Component-based generic approach for reconfigurable management of component-based SOA applications

Cristian Ruz
INRIA Sophia Antipolis,
CNRS, I3S, Univ. de Nice
Sophia Antipolis
2004 Rte. des Lucioles BP 93
F-06902 Sophia Antipolis
Cedex, France
Cristian.Ruz@inria.fr

Françoise Baude
INRIA Sophia Antipolis,
CNRS, I3S, Univ. de Nice
Sophia Antipolis
2004 Rte. des Lucioles BP 93
F-06902 Sophia Antipolis
Cedex, France
Francoise.Baude@inria.fr

Bastien Sauvan
INRIA Sophia Antipolis,
CNRS, I3S, Univ. de Nice
Sophia Antipolis
2004 Rte. des Lucioles BP 93
F-06902 Sophia Antipolis
Cedex, France
Bastien.Sauvan@inria.fr

ABSTRACT

Service Oriented Architecture (SOA) applications can be composed by sets of loosely coupled interacting heterogenous services from different providers. The Service Component Architecture (SCA) specification allows to build hierarchical applications, applying the principles of SOA and Component Based Software Engineering (CBSE). However, concerns like dynamic management, including reconfiguration and distribution handling for composite services are left as platform specific matters.

In this context, monitoring and management tasks are not trivial, since compositions and required QoS levels can change depending on the effective location that services and components are deployed onto. Service Level Agreements (SLA) can also evolve during the lifecycle of the deployed application. Several solutions for monitoring and adaptation of QoS-aware service compositions have been proposed so far, but they have rarely been designed in an integrated way and with evolution capabilities in mind.

In this work we advocate that a component based approach is an adequate one in order to implement a reconfigurable framework to handle tasks of monitoring and management of hierarchical component-based SOA applications. Our approach allows to address concerns like monitoring, SLA management and adaptation strategies, possibly autonomous ones, as a component-based distributed application. The main advantage is the capability to reconfigure this management architecture at runtime whenever needed, allowing to dynamically adapt it to the possibly evolving non functional requirements of the managed application.

The framework is illustrated through a scenario of a composite SOA application that is dynamically augmented with components to tackle non-functional concerns as it is needed. We describe an implementation over an SCA compliant platform that allows distribution and architectural reconfiguration of components.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Design, Management, Measurement

Keywords

Monitoring, Management, SLA Monitoring, Reconfiguration, Component-based software engineering

1. INTRODUCTION

Service Oriented Architecture (SOA) applications can be composed in several ways by using services from different heterogenous providers. At the same time these services can also be composed of, and consume other services, in a situation where providers also act as consumers of third-party services, and are usually called prosumers.

Moreover, by nature of SOA, it should be permitted to dynamically replace the actual services that compose the application by others that provide the same or equivalent functionality, given various and possibly changing conditions like price, availability, latency, energy consumption, response time, quality of the answer, and others.

Component-Based Software Engineering (CBSE) approaches have been used to tackle the complexity, dynamicity and heterogeneity of SOA applications. Among these, the industry supported Service Component Architecture (SCA) is a technologically agnostic specification used to build complex SOA applications, which has been set up to tackle the functional (business) complexity of SOA applications and allows to describe some non-functional concerns with the SCA policy framework. However, monitoring and management actions are usually left out of the "component world" and must be handled by SCA platform specific implementations, mainly because SCA is design-time and not runtime focused.

To ensure some Quality of Service (QoS) level in the composed application, contracts are established between consumers and providers in Service Level Agreements (SLA) stating conditions that must be met. The compliance to this SLA must be monitored at runtime, and requires to monitor several metrics related to the service. Due to both the dynamic and heterogeneous nature of the involved services and an evolving SLA scope, the set of services to monitor may change at runtime, and even the metrics and its associated aggregation/computation of values may be different for each service. In particular, the SLA conditions to monitor may also evolve at runtime due to new services involved in the composition, and different contracts may be established for different consumers which might not have been predicted at design and initial deployment time.

In such a dynamic context, monitoring and management of a composite service is a transversal activity impacting most (all) of the involved heterogenous services, and which may need to be implemented in different ways for each service depending on its built-in monitoring capabilities, available communication protocols to get and propagate the monitoring information, etc. However for implementing effective monitoring and management strategies, a common way to handle all these differences and specificities is desirable.

Our thesis is that monitoring and management concerns should be beneficially tackled by a component-based approach. We promote a solution to these concerns through a framework that permits to easily and flexibly implement monitoring and management features for complex, heterogeneous composite SOA applications, themselves component-based along the SCA specification. This framework benefits from component models characteristics, mainly the dynamicity one, allowing to add, update, or remove the monitoring and management concerns in order to adapt at runtime to the monitoring and management requirements of the application. More precisely, by splitting the traditional autonomic MAPE[1] control loop into separate and customizable components which interact through well-defined component interfaces, our approach allows to dynamically link the complete loop or part of it to the targetted services as it may be required, and in a personalized way.

