In general, web services are computational entities, which are autonomous and platform-independent and that can be composed with others in order to offer composite services [16].

The services are inherently dynamic and cannot be assumed to be always stable [1], because during the natural evolution of a service (changes in its interfaces, misbehavior during its operation, among others) can alter the resulting service. Additionally, in the case of service composition the failure of a single service leads to error propagation in the others services involved, and therefore the failure of the system. Such failures often cannot be detected and corrected locally (single service), thus it is necessary to develop architectures for enable diagnosis and correction of faults, both at individual (service) as global (composition) levels.

In this paper we propose an architecture for the diagnosis of fully distributed service compositions, which is not coordinated by any central component, in which the diagnosis of faults is performed through the interaction of the diagnoser present in each service, and the repair strategies are developed through consensus of each repairer that are equally distributed in the composition. Our architecture is based on the Autonomic Computing approach in order to facilitate its construction based on methods, algorithms, and tools that allow the developing of self-healing systems.

II. Related Works

Failures in web services can be classified at the level of the service itself and/or the sequence of calls in a composition of these services. Thus, in [2] it is proposed a taxonomy for the analysis of possible failures and perceived effects at both local (service) and composition levels, representing an excellent starting point for this work. In addition, a first attempt is made to correlate the failures and possible mechanisms to be implemented to solve them.

On the other hand, several architectures have been proposed for fault management and recovery in the web service composition. In [3] a reflective middleware called SOAR is defined for fault management in service composition, which is designed as a global structure (centralized) to monitor and adapt the complete system. The middleware is composed of two levels: the first (base level) is responsible for describing the basic characteristics of a SOA system, and the second level (meta level) is responsible for monitoring and adapting the SOA system. The reflection of the middleware is performed by making use of dynamic binding of web services composition, when connecting or disconnecting the services forming part of one joint task.

A second proposal is a centralized architecture for web services reparation [4], called self-healing architecture. It carries out the SOA system reparation using quality measurements from web services. The architecture consists of three modules: the monitoring and measurement module (it is

[1] Centro Nacional de Desarrollo e Investigación en Tecnologías Libres (CENDITEL), Mérida, Venezuela, jvizcarrondo@cenditel.gob.ve
[2] CEMSID, Dpto. de Computación, Facultad de Ingeniería, Universidad de Los Andes, La Hechicera, Mérida - Venezuela, aguilar@ula.ve
[3] CNRS, LAAS, 7, avenue du Colonel Roche, F-31400 Toulouse, France
[4] Univ de Toulouse, INSA, LAAS, F-31400 Toulouse, France ernesto.exposito@laas.fr, subias@laas.fr

responsible for observing and keeping records of QoS (Quality of service parameters that are relevant), the diagnosis and decision strategies engine (it detects degradation of the system and identifies reparation plans), and finally, the reconfiguration module (it implements the reparation plan). Also, in [11] a centralized architecture based on QoS monitoring is proposed.

In [5] it is proposed a decentralized architecture composed by 2 levels. The first level uses a local diagnoser for each service that is part of the composition, which communicates with a global diagnoser (central) for the diagnosis of the entire composition. The global diagnoser is responsible for the coordination of the local diagnoser making use of the exchange of messages to find the service and the activity responsible for the failure. Furthermore, this global entity is also able to implement mechanisms for the composition recovery. Each local diagnoser instances chronicles which describe failure patterns defined previously (off-line), that are propagated to the global diagnoser. It makes a calculation about the sequence of events in the services to find the occurrence of an error according to the chronicles instanced by the local diagnosers. Furthermore, in [12] it is proposed a structure composed of local diagnosers, which are coordinated by a global diagnoser that executes the repair tasks.

III. ARMISCOM Architecture Bases

ARMISCOM (Autonomic Reflective MIddleware for mangement Service COMposition) is a Reflective Middleware Architecture. The reflection is the ability of ARMISCOM to monitor and change its own behavior, as well as aspects of its implementation (syntax, semantic, etc.), allowing the ability to be sensitive to its environment. In this way, ARMISCOM has a dynamic behavior and is an adaptive architecture i.e. it can manage programs that are able to dynamically change or evolve. In general, it has two processes [10]:

- **Introspection:** Its ability to observe and reason about its own execution state.
- **Intersection:** Its ability to modify its own execution state, or alter its own interpretation or meaning.

In this proposal, our reflective middleware will be fully distributed through all services of the system, in order to have a closer view of the occurrence of events that happen in the application. ARMISCOM is divided in the classical two levels of all reflective middleware: the base and the meta level (see Figure 1):

Base Level: A service composition can be seen as a set of calculations and interactions of the services that compose a SOA application, and the set of rules and definitions that govern those interactions (SOA System). The base level of the middleware needs to know the interactions that occur in the choreography and the definitions and rules that govern these interactions.

