Using Components to Provide a Flexible Adaptation Loop to Component-based SOA Applications

Cristian Ruz, Françoise Baude, Bastien Sauvan
INRIA Sophia Antipolis Méditerranée
CNRS, I3S, Université de Nice Sophia Antipolis
France
{*Cristian.Ruz, Francoise.Baude, Bastien.Sauvan*}*@inria.fr*

***Abstract*—The Service Oriented Architecture (SOA) model fosters dynamic interactions of heteregeneous and loosely-coupled service providers and consumers. Specifications like the Service Component Architecture (SCA) have been used to tackle the complexity of developing such applications; however, concerns like runtime management and adaptation are left as platform specific matters. Though several solutions have been proposed, they have rarely been designed in an integrated way and with the capability to evolve the adaptation logic itself. This work presents a component based framework that allows the insertion of monitoring and management tasks, providing flexible autonomic behaviour to component-based SOA applications. Each phase of the autonomic control loop is implemented by a different component, in such a way that different implementations can be developed for each phase and they can be replaced at runtime, providing support for evolving non-functional requirements. We present an illustrative scenario that is dynamically augmented with components to tackle non-functional concerns and support adaptation. We use an SCA compliant platform that allows distribution and architectural reconfiguration of components. Micro-benchmarks and a use case are presented to show the feasibility of our proposed implementation, and illustrate the practicality of the approach. Overall, we show that a component-based approach is suitable to provide autonomic and adaptable behaviour to component-based SOA applications.**

***Keywords*-Monitoring; *Autonomic Management*; *SLA Monitoring*; *Reconfiguration*; *Component-based Software Engineering*.**

I. Introduction

According to the principles of Service Oriented Architecture (SOA), applications built using this model comprise loosely-coupled services that may come from different heterogeneous providers. At the same time, a provided service may be composed of, and consume other services, in a situation where service providers are also consumers. Moreover, SOA principles like abstraction, loosely-coupling and reusability foster dynamicity, and applications should be able to dynamically replace a service in a composition, or adapt the composition to meet certain imposed requirements.

Requirements over service based applications usually include metrics about Quality of Service (QoS) like availability, latency, response time, price, energy consumption, and others, and are expressed as Service Level Objectives (SLO) terms in a contract between the service consumer and the provider, called Service Level Agreement (SLA). However, SLAs are also subject to evolution due to different providers, environmental changes, failures, unavailabilities, or other situations that cannot be foreseen at design time. The complexity of managing changes under such dynamic requirements is a major task that pushes the need for flexible and self-adaptable approaches for service composition. Self-adaptability requires monitoring and management features that are transversal to most of the involved heterogeneous services, and may need to be implemented in different ways for each one of them.

Several approaches have been proposed for tackling the complexity, dynamicity, heterogeneity and loosely-coupling of SOA-based compositions. Notably, the Service Component Architecture (SCA) is a technologically agnostic specification that brings features from Component-Based Software Engineering (CBSE) like abstraction and composability to ease the construction of complex SOA applications. Non-functional concerns can be attached using the SCA Policy Framework. However, monitoring and management tasks are usually left out of the specifications and must be handled by each SCA platform implementation, mainly because SCA is design-time and not runtime focused.

In our previous work [1] we have proposed a component-based approach to ease the implementation of flexible adaptation in component-based service-oriented applications. Our solution implements the different phases of the widely used MAPE (Monitor, Analyze, Plan, and Execute) autonomic control loop [2] as separate components that can interact and support multiple sets of monitoring sources, conditions, strategies and distributed actions.

Our approach gives two kinds of flexibility: (1) we can dynamically inject or remove conditions, sensors, planning strategies, or adaptation actions in the MAPE loop in order to modify the way the autonomic behaviour is implemented in the application; and (2) we can insert or remove elements of the MAPE loop, modifying the composition of the autonomic control loop itself, and making the application more or less autonomic as needed.

In this work we extend the presentation of our component-

based framework detailing the design considerations for each phase of our autonomic control loop and how they provide the flexibility that we expect. We present a concrete use case of an application that is dynamically augmented with autonomic behaviour.

The rest of the paper is organized as follows. Section II presents the example that we use to motivate and illustrate the practicality of our work, and provides a general overview of our contribution. Section III describes the design of our framework from a technologically independent point of view. Section IV presents our implementation over a concrete middleware and component model. Section V shows a practical example of use of our framework and the evaluations we have carried on. Section VI describes related work and differentiations with our solution. Finally, Section VII concludes the paper.