Overall, our approach allows a great deal of flexibility as it allows to implement monitoring and management strategies that may be different for each service, and the different steps of the autonomic control loop can be implemented by different experts.

2. MOTIVATING EXAMPLE AND GENERAL VIEW OF OUR CONTRIBUTION

Consider a tourism office who has composed a smart service to assist visitors who request information from the city and provides suggestions of activities and touristic planning for visiting the city. The application uses a local database of touristic events (seasonal manifestations, carnivals), a set of attraction providers who sell tickets to parks, guided tours, etc., a weather service to guide the proposition of activities (if heavy rain is expected, only indoor activities are considered), a payment service to buy tickets in advance to some attractions, and a mapping service that will be used to create a document with a map including directions to each attraction. Once all information is gathered the application uses a local engine to compose a PDF document and optionally print it. The client can decide to buy immediately the tickets, in which case they will be provided with the printed document, or request only the map and planning. The composed design of the application is show in Figure 1.

[1]Monitor, Analyze, Plan, and Execute

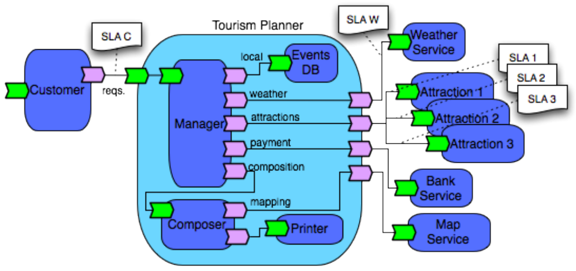

Figure 1: Scenario. SCA description of the application for tourism planning.

Such a composition involves some contracts for service provisioning established in SLAs. After the customer specifies the duration of the visit and the number of people, the Tourism Planner service agrees to propose a plan within 30 sec. (otherwise the customer may just leave). From the point of view of the application there are also contracts which are invisible for the customer, but important for the composed application. The Weather Service charges a fee for each forecast depending on the length of the demanded period and the granularity of the query (hourly, daily); the Mapping Service is freely available (it has no impositions on response time or availability); the Banking Service ensures an availability of the service of 99%; and each attraction can agree special conditions (discounts, promotions)

The set of conditions established with the providers and their runtime compliance to them may determine later decisions of the service. For instance, some of the Attraction Services may take too much time to deliver an answer, or the Mapping Service may not be available or have a poor performance at some moment. These kind of situations may arise and may require that a runtime decision be taken to avoid violating the SLA with the customer. The decision may involve actions like the removal or replacement of an slow provider, or changing some parameter on the composer to ignore the map while composing the document. To be able to make such kind of decisions, a precise and efficient runtime monitoring and SLA compliance system is required.

However, the monitoring requirements may be different for each service; for example, in the case of the printer it is important to measure the amount of paper or ink to avoid unavailabilities; in the case of the composer it is important to know the time it takes to create a document; some of the external services may provide their own monitoring metrics and, as they are not locally hosted and only accesible through a predefined API, it may be not possible to add specific monitoring on their side, nevertheless some metrics can be measured from the consumer side like response time, availability, cost of the answer, etc.

Having runtime monitoring capabilities is useful not only to verify certain SLA conditions, but also can help to drive optimizations of the service. An administrator can realize that the access to the Weather Service is too costly, and it may be better to cache the responses to previous requests, to be able to use them for some of the next requests.

As there are several external providers involved, the conditions expected from each one of them may change, and so the monitoring requirements over them. The Weather Service may decide to charge for daily forecasts, instead of queries, and the cost must be computed in another way;

or some attractions may offer temporary promotions which may influence the strategy to select them.

The composition of the application may also change. The composer service may decide to provide an alternative bluetooth service to transmit the composed document to a smartphone, less costly than the printer service. In that case, a new component must be added to the composition, and the monitoring and management infrastructure must be changed accordingly.

2.1 Concerns

As it can be seen from the example, concerns about SLA and QoS can be manifold. A monitoring system may be interested in indicators like performance, energy consumption, price, robustness, security, availability, etc., and the set of required values may be different for each monitored service. Plus, not only the values of these indicators change at runtime (that is quite obvious), but also the set of required indicators, as the monitoring requirements can also evolve because new components are involved in the service, or because the SLA has evolved. Moreover, due to the heterogenous nature of the providers, some of the services may require specific protocols to retrieve monitoring values or to perform modifications on them.

In general, the evolution of the SLA and the required metrics can not be foreseen at design time, and it is not feasible to prepare a system where all possible monitorable conditions are ready to be monitored. Instead, it is desirable to have a flexible system where only the required set of monitoring metrics are inserted and the required conditions checked, but as the application evolves, new metrics and SLA conditions may be added and others removed minimizing the intrusion of the monitoring system in the application.