Meta Level: This is the part of the middleware that provides the capacity for reflection. The base-level introspection is done by analyzing the message exchange among the services that are part of the composition and the components of the SOA system. To achieve an adequate level of introspection the meta level must observe both the SOA system and the SOA application. To achieve this, standard specifications for web services and composite services are used: WSDL[1], UDDI[2], OWL-S[3] and SCA[4]). In addition, the dynamic reconfiguration solution is provided by FraSCAti[5] for the intersection process of the services choreography platform. The meta level is instantiated for each service of the choreography.

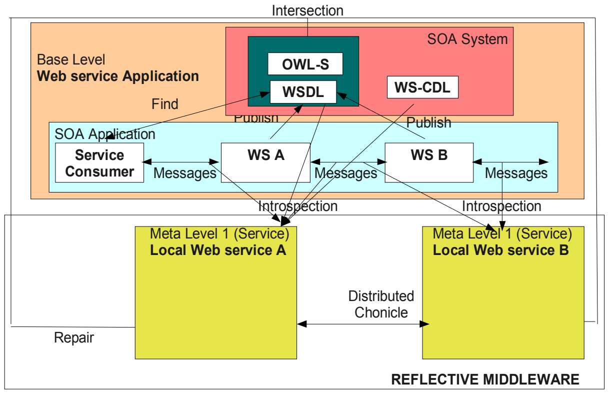

Fig. 1 ARMISCOM Reflective middleware architecture.

IV. ARMISCOM Architecture Based on Autonomic Computing

Autonomic Computing [6] is a self-managing computing model inspired by the autonomic nervous system of the human being. It creates systems that are able to be self-managed with high-level management while keeping the system's complexity invisible to the user. It incorporates sensors and actuators to the systems to allow collecting details about the system behavior and to act accordingly. Autonomic Computing defines an architecture composed of 6 levels (see Fig 2) [6]:

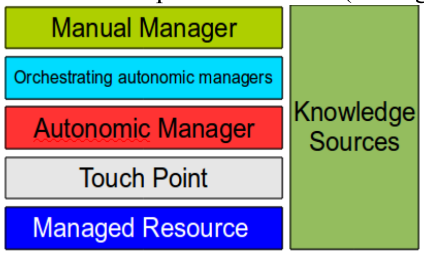

Fig. 2 Autonomic Computing Architecture.

- **Managed Resource:** can be any type of resource (hardware or software) and may have embedded self-managing attributes.
- **Touch Point:** implements the sensor and/or actuator mechanisms for the managed resources.

[1] Web Services Description Language (WSDL) Version 2.0 Part 1, http://www.w3.org/TR/wsdl20.
[2] Universal Description, Discovery and Integration, http://uddi.xml.org/.
[3] OWL-S: Semantic Markup for Web Services, http://www.w3.org/Submission/OWL-S/. .
[4] SCA (Service Component Architecture**)** provides a model for composing applications that follow Service-Oriented Architecture principles.
[5] FraSCAti project, http://frascati.ow2.org/.

- **Autonomic Manager:** implements the intelligent control loops that automate the tasks of self-regulation of the applications. Autonomic manager is composed by four phases identified as the MAPE (Monitoring, Analysis, Planning and Execution) autonomic control loop. The Monitoring phase, getting events/data from the sensor interface. The Analysis phase given by the diagnosers, the Planning phase to decide how to repair and the Execution phase to send the orders to the components via the actuator interface.
- **Orchestrating autonomic managers:** Provides coordination among the Local Autonomic Managers.
- **Manual Manager:** Creates a consistent human-computer interface for the autonomic managers.
- **Knowledge Sources:** Provides access to the knowledge according to the interfaces prescribed by the architecture.

Additionally, FraSCAti [7] is a platform for the implementation of SCA and fractal components[6] [8], flexible and extensible, that allows [9]: a run-time adaptation, property management and reflective capabilities. With Frascati we can implement a component as a fractal component with a set of controllers called membranes (these controllers allow introspection, configuration and reconfiguration).

Our middleware is composed of a set of distributed resources that work together to reach a global goal, which can be seen as an autonomic computing system. For this, we extrapolate the two levels of our middleware (meta and base) into the 6 levels of an Autonomic Computing Architecture (see fig 3).

In our case, the Managed resources and the Touch Point correspond to the base level; the autonomic Manager, the Ontology Framework and the Choreography Autonomic Manager define the meta level. The Touch Point interface is implemented/required by the services in order to retrieve monitoring data via the sensors interface and to enforce the decisions for repairing via the actuators interface.

