Towards Autonomic Enterprise Service Bus

Denis Morand, Isaac Noé Garcia Garza, Philippe Lalanda

Towards Autonomic Enterprise Service Bus

Denis Morand
Grenoble Univ. / Schneider Electric
220 rue de la chimie
38 000 Grenoble
France
Denis.Morand@imag.fr

Issac Garcia
Grenoble University
220 rue de la chimie
38 000 Grenoble
France
Issac-Noe.Garcia-Garza@imag.fr

Philippe Lalanda
Grenoble University
220 rue de la chimie
38 000 Grenoble
France
Philippe.Lalanda@imag.fr

ABSTRACT
In this paper, we describe an ongoing work tending to make autonomic a mediation framework called Cilia. Cilia is an open source component-based mediation framework initiated by the LIG/Adele team at Grenoble University and France Telecom. Cilia has been designed for data and application mediation and is used in several industrial use cases. This paper presents approaches that are currently pursued to obtain a self-managed mediation framework.

Keywords
Enterprise Service Bus, Autonomic.

1. INTRODUCTION
Service-oriented Computing (SOC) has recently emerged in the software engineering community [1][2][3]. The very purpose of this reuse-based approach is to build applications through the late composition of independent software elements, called services. Services are described and published by service providers; they are chosen and invoked by service consumers at runtime. This is achieved within a service-oriented architecture (SOA), providing the supporting mechanisms.

Service orientation brings in major software qualities. First, it favors the rapid development of quality software applications. It also promotes weak coupling between consumers and providers, reducing dependencies among composition units. Finally, late binding and substitutability improve adaptability. Since a service can be chosen or replaced at runtime, it is easier to improve the way requirements are met.

A number of implementations have been proposed in the last years. Web Services (www.w3c.org), for instance, represent a solution of choice for software integration. UPnP (www.upnp.org) and DPWS (Devices Profile for Web Services) are heavily used in pervasive applications in order to implement volatile devices. OSGI (www.osgi.org) and iPOJO (www.ipojo.org) provide advanced dynamic features to many software systems.

That being said, service-oriented computing also suffers from important limitations. In particular, it is complex to conceive and implement an application made of dynamic, heterogeneous services and required to meet non functional requirements. Doing so requires deep expertise. Cross-technology applications require almost unavailable skills. In addition, as of today's state-of-the-art, service composition cannot be based only upon service specifications. Syntactic compatibility does not ensure semantic compatibility. In practice, service composition is based on unexpressed assumptions and rules allowing reaching the expected results. A composition of services has also to reach a set of pre-defined non functional qualities (like security for instance) which requires the production of complex, often non flexible code. In the general case, such code cannot be automatically generated at composition time.

We believe that without effective solutions for easy and correct service composition, SOC orientation will be limited to narrow, very specific domains of applications. In this paper, we present a mediation tool allowing the effective integration of services. This tool, based on a domain-specific component model, allows the creation of mediation chains implementing the necessary non functional operations when calling a service. It has been successfully used in several use cases, at France Telecom in particular. It however appears that the management of such tools is difficult in the sense that it has to deal with the high volatility of services. The provisioning of high-level services based on heterogeneous, distributed and mobile software applications and hardware devices is a difficult task. Dynamism is a particularly complex and remains an important issue in service-oriented computing. This is required as applications evolve with their execution contexts, when software and hardware components get modified, or as users change their computing environments or desires.

Autonomic Computing promises a solution to the aforementioned problem, by endowing software systems with self-management capabilities that would minimize or eliminate the need for human intervention [Joh]. If successfully implemented, autonomic pervasive applications would inherently feature critical properties such as safety (including fault-tolerance and security) and self-adaptation to internal and external changes (including self-configuration, self-optimization and self-repair). However, building autonomic properties into pervasive systems remains a difficult task. Reusable solutions for the development of Autonomic Management (AM) systems remain rather limited and generic. There is a stringent need for more specific, readily-usable frameworks for facilitating the development of AM solutions for different computing domains

In this paper, we also examine how Cilia can be made autonomic. The paper is organized as follows. First, background about Enterprise Service Buses is given. Then, the CILIA component model and associated runtime framework is presented. The fourth section is concerned with autonomic extensions brought to Cilia. More precisely, the notion of state variable is presented through the definition of a metamodel. The paper ends with concluding remarks.