2.2 Contribution

We argue that a component-based approach can tackle the dynamic monitoring and management requirements of a composed service application. We propose a component-based framework to add flexible monitoring and management concerns to a running component-based application.

In this proposition we separate the concerns involved in a classical autonomic control loop (MAPE) and implement those concerns or a set of them as separate components attached to each managed service, in order to provide a custom and composable monitoring and management framework. The framework allows to build distributed monitoring and management architectures that are associated to the actual functional components in an integrated way. Our framework leverages the monitoring and management features of each service to provide a common ground where monitoring, SLA checking/analysis, decisions, and actions can be taken by different components of the framework and be added or replaced separately.

The system relies on a distributed hierarchical and reconfigurable component-based approach attached to each service that composes the application.

2.3 Related Work

Several works exist regarding monitoring and management of service-oriented applications, where most of them tackle separately concerns like monitoring infrastructures [18], SLA monitoring [14], SLA fulfillment [12], and planning strategies for adaptation [7, 8, 11]. A few others, like us, propose a complete framework. The work [3] is similar to ours in that they propose a generic context-aware framework that separates the steps of the MAPE control loop to provide self-adaptation; they focus in the implementation of the self-adaptive strategies, whereas we are interested in the runtime replacement of the elements that handle the monitoring and SLA requirements as well the planning strategy, and we do not necessarily consider that all services require the same level of autonomicity; also we take full benefit of the business-level components distribution and hierarchy to split the implementation of monitoring and management requirements across different levels, thus enforcing scalability.

[9] proposes a hierarchical monitoring of SLAs with support for event-based communication, pull/push modes and different kinds of metrics. The monitoring requirements are tightly coupled to the services and accessed through a common interface. Their approach differs with ours in that they do not consider the modification of the monitoring requirements, or even SLAs at runtime (nor do they consider components and possible associated hierarchy as we do, in order to ease monitoring information aggregation).

[13] presents a service-oriented framework for integrating already available autonomic strategies promoting reuse and allowing the runtime modification and collaboration of autonomic tasks. Although we do not describe such specific interactions, our approach allows to implement that kind of collaborative tasks not only at the autonomic management process level, but also for monitoring and SLA analysis.

We believe that the dynamic inclusion and removal of monitoring and management concerns allows (1) to add only the needed monitoring operations (minimizing overhead), (2) to better adapt to the monitoring needs, raised by changing requirements or raised by evolutions of the adaptation strategies, without enforcing a redeployment and redesign of the application, and increasing separation of concerns.

3. DESIGN OF THE COMPONENT-BASED SOLUTION

Our solution relies on the separation of the steps of the classical MAPE autonomic control loop. Namely, we envision separate components for monitoring, SLA analysis, decision taking, and execution of actions. The components are attached to each managed service to provide the required features.

The framework works by monitoring data from each individual service and calculating a set of metrics from them. The list of available metrics is exposed by a *Monitoring* component, and they are used by an *SLA Analysis* component which can read them and check if the specified conditions are being fulfilled or not. In case a condition is not fulfilled, or has some risk of not being fulfilled, a *Decision* component is activated, which must take a decision of some action to perform; for taking a decision, the *Decision* component can ask the *Monitoring* component for more data. Finally, once an action or set of actions is decided on, these are transmitted to an *Executor* component which executes them over the managed service. This general structure is shown for an individual service “A” in Figure 2.

We benefit from the component features to provide a specific implementation of Monitoring and Execution components for each targeted service, integrating them with the means that each service may provide (this is constrained ac-

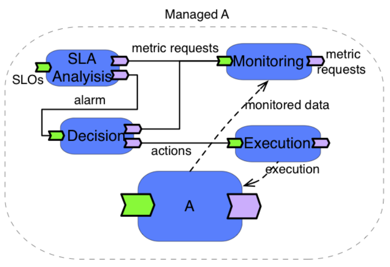

Figure 2: SCA component "A" with all its attached monitoring and management components

tually by the basic monitoring means provided by the service or the middleware it runs over) and leveraging this information to a common interface where they can interact with the other components of the framework.

The hierarchical approach allows to monitor the required conditions only at the level they are relevant to make a decision, avoiding unnecessary propagation.

By distributing the implementation of the monitoring and management capabilities, we avoid centralizing the information gathering, while it is still possible to get monitored data from other services.

We allow to add and remove at runtime different components of the framework, which means that, for example, a service from which no monitoring information is required needs not to have a Monitoring component and may only have an Execution component to modify some parameter of the service. Later, if needed, it is possible to add other components of the framework to this service.

In the following, we describe the components considered in the monitoring and management framework, their function and some design decisions that have been taken into account.

3.1 Monitoring

Monitoring involves the collection of information from the target service (sensing), storage, filtering, and possibly processing (apply functions over the sensed values) of the data to obtain a set of *metrics* that is made available for other components.