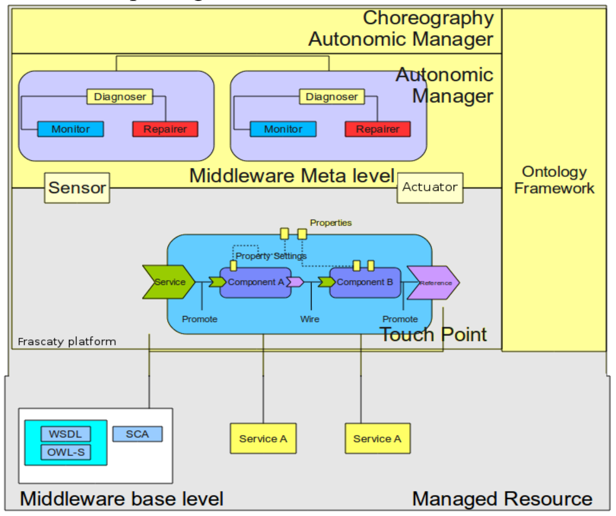

Fig. 3 Middleware Architecture based on Autonomic Computing.

[6] Fractal components are used in reflective systems to design, implement, deploy and reconfigure various systems and application.

V. MAPE MODULE

Our Autonomic Manager is composed by the three modules of our meta level: Monitor, Diagnoser and Repairer, which correspond to the MAPE structure of the Autonomic Computing Architecture (see fig 4). Each autonomic manager can work locally (level of each service - Internal failures of the service) and/or globally i.e. with the interaction among the autonomic managers is possible to diagnose failures in the services composition. Thus, this component must communicate with the rest of the Repairers of the other services of the composition.

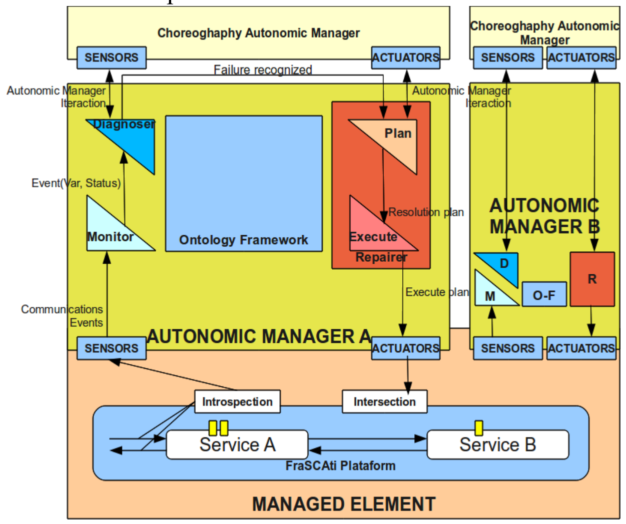

Fig. 4 The MAPE Middleware Architecture.

A. Monitor Component

The monitor is responsible for verifying the operation at run-time of a web service (observed behavior), according to the requirements specified in the choreography for the proper coupling within the composition (expected behavior). At time of performing the interaction of the services in the composition, there is a set of communications among them that allow the invocation of its operations, which are intersected by the monitor to find contradictory behaviors.

The execution of a service is performed by exchanging messages, when other service invokes the operation of a service (call) and when the invoked service generates the result of the operation (response); wherein the message contents are widely described in the WSDL. When we analyze the exchange of messages, it is clear that this can be seen as the occurrence of two events that occur sequentially:

- The invocation of the operation of a service where a set of input variables is given at a time t0: call(Operation, var, t0).
- The answer to the requested operations where a set of output variables is given at a time t1, which is the result of the invocation operation: response(Operation, var, t1).

To find anomalies in the exchange of messages, it is necessary to enrich the call and answer events previously defined with quality characteristics that allow the monitor to infer the occurrence of abnormalities. To do this, the monitor performs an analysis of consistency among the messages obtained and specified in the WSDL for the call and answer events, allowing it to determine abnormal behavior. Additionally, the status of the answer event is enriched with quality characteristics that are commonly used in web services as the service performance, etc. Thus, to determine the status of events we consider:

Call:

- The consistency among the variables defined for the call event in the WSDL and obtained in the message.

Answer:

- The error response value of the services.
- The consistency among the variables defined for the answer event in the WSDL and obtained in the message.
- The QoS requirements of the services.

B. Diagnoser Component

It makes the diagnosis of failures of the services involved in the choreography. The monitor component sends to the diagnoser the evolution of the events occurring in the execution of services in the choreography, which allows the diagnoser to obtain the sequence of events and thus be able to recognize a temporal pattern that determines a fault. This temporal pattern is called a *Chronicle*.

A chronicle is a set of observable events that have time restrictions among them. Chronicles allows to characterized a given situation [14]. In our case we have enriched the event to indicate when a service sends (called operation -) or receives an information (called operation +) and the status of the variables exchanged (consistency errors, etc.) in the messages.