2. ENTERPRISE SERVICE BUS

The activity of integrating disparate information sources in a timely fashion is known under the name of mediation. Mediation has been historically used to integrate data stored in IT resources like databases, knowledge bases, file systems, digital libraries or electronic mail systems [4,5,6]. It is now also used to allow interoperation between heterogeneous software applications. In this context, mediation software stands between client applications and provider applications. Its purpose is to enable a consumer to easily and properly use a provided service. We use the term mediation service to refer to software allowing the integration of service-based applications.

Service mediation implements all the operations that are necessary to enable the actual communication between a set of service-based applications. The most common functions to be provided are the following:

- Communication. The primary purpose of mediation is to enable applications using different communication protocols to interoperate. This is implemented by means of protocol transformations as in a network bridge. This function can also play the role of a broker, hiding for instance the applications network addresses
- Syntactic alignment. The purpose of this function is to align data formats. This can be done between each application or through an intermediary format. In the latter case, the number of data transformations to be made is obviously reduced.
- Semantic alignment. The purpose of this function is to align data semantics. In the absence of recognized and used standards, applications develop different ontologies to represent (static and dynamic) knowledge. Automating ontologies alignment is a major research challenge of the service community.
- Non-functional properties. The purpose of this function is to ensure certain quality properties in the application exchanges, like for instance security or availability.
- Persistency. The purpose here is to keep track of all exchanges between applications. The mediation layer can provide logging support for all requests, responses and data.
- Monitoring. The purpose of this function is to collect data for monitoring systems that verify that the expected quality of service is being achieved.
- Business logic code. The mediation layer can be used to insert business logic code, like an access to a database for instance. Even though this approach can be particularly useful, its use is rather not recommended. It actually introduces confusion as it produces an architecture where the business logic code and technical code are mixed.

Encapsulating such operations in dedicated software is clearly a good practice. Mediation software provides a single point of interface to the different applications implied in the communication. This reduces the number of connections needed and facilitates change management. Mediation also provides an isolation layer from software details and, if appropriately configurable, permits the quick and cost-effective development of new services. The mediator layer improves reusability and evolution of applications. It also permits the transparent addition of new QoS properties such as security, reliability, etc. Finally, it leads to the improvement of the scalability of the whole system.

The mediation layer is often achieved through the use of an EAI (Enterprise Application Integration). EAIs usually appear as monolithic software based on the *hub and spoke* pattern. EAIs have been widely used in the last few years. They now must face heavy criticism, due to their cost and size. We believe that this is partly due to the fact that EAI have gone too large, exceeding their initial functional scope. Also, a single EAI is often used to integrate all applications of a company. Any new service using existing applications has to go through such unique EAI.

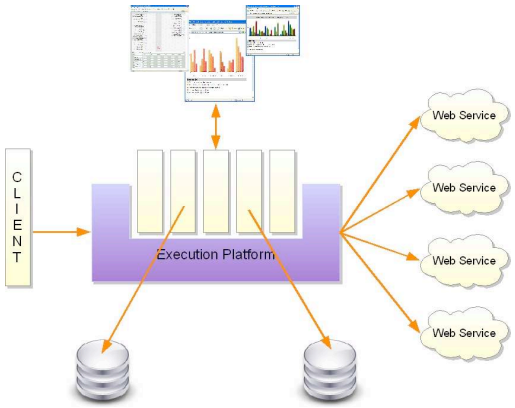

Figure 1. An ESB provides a run-time environment mediating Web service applications.

The emergence of service-oriented computing has fostered architectural evolutions. In particular, Web services aim for lighter integration solutions and have initiated the definition of Enterprise Service Buses, or ESB [7]. An ESB is a communication bus located between clients and Web Services and hosting potentially distributed mediation operations. Mediation is frequently organized as mediations chains that transport the request from the client application to the service provider and the answer the other way around. The mediation chains can be decomposed into light weight components called mediators that implement simple operations. ESsB provide a unique interface to all applications and eliminate all direct contact between applications, as all communication is made through the bus.

ESBs raise major design and implementation-related issues. They have to meet stringent requirements, including:

- **Lightweight**. The primary demand to be satisfied by an ESB is to be lean. The purpose of companies we are working with is to install ESBs on demand to make targeted applications interoperate. They want to avoid the EAI syndrome where all possible applications are linked to a single, fat EAI.
- **Efficient**. This is a major feature for all integration software. Time needed to align data and to perform non functional operations should not impact the quality of the overall service provided by the integration.
- **Easy to install and manage**. Since ESBs are to be used frequently, on the creation of a new service, their installation, configuration, management have to be simple.
- **Flexible**. Non functional requirements evolve over time. Then, it should be easy to modify or add mediation operations in order to adapt at run time the way applications are integrated.
- **Easy to use**. Programming and run-time models have to be simple. Once again, the point is to avoid getting back to nasty solutions where programming and maintaining mediation code is so hard that consultancy or dedicated teams, expensive in both case, is unavoidable.
- **Error handling**. One of the salient requirements brought by our industrial partners is the ability to easily deal with errors.