The implementation of the *Monitoring* component must include the specific sensors or, alternatively, support the communication with sensors provided by the target service according to a particular protocol. This way the *Monitoring* component is effectively attached to the service, which becomes a "monitored service" from the framework point of view.

In the presence of a high number of services or entities to be monitored, the computing and storage of collected information can be a high-demanding task, specially if it is done in a centralized manner, f.e. gathering the monitored information at a single point of analysis. This can be harmful to the monitored application; consequently, the monitoring task must be as decentralized and low-intrusive as possible. Our design considers one *Monitoring* component attached to each monitored service by the specific means that the service may require. This approach is highly decentralized and specialized with respect to the monitored service while exposing a common interface to access the collected data. On the other side, in such a decentralized context, some decisions or metrics may be harder to compute as they may require information of other services: for example, to compute the cost of a composition, the monitor component would require to know the cost of all the services used while serving some request. To tackle this kind of situation, the monitored component is capable of connecting to the monitoring components of other services. The set of *Monitoring* components are inter-connected forming a hierarchy that reflects the composition of the monitored services: this provides a "monitoring backbone" as shown in figure 3 where the metrics collected at each service can flow and can be used by another component, while also avoiding centralization.

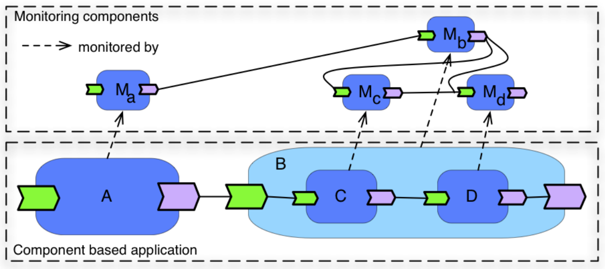

Figure 3: Monitoring layer for an SCA application

The input data for this component is a (big) amount of sensed information related to the target service, which can be collected in different ways: interception of messages, event listeners, hardware-dependant sensors, etc. As a result, the *Monitoring* component exposes an interface to get the value of the metrics stored. The method for computing the metrics can be very simple, or arbitrarily complex (a Complex Event Processing engine), but the principle remains in receiving a set of information, and compute a set of meaningful or processed output set from them.

The monitored data may be exposed in a *pull* mode, where the interested component asks explicitly for a value, or in a *push* mode, where the component can ask for a metric and receive a periodic update, or an update each time that the value changes. Depending on the circumstances and the nature of the monitored value, one mode or the other may be convenient.

3.2 SLA Analysis

The *SLA Analyzer* component checks the compliance to a previously defined SLA. An SLA is defined as a set of simpler *Service Level Objectives* (SLOs), which must be verified at runtime.

The conditions to be checked may be very simple ones, as triples $\langle metric, comparator, value \rangle$, expressing for example "$avgResponseTime \leq 5sec$"; or more complex expressions involving other metrics or operations on them like "$cost(B) < 2 \times cost(C)$", where the amount of energy used by different services is required.

There is an important challenge in the way of expressing the desired condition, as we are passing from a "humanly specified" condition that must be converted to a machine understandable condition. For this matter a domain-specific language (DSL) can be used to describe the condition. Also an XML description (like WSLA) can be used. In any case a parsing component specialized in reading this description and creating the desired SLO must be used.

The *SLA Analyzer* requests the value of the metrics it needs from the *Monitoring* component, as exemplified in figure 4. It can request the values as he needs them, or subscribe to them and receive update notifications, so to analyze immediately the stored conditions.

As input, the *SLA Analyzer* receives a set of conditions (SLOs) to monitor. The *SLA Analyzer* checks the compliance of all the stored SLOs according to the metrics reported by the *Monitoring* component. The *SLA Analyzer* checks if the SLA is being fulfilled, and if not, it sends an alarm notification. The consequences of this alarm are out of the scope of the *SLA Analyzer* and the destination can be a human agent (in the form of an email/SMS), or another component implementing a reacting strategy (see Sec. 3.3). The *SLA Analyzer* may also be configured in a proactive way to detect not only SLA violation, but also foreseeable SLA violations, which may be more useful in some contexts, as it can allow to take actions to prevent an SLA violation [12].

By having the *SLA Analyzer* attached to each service, the conditions can be checked closely to the monitored service and benefit of the hierarchical composition. This way, the services do not need to take care of SLAs in which they are not involved.

Finally, some conditions can be deduced by checking compliance from the provider and from the consumer point of view. For example, while monitoring the response time of a service, the provider may detect locally a given response time of 1 sec., whereas the monitoring of the same request on the consumer side may find a response time of 5 sec. The SLA Analyzer on the consumer side can request the response time to the Monitoring component on the provider side and find that there is an incurred latency of 4 sec.