Our proposal allows a fully distributed recognition of the chronicles, where the events will be detected locally by the different sub-chronicles allocated to local diagnosers (a diagnoser for every web service). The recognition emerges from the composition of local diagnosers.

C. Repairer Component

It has mechanisms for the resolution of faults present in the composition of services. When a fault is diagnosed, repairer is invoked to execute the relevant mechanisms to solve the problem.

This component has mechanisms that allow skipping, reallocating web services, among other things. Additionally, it can modify the execution of the flow of the choreography replacing and/or adding services, and compensating the connections among these using adapters. The repairer has 2 sub-components:

- **Plan:** To develop the set of strategies to solve the fault.
- **Execute:** To execute the reparation procedures in the FraSCAti platform according to the plan defined previously

VI. The ontology framework

The ontology Framework manages all the knowledge used by the middleware to perform its tasks. Our middleware will be composed by the SOA System, QoS and Fault-Recovery ontologies, chronicles as fault signatures, and specific web services ontologies.

A. SOA System

SOA system provides a way to integrate heterogeneous services defining APIs for easy implementation [17]. The SOA requirements for the proper functioning of our middleware are related to those that allow us to make the description, search and interaction of services. Thus, the standards to be used by our middleware are:

Web Services Description Language (WSDL): it provides a model for describing how the services can be called, what parameters are expected and what functionalities are offered.

Web Services Choreography Description Language (WS-CDL)[7]: it defines a model for describing Web Services Choreography. It is focused on the peer-to-peer collaboration among services.

Semantic Markup for Web Services (OWL-S): it defines an ontology for describing semantically Web services [18], allowing to automate tasks of discovering, invoking, composing, and monitoring of web services.

B. QoS Ontology

Our QoS ontology will provide a structured collection of concepts and relationships that describe the quality of services. Below are shown the QoS requirement ontology for each component of the MAPE module:

Monitor: Each time the monitor performs the invocation of a service, then it needs to calculate, compare and store the values of QoS

- Storing the measured values of QoS that are generated (QoS measured)
- Compare the measured values with those required (QoS required).

Repairer: The repairer needs to find and select Web services to perform its reparation tasks. Examples of utilization of the ontology by reparation tasks are:

- QoS values could be in different units of measurement that will be necessary to convert to achieve consistency of information.
- Compare the QoS required with those providing by the services (QoS provided).

In considering the requirements of our MAPE, we need an ontology of QoS in terms of:

- **QoS required:** defines the quality requirements of each service involved in a choreography. When a service is implemented in a choreography, it has some quality requirements for a proper operation

[7] Web Services Choreography Description Language, http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/

(expected behavior). QoS required is used by the Monitor to compare the measured values with the required values and by the repairer to establish the required values in the services.

- **QoS provided:** defines the service quality provided. It is used for the repairer to services search when it needs to perform a service substitution.
- **QoS measured:** Is used mainly by the monitor. Corresponds to the level of quality measured.

Figure 5 shows the relationship of our QoS ontology with the SOA system, the QoS Required is mainly used by WS-CDL because it describes the quality requirements in terms of operations and participants. In addition, QoS Provided is used both by the WSDL as the Owl-S to define the values provided by the service

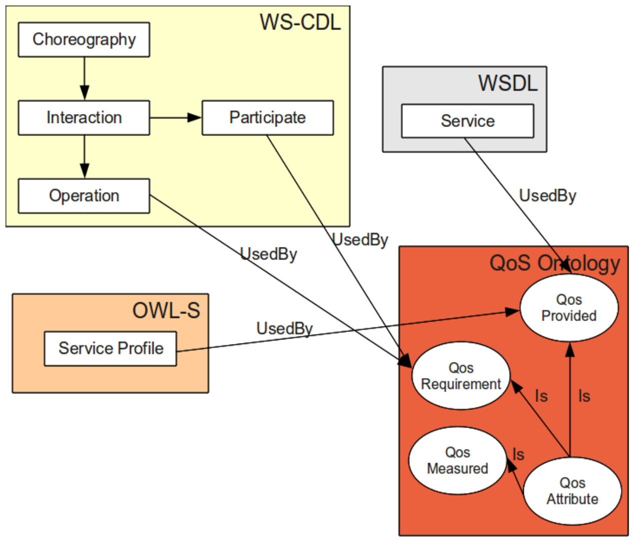

Fig. 5 QoS Ontology and SOA Systems relationship.