Application integration is subject to many errors (communication failures, inadequate or non running services, incorrect answers, etc.). An important part of integration code is actually dedicated to error handling.

Several solutions have been recently proposed. We can actually distinguish between two architectural approaches. The first approach is to extend a J2EE server. This solution consists in developing an ESB on top of an existing J2EE application server (WebSphere ESB or SpringIntegration for instance). The first appeal of this approach is obviously to reuse the J2EE programming model and the servers facilities. The result is however often very big in size. In addition, the programming model is not perfectly adapted to mediation. As a remedy, domain-specific tools like Camel[1] have been proposed.

A second approach is to develop dedicated tools. Many tools have been proposed in recent year like Codehaus Mule[2]. A standard, called JBI for Java Business Integration, has actually been proposed to structured ESB. JBI is based on the JSR 208 that standardizes a component-based architecture. Components are simple objects orchestrated by a controller named Normalized Message Router. A distinction is made between pure mediation component (Service Engines) and components used to interact with other resources (Binding Components). Some open source tools based on JBI are emerging like Apache ServiceMix[3] or ObjectWeb Petals[4].

But, in all cases, the mediation solutions are very technical and technology-driven. Mediation chains are hard to build, deploy and maintain. They are also uneasy to change and to reuse. Most ESBs mentioned here have been tested by our partner in real-size industrial use cases and failed to meet the requirements presented here before. We believe that there is a clear need to focus on mediation operations, to consider them as first order objects and to treat them accordingly. Complex, low-level technical details should be hidden by a mediation tool in order to allow developers to focus on their business, that is the integration of heterogeneous applications.

3. CILIA

In this section we present a mediation component model, called Cilia [8], which addresses the interoperability issues between heterogeneous data sources (e.g. applications, devices, etc.) and targeting systems. Cilia is based in a component model approach which emphasize the reuse and the separation of concerns. A mediation application in Cilia is a set of component instances interacting in a loosely coupled way through, but not limited to, event-based protocols. As with any component based model, Cilia relies on two main models, the specification model and the composition model. The specification model is used to define component type specifications. The composition model defines the way components instances are combined in application architecture.

Components are specified at development time. They are made of some java classes and a Cilia specific XML-based specification. More precisely, a component includes the following Java classes:

- **A scheduler class**. The purpose of this constituent is to synchronize data reception. It intercepts incoming data (requests), store them and launch their processing. The processing decision can be time-based, content-based or, any other condition on relation with the mediation context (e.g., waiting all needed data). For instance, a *periodic* scheduler triggers the processing with the collected data periodically, a *correlation* scheduler waits for all the correlated messages to trigger the processing, and an *immediate* scheduler triggers processing upon data arrival.
- **A processor class**. The processor performs the mediation algorithm *per se*. When notified by the scheduler, it processes the collected data and passes them to the dispatcher. *StringSplitter* is an example of processor that splits the received data using a regular expression. *StringAggregator* in another example that builds a new data concatenating the received ones.
- **A dispatcher class**. The dispatcher receives the processed data from the processor. This constituent decides on the data destination and triggers their delivery. The dispatcher choice is a logical destination because of loosely coupled relations between mediators. The *Multicast* Dispatcher is an example where processed data are delivered to all the connected components. The *content-based* dispatcher delivers the processed data to the chosen destination based in the data content.

Bindings are also defined at development time. They are based on the two following elements associated with mediators:

- **Collector**: The collector is the binding constituent which implements the communication protocol to receive data. The data could be received from other mediator or from external communication protocol or application.
- **Sender**: The sender is the binding constituent which implements the communication protocol to send the resulting data. This data could be sent to another mediator, could also be sent using some standard communication protocols, or sent to another application. This component is associated to the dispatcher mediator instance in execution.

A binding specification describes how communication is established. That is, a binding specifies which collector and sender are used for the communication between two mediators and how they need to be configured in order to assure correct communication. For a binding specification which uses a topic-based system, the binding should known which collector/sender it needs to use and how to set-up them with the correct topics when adding them to the corresponding mediators.

Bindings specifications are independent of mediators logic, thus mediators could use any binding specification. There are three binding types. The first binding type describes how to communicate between two mediators, thus a sender and collector must respectively be declared for the receiving mediator and the transmitting mediator. The second binding type is the one that allows mediators to receive data from an external system, e.g. a database or through a communication protocol. Therefore, only a collector is defined. The third binding type, is the one that is used to deliver data to an external service or application, thus, only a sender is defined.