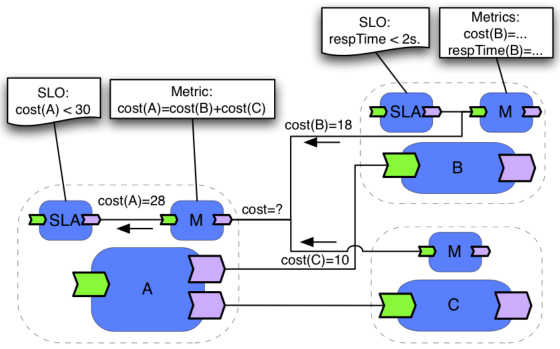

Figure 4: SCA Components with SLA and Monitor components. *A* and *B* have different SLAs. Metric *cost* is computed in *A* by calling the monitors of *B* anc *C*.

3.3 Decision

The *Decision* component implements the strategy defined for reacting to an alarm notified by the *SLA Analyzer* component. The implemented logic can be a very simple strategy like changing the parameter of a service, or replacing one service for another service selected from a list; or a more complex strategy that requires collecting perfomance metrics of other components in order to select a subset of them that maximizes an objective function.

As input, this component receives a notification from the *SLA Analyzer* component indicating that some condition is not (or may not be) fulfilled. The decision component executes the implemented strategy and generates a sequence of actions to take the application to an objective state. If required, it can use the *Monitoring* Component to request certain metrics.

The list of actions, once again, can take a very simple form (a shellscript or an individual action) or a more complex one, as a list of actions described in a domain-specific language that can be interpreted or executed by a set of actuators.

This encapsulation allows the framework to replace at runtime the strategy to reach the objective, for example from a cost-optimizing planner to an energy-efficiency planner, or well taking no action at all.

3.4 Execution

The *Execution* component carries on the decided modifications to the service, or the set of services, as indicated by a *Decision* component.

The execution requires an integrated means to access the managed service in order to execute the actions upon it. In a similar way to the *Monitoring* component, the *Execution* component must implement any specific protocol required by the managed service or middleware to be adapted or reconfigured.

The set of actions demanded may involve not only the managed service, but also different service(s). For this, the *Execution* component is also able to communicate with *Execution* components attached to some other components and send actions to them as part of the main reconfiguration action. The set of connected *Execution* components forms an "execution backbone" that propagates the actions from the component where the actions have been generated to each of the specific components where some part of the actions must take place, possibly hierarchically down to their respective inner components. This approach allows to decentralize the execution of the actions.

One of the challenges is to ensure that the action will not make the application enter in an unsafe state. This problem is left to the execution implementation.

4. IMPLEMENTATION

We have prototyped a framework along the design presented in section 3, for the GCM platform. This SCA compatible platform bases upon the GCM/ProActive middleware, which enables large-scale grid/cloud deployment of components, support for heterogeneous communication protocols between their hosting SCA runtimes, and reconfiguration capabilities of the SOA component-based application. We describe each of the pieces of the framework and exemplify how they can be used in the context of the scenario described in section 2.

4.1 SCA compliant GCM/ProActive

The ProActive Grid Middleware [1] is a Java middleware which aims to achieve seamless programming for concurrent, parallel and distributed computing, by offering an uniform active object programming model, where these objects are remotely accessible via asynchronous with futures method invocations. Active Objects are instrumented with MBeans which provide notifications about events at the implementation level, like the reception of a request, and the start and end of a service. The notification of such events to interested third parties is provided by an asynchronous and grid enabled JMX connector.

The Grid Component Model (GCM) [4] is a component model for applications to be run on computing grids, that takes the Fractal component model [6] as a base for its specification. Fractal defines a component model where components can be hierarchically organized, reconfigured, and controlled offering functional server interfaces (on the left side of component A in figure 5) and requiring client interfaces (on the right side of component A in figure 5). GCM extends that model providing to the components the possibility to be remotely located, distributed, parallel, and deployed in a grid environment, while adding collective communications (multicast and gathercast). In GCM it is also possible to have a componentized membrane [5] that allows the existence of non-functional (NF) components, also called *component controllers*, which take care of the non-functional concerns. NF components can be accessed through NF server interfaces (top of component A, in figure 5), and components can make requests to NF services using NF client interfaces (bottom of component A, in figure 5).

The use of NF components allows to have a more flexible control of NF concerns and to develop more complex implementations as they can be bound to other NF components within a regular component application. In this sense, this paper complements some previous ones about componentized membranes in general [5, 4, 2] as it particularly addresses concerns that pertain to SLA monitoring.

GCM/ProActive is the reference implementation of GCM, within the ProActive suite, where components are implemented by Active Objects. The GCM/ProActive platform provides asynchronous communications between bound components through GCM bindings. This particular communication semantics has been taken into account when implementing the prototype of our framework. GCM bindings can also be used to connect to other technologies and communications protocols, like Web Services, by implementing the compliance to these protocols via specific controllers in the membrane.