C. Chronicle Ontology

A chronicle is a set of observable events that have temporal restrictions and characterizes a situation [14]. To describe a chronicle in ontological terms, it is necessary to determine the knowledge that will be expressed:

- A chronicle is composed of n sub-chronicles.
- A sub-chronicle is composed of a series of events.
- In our case, an event consists of the operation name: sending (-) or receiving (+). Additionally, it describes the status of the variables of the event, and the occurrence date.
- The events have temporal constraints among their occurrences dates.

This ontology is used to characterize the events that are happening in the execution of the services (monitor) and to define the sequence of events that are part of each sub-chronicle (diagnoser). Figure 6 shows the relationship of our Chronicle ontology with the SOA system: events are mapped to the service operations (WSDL and WS-CDL) in a choreography.

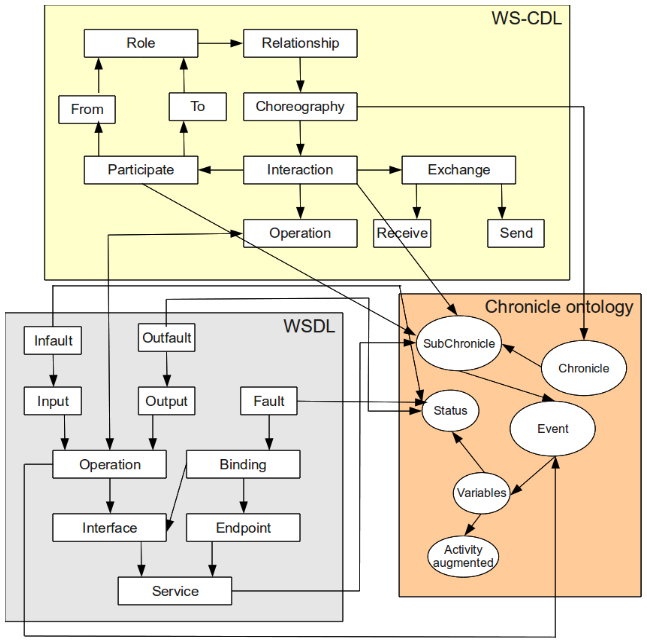

Fig. 6 Chronicle Ontology and SOA Systems relationship.

D. Fault-Recovery Ontology

This ontology is used mainly by the repairers. It defines the relationships between the faults that may occur in the choreography and the mechanisms for the resolution of the faults. Thus, it is based on the fault taxonomy proposed in [2], which specifies the failures that may occur in the service invocation and in a composition, and the reparation mechanisms proposed in [13]. Figure 7 shows the relationship of our Fault-Recovery ontology with the SOA system. Failures occur in the different interactions of service operations and reparation mechanisms are necessary to modify the service performed or to redefine the flow interactions in services choreography. Additionally, our ontology needs to be connected with the methods of resolution of faults (scripts to be implemented on the platform Frascati).

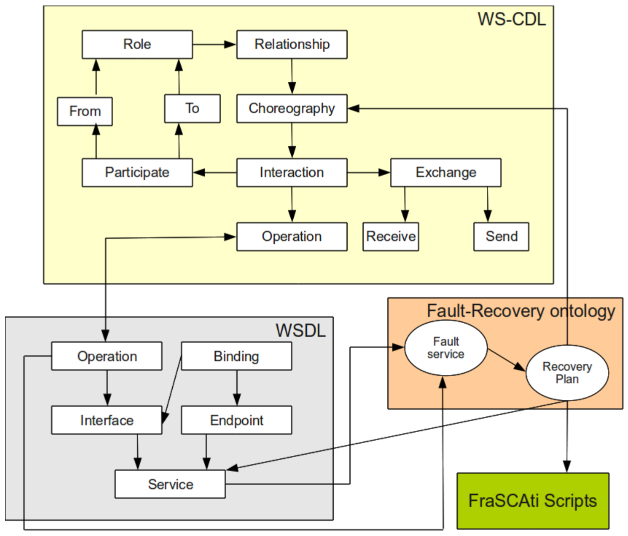

Fig. 7 Fault-Recovery Ontology and SOA Systems relationship.

Finally, figure 8 shows the different ontologies used by our middleware and the different relationships of these ontologies with the MAPE components.

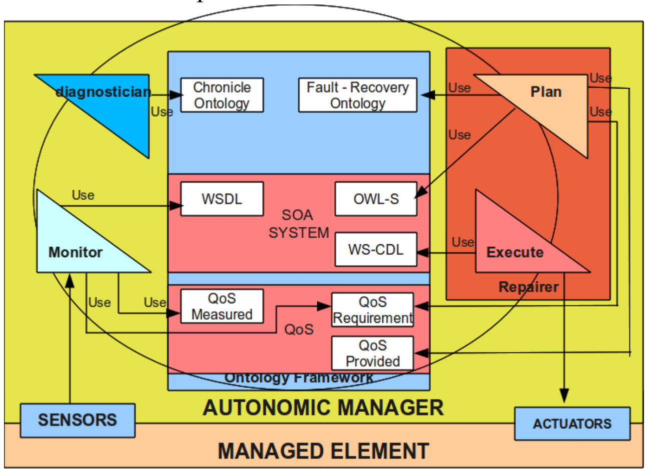

Fig. 8 The ontology framework.