Let us now look at a simple example to illustrate these notions. The purpose of this use case is to implement a “split / aggregate” pattern which regularly occurs in applications integration. This pattern is structured as it follows:

[1] http://camel.apache.org/

[2] http://mule.mulesource.org/

[3] http://servicemix.org/site/

[4] http://wiki.petals.objectweb.org/

- A *split* phase: a request, sent by an application, is divided into three requests (potentially more). Each request corresponds to a call to a web service (exposing an application, a database …).
- An *aggregate* phase: the results of the three requests have to be aggregated to form a single answer. The way results are put together is domain specific. The answer has then to be delivered to the initial application.

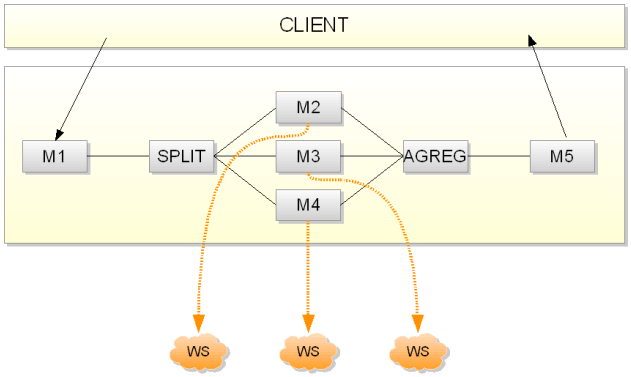

Figure 2. Split/aggregate use case

The way we deal with this use case is presented in Figure 2. We actually defined the following mediators:

- A first mediator, called M1 on the figure, gets the initial request from the client. It has to split the request into three requests and send them to the next mediators. To do so, we use available scheduler and dispatcher. The scheduler is very simple since it simply waits for a single request. The dispatcher is a bit more complex. It is configured with the output topics. The processor can be developed or reused. It is reused if the way the initial request is divided is somehow standard. For instance, the request can be divided based on XML).
- M2, M3, M4 are responsible for calling the Web Services and send the result to the next mediator. Here, schedulers and dispatchers are very simple (and obviously reused). A processor can be added to perform mediation operations if necessary.
- A last mediator, called M5 on the figure, has to aggregate the information collected by M2, M3 and M4 and form a single message which will be the client response. Here, the scheduler is complex. It is reused and configured.

4. AUTONOMIC CILIA

In order to make cilia autonomic [9,10], we identified the following tasks to be done:

- Identify the internal aspect of Cilia that were to be monitored by an autonomic manager,
- Capture these aspects at runtime,
- Define a model to store these aspects.

An important point to be understood here is that Cilia is a framework and that autonomic computing is very much concerned with application management. So, the purpose of an autonomic layer for Cilia is to provide the information necessary to conduct application-specific reasoning (through the implementation of a MAPE-K loop).

In an enterprise service bus, we actually can distinguish two kinds of data:

- State variables that can be used to quantify the process stability over time
- Action variables that can be used to modify the process under controlled.

The notion of state variable is inspired from work in control theory. A state variable is a data that quantifies an important aspect of a supervised process. For instance, it can be the size of a buffer or the number of running threads in Java. A state variable is a numeric data that comes with a validity interval that is used to specify a viability zone for a process. A state variable set is a set of such variable. It is used to define a viability zone for a process. This means that when all the variables in the set are all in a well defined interval, then the process is executing correctly.

The notion of action variables also comes from work in control theory. An action variable corresponds to a data related to the supervised process and that can be changed. It can be, for instance, the size of a buffer. An action variable can be directly related to a state variable, but it is not mandatory. For instance, the number of threads can be a state variable and not an action variable (it cannot be externally modified).

Figure 3 presents the metamodel that has been defined in Cilia in order to specify the notions of state and action variables.

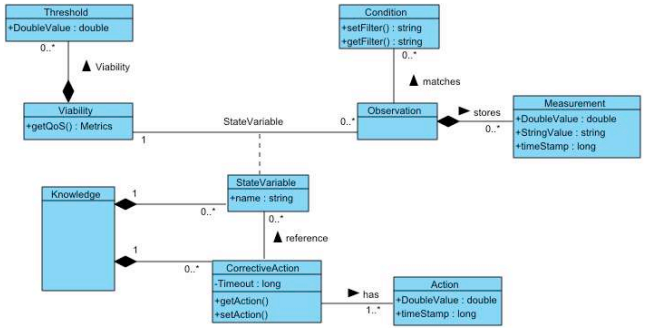

Figure 3. State and action variables

The following state variables are now captured and stored in Cilia:

- **The propagation Delay**. This variable measures the time needed to traverse a mediation chain in Cilia.
- **The transmission delay**. This variable measures the time of communication between two Cilia mediators.
- **The processing delay**. This variable measures the latency time in a Cilia mediator.