Specific controllers have also been defined to allow GCM to act as an SCA compliant platform, in a similar way as achieved by the SCA FraSCAti [17] platform, which however bases upon non distributed components (Fractal/Julia) in contrary to GCM ones.

4.2 Framework Implementation

The framework is implemented in the GCM/ProActive middleware, as a set of NF components that can be added or removed at runtime to the membrane of any GCM component, which becomes a monitored service of the SCA based application. The ability of reconfiguring the composition of the membrane at runtime is provided by the middleware [5].

We have designed a set of predefined components that implement each one of the elements we have described in section 3, and whose specific functionality can be adapted to the monitoring and management needs of the application. The general view of the framework implemented for a single GCM component is shown in Figure 5 (using the GCM graphical notation [4]). The framework is weaved in the primitive GCM/SCA component *A* by inserting NF components in its membrane. Monitoring and management features are exposed through the NF server interfaces "Actions", "SLA Config" and "Monitoring Service". The NF components, as well, can communicate with the monitoring and management features of other GCM/SCA components through the NF client interfaces "Actions" and "Monitoring".

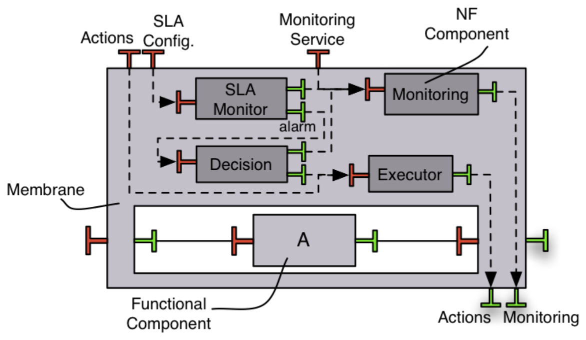

Figure 5: Framework implementation weaved to a primitive GCM/SCA component *A*

4.3 Monitoring

We have designed a set of event detectors and sensors, specific to the GCM/ProActive platform [15]. Over them, we provide a basic *Monitoring* component which includes (1) an *Event Listener* which listens to the events of a GCM component and provides a common ground to access them; (2) a *Record Store* to store records of monitored data that can be used for later analysis; (3) a *Metric Store* which stores objects that we call Metric, which actually compute the desired metrics using the records stored, or the events caught; and (4) a *Monitor Manager* which provides the interface to access the stored metrics, and add/remove them to/from the Metrics Store. This composition is shown in the right side of Figure 6.

The value of each metric may be collected in *push* or in *pull* mode. In *push* mode, the manager inserts a new metric in the *Metric Store* and subscribes it to a set of events needed to compute it; when new data is available from the *Event Listener*, it is transmitted to the *Metric Store* and the metric is updated. For example, the metric *avgRespTime* of a service can be updated upon every completed request. Under the *pull* mode, no subscription is made; instead, the interesting events or sensed values are stored as records in the *Record Store*, and retrieved when the *Monitoring Manager* receives a request for a specific metric. For example, the same metric *avgRespTime* can be computed in a *pull* mode by reading the records stored for the served requests.

The *Monitor Manager* receives a Metric, which in our case is a Java object with a "compute" method, and inserts it in the *Metric Store*. The *Metric Store* provides to the Metrics the connection to the sources that they may need; namely, the *Record Store* to get already sensed information, the *Event Listener* to receive sensed information directly, or the *Monitoring* component of other external components, allowing access to the distributed set of monitors (i.e. to the monitoring backbone).

A simple *respTime* metric can be used to compute the response time of the last request served by the monitored component. This metric is subscribed to a middleware event that is generated every time a request has been served.

As a more complex example, the Tourism Planner composite may need to know the decomposition of the time spent while serving a specific request r_0. For this, a metric called *requestPath* for a given request r_0 is required, which computes the ordered set of all services involved while serving r_0. The *requestPath* metric uses the local records to find out which components were called while serving r_0, and invokes the *Monitoring* component of each one of these services to get *requestPath* of r_0 from them; when no more calls are found, the composed path is returned together with the value of the *respTime* metric for each one of the services involved in the path. Once the information is gathered in the *Monitoring* component of the Tourism Planner, the complete path is built and it is possible to identify the time spent in each service.

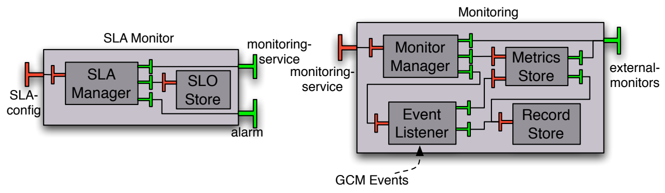

Figure 6: Internal Composition of the Monitoring component (right) and the SLA Monitor component (left)

4.4 SLA Monitor

The *SLA Monitor* is implemented as a component that queries the *Monitoring* component. The *SLA Monitor* consists in (1) an *SLA Manager* which exposes an interface that allows to add/remove SLOs expressed in a specific format, checks the fulfillment of the SLOs, and sends a notification when some of them are not fulfilled; (2) an *SLO Store* which maintains the list of SLOs. The composition is shown in the left side of Figure 6.