VII. Case Study

In order to test our proposal, we are going to use a very common example of e-commerce composed by three business processes involved in the choreography (see figure 9):

- **Shop:** is the store where users buy products.
- **Supplier:** provides products to the shop; needs to check the warehouse before issuing an answer to the store.
- **Warehouse:** is where products are stored. This process has a service agreement (SLA) which consists in that at least one product from the list must be returned. It can carry out searches on external sites to buy products.

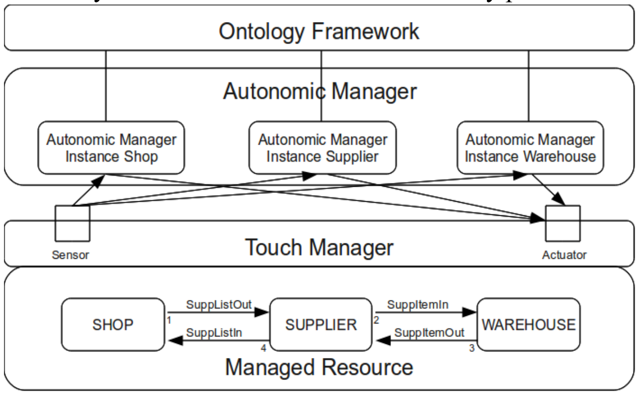

Fig. 9 Choreography example.

Now, we describe a classical behavior of this application:

(1) SuppListOut: Shop provides the list of products required to the supplier.

(2) SuppItemIn: Supplier checks its deposit invoking the Warehouse process.

(3) SuppItemOut: Warehouse provides the answer about the list of products in the deposit to the Supplier, which must contain at least one product.

(4) SuppListIn: The Supplier notifies the Shop which products can be provided.

Remember that the middleware contains many instances of the autonomic manager as services are in the composition. In general, the Ontology Framework is consulted by the Autonomic Managers to perform their functions, and the touch points are sensors and actuators provided by the service-composition interface. To show the operation of our autonomic managers we are going to consider the following situations:

A. Warehouse SLA violation (Web service fault)

The iterations (1) and (2) are normally executed, but (3) provides an empty list of products to the Supplier. This is a violation of the SLA for the Warehouse service. The solution is to adjust the Warehouse service configuration to perform an external search of products in order to provide at least one product. The middleware must perform the reparation (warehouse settings) to ensure the proper functioning of the choreography (see figure 10):

i. Shop monitor emits SuppListOut event to the shop diagnoser.

ii. Supplier monitor emits SuppListOut event to the supplier diagnoser.

iii. Supplier monitor emits SuppItemIn event to the supplier diagnoser.

iv. Warehouse monitor emits SuppItemIn event to the Warehouse diagnoser.

v. Warehouse monitor emits SuppItemOut event to the Warehouse diagnoser.

vi. Supplier monitor emits SuppItemOut event to the supplier diagnoser.

vii. Supplier diagnoser recognizes the sub-chronicle of external error (incorrect result) and propagates the SuppItemOut event to the Warehouse diagnoser.

viii. Warehouse diagnoser recognizes the subchronicle of internal error (incorrect result) and invokes Warehouse repairer.

ix. Warehouse's repairer performs the reparation using the knowledge base. It determines that the solution is to change the properties of the service so that it can search products in external storages (for that, it must execute the FraSCAti setAttribute script).

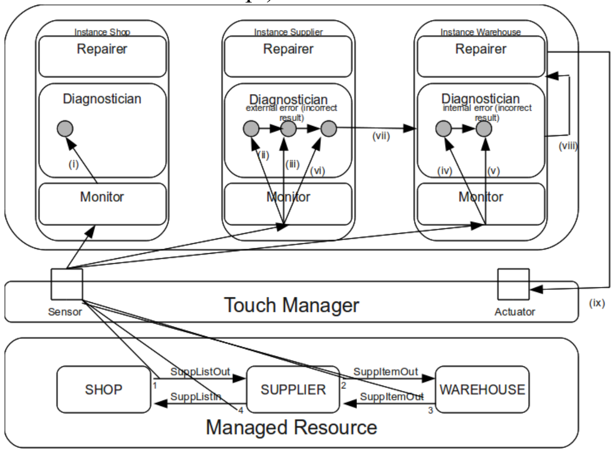

Fig. 10 Middleware self-healing due to SLA service violation

B. Warehouse Service Delay (Choreography flow fault)

Again, the iterations (1) and (2) are normally executed, but (3) has a delay which generates an error in the composition because the supplier cannot produce results for the shops (Supplier is unable to give its response in time). A possible solution would be a warehouse service reallocation (see figure 11):