We also collect global information about mediators and mediation chains. More precisely, the following pieces of information are constantly traced in Cilia:

- The number of incoming messages per port and for each mediator,
- The number of outgoing messages per port and for each mediator,
- The number of calls to the processor,
- The mean processing time of the processors.

This is illustrated here after by figure 4.

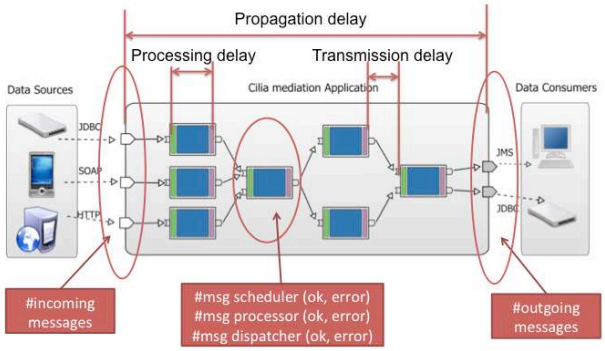

Figure 4. Captured information

These collected data are then used in a "classical" autonomic loop in order to adapt the mediation chain as indicated. Actions that can be undergone currently are much related to the way the Cilia framework is implemented. For instance, the size of different buffers (storing input and output messages) can be modified at runtime. Similarly, the number of threads used to manage messages can be adapted.

5. Conclusion

Cilia is a domain-specific component model dedicated to mediation. It is built on top of service-oriented technology [3] and is then adaptable at runtime. It however shows important management complexity, as any other current Enterprise Service Bus, and autonomic solutions are much required today to be used in industry.

In order to make it autonomic, we have defined the notion of state variables and action variables, inspired from works in control theory. These variables allow us to follow low level programming aspects belonging to the Cilia core framework and to adjust then whenever necessary.

We are now investigating the design and implementation of higher level autonomic decisions. The point here is to be able to change the topology of a mediation chain. This means first that a mediator can be added, removed or replaced. Similarly, new integration patterns can be inserted in order to form more adapted mediation chains. Our approach here is to rely on models expressing reference chains that can be adapted to run time situations. These models include explicit variability making room for run time decisions [11].

6. REFERENCES

[1] M. P. Papazoglou and D. Georgakopoulos. Service-Oriented computing: Introduction. Commununications of the ACM, 46 (10):24–28, October 2003

[2] SECSE team, "Toward service-centric system engineering", ICSOC, Trento, Italy, 2003.

[3] C. Escoffier, R. S. Hall, and P. Lalanda. iPOJO: an Extensible Service-Oriented Component Framework. IEEE International Conference on Services Computing (SCC), pages 474–481, 2007.

[4] G. Wiederhold, "Mediators in the Architecture of Future Information Systems," Computer, vol. 25, no. 3, 1992, pp. 38–49

[5] G. Wiederhold and M. Genesereth, "The Conceptual Basis for Mediation Services," IEEE Expert, vol. 12, no. 5, 1997, pp. 38–47

[6] P. Lalanda, L. Bellissard and R. Balter, " Asynchronous Mediation for Integrating Business and operational Processes," IEEE Internet Computing, vol. 10, no. 1, 2006, pp. 56–64

[7] C. Herault, G. Thomas, P. Lalanda, "A service oriented mediation tool" in Proceedings of the 4th IEEE International Conference on Services Computing (SCC'07), 2007, Salt Lake City, USA

[8] Garcia, Pedraza, Debabbi, Lalanda, Hamon, "Towards a service mediation framework for dynamic applications", IEEE APSCC, 6-10 december, 2010, Hangzhou, China

[9] D. M. Kephart, Jeffrey O. et Chess. The vision of autonomic computing. Computer, 36, 2003.

[10] M. C. Huebscher and J. A. McCann. A survey of autonomic computing—degrees, models, and applications. ACM Comput. Surv., 40(3):1–28, 2008.

[11] Yu and Lalanda, "An approach for dynamically building and managing service-based applications architectures", IEEE APSCC, 6-10 december, 2010, Hangzhou, China