An SLO is described as a triple $\langle \mathit{metricN}, \mathit{comparator}, \mathit{fixedVal} \rangle$, where *metricN* is the name of a metric that can be queried through the *Monitoring* component. The SLA Monitor subscribes to the *metricN* from the *Monitoring* component, so it can get the updated values and check the compliance of the SLO.

For example, the Tourism Planner service includes the SLO: "All requests must be served in less than 30 secs", which can be described as $\langle \mathit{responseTime}, <, \mathit{30} \rangle$. The *SLA Manager* receives this description and sends a request to the *Monitoring* component for subscription to the *respTime* metric. The condition is then stored in the *SLO Store*. Each time an update on the metric is received, the *SLA Manager* checks all the SLOs associated to that metric. In case one of them is not fulfilled, a notification is sent, through the *alarm* interface, which includes the description of the faulting SLO.

4.5 Decision

We have implemented a simple decision strategy. The Tourism Planner component wants to ensure that the requests take less than 30 sec., in average, to finish. When the *Decision* component receives an alarm indicating that some request r_0 has not met that SLO, it obtains the request path for r_0 from the Monitoring component including the time spent in each service invoked. The *Decision* component identifies the service that took the most time: if it is bound to a multicast interface (1-to-N communication), then it decides that the service must be unbound from the multicast interface, so that it will not be queried again, unless there is only one service bound, in which case it is not unbound and a notification is sent to a predefined email address. In the example, this strategy is applied w.r.t. the "Attraction" providers.

Clearly this strategy does not intend to be general, but represents an example of how decision strategies added to a component can gather information from the *Monitoring* component and use it to take a decision. Other strategies could involve the replacement of the Map Service if an alternative is provided, or can involve triggering an automated discovery of a similar service.

In general, decision strategies are activated from the *SLA Monitor*, and optionally can use the *Monitoring* component to obtain the data they may require to take a decision. If a decision step is not required anymore it can be removed from the membrane of the component. In this example we consider only a *Decision* component attached to the Tourism Planner component, and not to the other internal services.

4.6 Execution

The *Execution* component uses a domain specific language called PAGCMScript, an extension of the FScript [10] language (designed for Fractal components), which supports GCM specific features like distributed location of components, inter-component collective communications, and addition/removal of components.

The *Execution* component receives actions from the *Decision* component as a set of PAGCMScript sentences and executes them using an embedded PAGCMScript engine.

In the example, once the "Attraction2" provider has been selected, it can be unbound from the "Tourism" composite using a PAGCMScript command like:

```
unbind-unit($tourism/interface::"attractions",
$attractionX/interface::"service")
```

4.7 Generalization

As GCM being an SCA compliant platform, the GCM-based framework can be described in SCA terms providing a view which can be realized for any SCA runtime platform.

However, in our implementation, we profit of the inclusion of components in the membrane for managing the non-functional concerns of the application, in this case, the monitoring and management concerns. This approach allows us to separate the implementation of both concerns, as the programmer of the business code does not need to include monitoring and management hooks in the code to enable the interaction with our framework. Upon (and after) deployment, the application can be enhanced with monitoring and management capabilities, which can also be removed if necessary.

From a general point of view this set of non-functional components can be viewed as a set of additional SCA components that are deployed along and according to the hierarchical architecture of the business code. The SCA business components are wrapped inside an SCA composite to include the monitoring and management components and which provides the required interfaces. This equivalency is depicted in Figure 7, focusing on the *Monitoring* component only for better readability (and is similar to the approach of [16], which does not, however, targets runtime reconfiguration of the management layer).

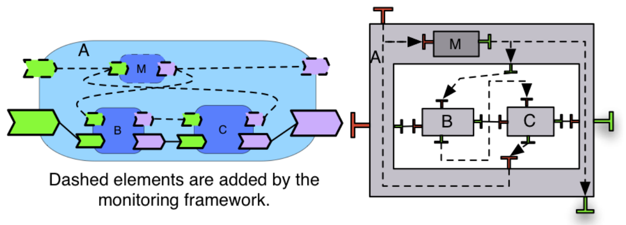

Figure 7: Location of Monitoring component in a composite SCA, and the equivalent location in the membrane of GCM components. Components *B* and *C*, have also associated monitoring components.

5. CONCLUSIONS AND PERSPECTIVES

We have presented a generic component-based framework for supporting monitoring and management tasks of component-based SOA applications. The component based approach allows a clear separation of concerns w.r.t. the management tasks. We have implemented a prototype framework thanks to the development of well identified components needed for monitoring (sensors, event handlers), SLA management and SOA application reconfiguration, that we composed together. This prototype has been developed in the context of an SCA compliant platform that includes dynamic reconfiguration and distribution capabilities.