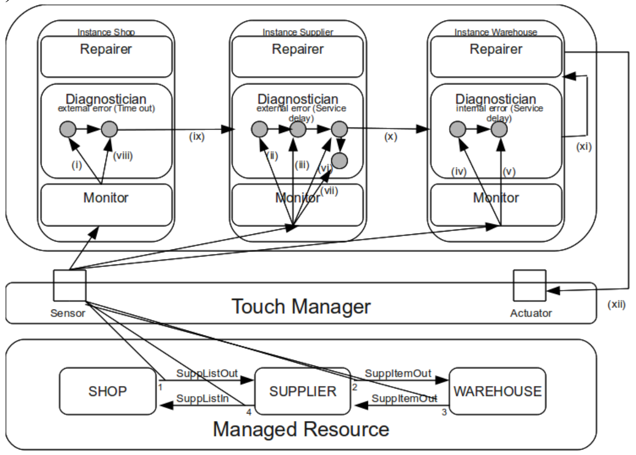

Fig. 11 Middleware self-healing due to faults in service Delay.

i. Shop monitor emits SuppListOut event to the shop diagnoser.

ii. Supplier monitor emits SuppListOut event to the supplier diagnoser.

iii. Supplier monitor emits SuppItemIn event to the supplier diagnoser.

iv. Warehouse monitor emits SuppItemIn event to the Warehouse diagnoser.

v. Warehouse monitor emits SuppItemOut event to the Warehouse diagnoser.

vi. Supplier monitor emits SuppItemOut event to the supplier diagnoser.

vii. Supplier monitor emits SuppListIn event to the supplier diagnoser.

viii. Shop monitor emits SuppListIn event to the shop diagnoser.

ix. Shop diagnoser recognizes the sub-chronicle of external error (Time out) and propagates the SuppListIn event to the Supplier diagnoser.

x. Supplier diagnoser recognizes the subchronicle of external error (service delay) and propagates the SuppItemOut event to the Warehouse diagnoser.

xi Warehouse diagnoser recognizes the sub-chronicle of internal error (service delay) and invokes Warehouse repairer.

xii. Warehouse's Repairer performs the reparation using the knowledge base. It determines that the solution is to reallocate the warehouse service (for that, it executes the FraSCAti reallocateService script).

C. Results Discussion

The architecture of our middleware can perform diagnostic and repair tasks in web service composition. The results show that in both case studies, the different Monitor modules in each service translates the state of the interaction of web services to the status of the events occurring in the composition making use of the QoS Ontology.

Each diagnoser performs events analysis based on distributed chronicles. It can infer the presence of faults. As shown in the two case studies, sub-chronicles are recognized at different level of services and the communication among the diagnosers allows to recognize a problem in the composition. In case 1 (Warehouse SLA violation) the Supplier diagnoser module recognizes the external error sub-chronicle (incorrect result) and Propagates the event to the Warehouse SuppItemOut diagnoser, which allows to recognize the sub-chronicle of internal error (incorrect result). The same happens in the case study Delay Service Warehouse. Finally, having diagnosed the faults repairer modules are invoked, which execute the reparation mechanisms to keep the correct functioning of the composition.

As shown in both case studies, our middleware architecture can perform the tasks of monitoring, diagnosis and repair fully distributed, locating each MAPE instance on each service, and through the interaction of them make the diagnosis and repair faults present in the composition.

VIII. CONCLUSIONS

In this paper we propose a reflective middleware Architecture for management of service-oriented applications. Our middleware is designed to be fully distributed through all services of the SOA Application, counting for this with the base level that contains both the SOA system and the SOA Application, and the meta level with components to execute the reflection. The architecture uses the model of Autonomic computing which will allows easy adaptation of our self-healing system. Additionally, our base level will use the Frascati framework to facilitate the implementation of introspection and intersection in our middleware.

We have evaluated the design of our middleware, shown the distributed adaptive capabilities of our architecture for failures covering both services and composition of services, thus our design is able to fix these failures, which contrasts with other approaches that do not show a fully distributed approach (see table I).

Architecture	Monitoring phase	Diagnosis phase	Recovery phase
SOAR [6]	Centralized	Centralized	Centralized
self-healing Architecture [7]	Centralized	Centralized	Centralized
self-healing in Dynamic Web Service Composition [14]	Centralized	Centralized	Centralized
Chronicle Architecture [8]	Distributed	Semi-centralized[8]	Centralized
Web Services with Diagnostic Capabilities [15]	Distributed	Semi-centralized	Centralized
Our Reflective Middleware	Distributed	Distributed	Distributed

Table 1 Comparing our architecture with other works.

[8] Distributed but coordinated by global diagnoser.

Finally, the ontology framework represents the knowledge needed to perform the operations of the middleware, which also is distributed completely. In future works we need to develop structures to represent this distributed knowledge (chronicles, etc.) and mechanisms to use it. This work is still in progress, but the initial results are interesting and promising.

Acknowledgment

This work has been supported by FP7-ICT IMAGINE research and development project, funded by the European Commission under the Virtual Factories and Enterprises (FoF-ICT-2011.7.3, Grand Agreement No: 285132) and the PCP program "Supervision and maintenance tasks in a shared organizational environment**".**

.

References

[1] K. Czajkowski, S. Fitzgerald, I. Foster and C. Kesselman, "Grid Information Services for Distributed Resource Sharing". Proceeding of the 10th IEEE International Symposium on High Performance Distributed Computing, pp. 181--184. 2001.

[2] K. Chan, J. Bishop, J. Steyny, L. Baresi and S. Guinea, "A Fault Taxonomy for Web Service Composition", Proceeding of the Service-Oriented Computing Workshop, pp. 363-375, 2007.

[3] G. Huang, X. Liu and H. Mei, "SOAR: Towards Dependable Service-Oriented Architecture via Reflective Middleware". International Journal of Simulation and Process Modelling, Vol. 3, Issue 1/2, pp. 55-65, 2007.

[4] R. Halima, E. Fki, K. Drira and M. Jmaiel, "Experiments results and large scale measurement data for web services performance assessment". Proceeding of the IEEE Symposium on Computers and Communications, pp. 83-88, 2009.

[5] WS-Diamond project, "WS-Diamond, IST-516933, Deliverable D4.3, Specification of diagnosis algorithms for Web Services – phase 2", http://wsdiamond.di.unito.it/.

[6] IBM Corporation. "An architectural blueprint for autonomic computing". Autonomic Computing", Fourth Edition, http://www.ginkgo-networks.com/IMG/pdf/AC_Blueprint_White_Paper_V7.pdf, 2006.

[7] OWS2 Consortium, "FraSCAti project", http://frascati.ow2.org/, 2011.

[8] J. Adamek, T. Bures, P. Jezek, J. Kofron, V. Mencl, P. Parizek, P and F. Plasil, "Component Reliability Extensions for Fractal Component Model", http://kraken.cs.cas.cz/ft/public/public_index.phtml, 2006.

[9] P. Hnetynka, L. Murphy and J. Murphy, John, "Comparing the Service Component Architecture and Fractal Component Model", The Computer Journal, Vol. 54 Issue 7, pp. 1026-1037, 2011.

[10] P. Maes, "Concepts and Experiments in Computational Reflection", Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, pp. 147-155, 1987.

[11] S. Poonguzhali, R. Sunitha, and G. Aghila, "Self-Healing in Dynamic Web Service Composition". International Journal on Computer Science and Engineering, Vol. 3, No. 5. pp. 2054-2060, 2011.

[12] L. Ardissono, L. Console, A. Goy, G. Petrone, C. Picardi, and M. Segnan, "Enhancing Web Services with Diagnostic Capabilities". Proceedings of the Third European Conference on Web Services, pp. 182-191, 2005.

[13] K. Wiesner, R. Vaculín, M. Kollingbaum and K. Sycara, "Recovery mechanisms for semantic web services". In Distributed Applications and Interoperable Systems, Lecture Notes in Computer Science, Vol. 5053, pp 100-105, 2008.

[14] X. Le Guillou, M. Cordier, S. Robin and L. Rozé, "Chronicles for On-line Diagnosis of Distributed Systems". Proceedings of the 18th European Conference on Artificial Intelligence, pp. 194-198, 2008.

[15] Z. Mahmood. "Service Oriented Architecture: Potential Benefits and Challenges". Proceedings of the 11th WSEAS International Conference on Computers, pp. 497-501, 2007.

[16] A. Lin, "Conceptual Model for Business-Oriented Management of Web Services", Proceedings of the 6th WSEAS Int. Conf. on Software Engineering, Parallel and Distributed Systems, pp. 80-85, 2007.

[17] F. Ismaili and B. Sisediev. "Web services research challenges, limitations and opportunities", WSEAS Transactions on Information Science and Applications archive Vol. 5, Issue 10, pp. 1460-1469, 2008.

[18] D. Chiribuca, D. Hunyadi, E. Popa, "The Educational Semantic Web", 8th WSEAS International Conference on Applied Informatics and Communications, pp. 314-319, 2008.