The flexibility degree of the approach is high as the framework can be personnalized to e.g. support different QoS-aware composition strategies: it is a matter of providing a customized actuator component. The monitoring component can also be easily adapted to the appropriate sensors for providing the input data that these strategies may need (for example, performance, cost, energy consumption, availability). The SLA manager can be configured to implement reactive or proactive strategies, as the triggering of alarms is independent of the QoS-aware composition strategy. The architectural reconfiguration capability allows to modify architectural bindings at runtime as dictated by the composition strategy.

6. REFERENCES

[1] ProActive Parallel Suite. http://proactive.inria.fr/.

[2] M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, M. Danelutto, P. Pesciullesi, R. Ravazzolo, M. Torquati, M. Vanneschi, and C. Zoccolo. A framework for experimenting with structure parallel programming environment design. In G. R. Joubert, W. E. Nagel, F. J. Peters, and W. V. Walter, editors, *Parallel Computing: Software Technology, Algorithms, Architectures and Applications, PARCO 2003*, volume 13 of *Advances in Parallel Computing*, 2004.

[3] F. Andre, E. Daubert, and G. Gauvrit. Towards a generic context-aware framework for self-adaptation of service-oriented architectures. *Intl. Conf. on Internet and Web Applications and Services, ICIW*, 2010.

[4] F. Baude, D. Caromel, C. Dalmasso, M. Danelutto, V. Getov, L. Henrio, and C. Pérez. GCM: a grid extension to Fractal for autonomous distributed components. *Annals of Telecommunications*, 64(1-2):5–24, 2009.

[5] F. Baude, D. Caromel, L. Henrio, and P. Naoumenko. *Post-Proceedings Selected Papers From The Coregrid Workshop On Grid Programming Model, Grid And P2p Systems Architecture, Grid Systems, Tools And Environments, June 2007*, chapter A Flexible Model And Implementation Of Component Controllers. Springer, 2008.

[6] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The Fractal Component Model and its Support in Java. *Software Practice and Experience (SPeE)*, 36, 2006.

[7] G. Canfora, M. D. Penta, R. Esposito, and F. Perfetto. Service composition (re) binding driven by application specific QoS. In *ICSOC 2006, 4th Intl. Conf. on Service-Oriented Computing*, 2006.

[8] V. Cardellini and S. Iannucci. Designing a Broker for QoS-driven Runtime Adaptation of SOA Applications. In *IEEE Intl. Conf. on Web Services (ICWS'10)*, 2010.

[9] M. Comuzzi and G. Spanoudakis. A Framework for Hierarchical and Recursive Monitoring of Service Based Systems. *4th Intl. Conf. on Internet and Web Applications and Services, ICIW*, May 2009.

[10] P.-C. David, T. Ledoux, M. Léger, and T. Coupaye. Fpath and fscript: Language support for navigation and reliable reconfiguration of fractal architectures. *Annals of Telecommunications*, 64, 2009.

[11] C. Ghezzi, A. Motta, V. Panzica, L. Manna, and G. Tamburrelli. QoS Driven Dynamic Binding in-the-many. *QoSA 2010*, pages 68–83, 2010.

[12] P. Leitner, B. Wetzstein, F. Rosenberg, A. Michlmayr, S. Dustdar, and F. Leymann. Runtime prediction of service level agreement violations for composite services. *NFPSLAM-SOC'09. Proc. of the 3rd Workshop on Non-Functional Properties and SLA Management in Service-Oriented Computing*, 2009.

[13] Y. Maurel, A. Diaconescu, and P. Lalanda. CEYLON: A Service-Oriented Framework for Building Autonomic Managers. *7th IEEE Intl. Conf. and Workshops on Engineering of Autonomic and Autonomous Systems*, Mar. 2010.

[14] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar. Comprehensive QoS monitoring of Web services and event-based SLA violation detection. *MWSOC '09. Proc. of the 4th International Workshop on Middleware for Service Oriented Computing*, 2009.

[15] C. Ruz, F. Baude, and B. Sauvan. Enabling SLA Monitoring for Component-Based SOA Applications – A Component-Based Approach. In *36th Euromicro Conf. on Software Engineering and Advanced Applications, SEAA, Work In Progress Session*, 2010.

[16] M. Schmid and R. Kröger. Decentralised QoS-Management in Service Oriented Architectures. In *DAIS*, pages 44–57, 2008.

[17] L. Seinturier, P. Merle, D. Fournier, N. Dolet, V. Schiavoni, and J.-B. Stefani. Reconfigurable SCA Applications with the FraSCAti Platform. In *6th IEEE Int. Conf. on Service Computing (SCC'09)*, 2009.

[18] A. Van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst. Continuous Monitoring of Software Services: Design and Application of the Kieker Framework, 2009. http://se.informatik.uni-oldenburg.de:30000/458/.