II. Motivating example and Overview of Our Contribution

Consider a tourism office that has composed a smart service to assist visitors who request information from the city and provides suggestions of activities. The application uses a local database of touristic events and a set of providers who sell tickets to museums, tours, etc. A weather service can be used to complement the proposition of activities, and a mapping service creates a map with directions. A payment service is used to process online sells in some cases. Once all information is gathered, a local engine composes a PDF document and optionally prints it. The composed design of the application is shown in Figure 1 using the SCA [3] diagram notation.

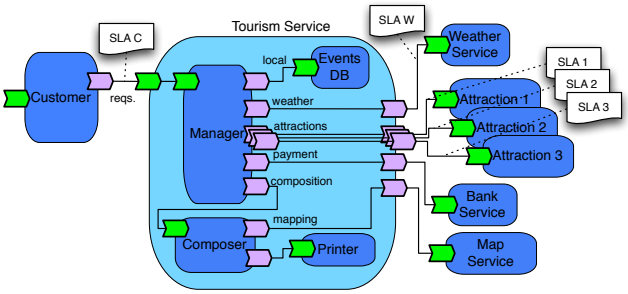

Figure 1. The SCA description of the application for tourism planning scenarios.

Such a composition involves some terms for service provisioning. For example, the Tourism Service agrees to provide a touristic plan within 30 sec.; the Weather Service charges a fee for each forecast depending on the level of detail; the Mapping Service is a free service but has no guarantees on response time or availability; the Payment Service ensures 99% of availability. All these conditions are formally established in several SLAs.

The runtime compliance to the SLAs may influence certain decisions on the composed service. For instance, if the Mapping Service is not reachable at a certain moment or if it takes too much time to deliver a response, then the Tourism Service may provide a touristic plan without maps in order to meet the agreed response time at the expense, however, of a lower quality response (workflow modification). Another situation may happen if the Weather Service increases its costs, thus violating the agreement, then the Tourism Service may decide to replace it for another equivalent cheaper service (service replacement). Finally, if the Printer service is running short on color cartridge, then the Tourism Service may decide to use only black and white printing (parameter modification).

In all these cases the decisions should, ideally, be taken in an autonomic way. This requires to constantly monitor certain parameters of the application and, in order to timely react, an efficient analysis and decision taking process. However, it should not be a task of the programmer of each service to code all these autonomic behaviours. Instead, it is more desirable to compose the autonomic behaviour in a separate way and insert it or remove it from the service activity as needed. Moreover, if an autonomic behaviour requires to collect information from different services, then forcing each service to be explicitly aware of the details of other services would increase the coupling of the services.

Also because of the heterogeneity of the services, 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; in the case of the touristic plan 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 not be possible to add specific monitoring on their side. So, in any case the monitoring capabilities will be limited by the monitoring features available from each service. This situation imposes a requirement for supporting heterogeneous services and adaptable monitoring.

A. 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 for performance, energy consumption, price, robustness, security, availability, etc., and the range of acceptable values may be different for each monitored service. Moreover, not only the values of these indicators may change at runtime, but also the set of required indicators. Also, heterogeneity plays a role at the moment of programming the access to the required values.

In general, the evolution of the SLA and the required indicators 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 conditions may be added and others removed minimizing the intrusion of the monitoring system in the application.

B. Contribution

We argue that a component-based approach can tackle the dynamic monitoring and management requirements of a composed service application while also providing the capability to make the application self-adaptable. 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) [2] and implement those concerns as separate components. These components are attached to each managed service, in order to provide a custom and composable monitoring and management framework. The framework allows distributed monitoring and management architectures to be built in a way that they are clearly associated to the actual functional components. The framework leverages the monitoring and management features of each service to provide a common ground in which monitoring, SLA checking/analysis, decisions, and actions can be carried on by different components, and they can be added or replaced separately.

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

III. Design of the Component-Based Solution

Our solution relies on the separation of the phases of the classical MAPE autonomic control loop. Namely, we envision separate components for monitoring, analysis, planning, and execution of actions. These components are attached to each managed service.

From an external point of view, a regular service A is augmented at design time with a set of additional interfaces. These interfaces define the entry points to the management framework for each service A, which is transformed into *managed service* A, as shown in Figure 2. The management interfaces allow the service to interact with other managed services and take part in the framework; however the services are not forced to provide an implementation of all these management interfaces. Instead, these implementations can be dynamically added.

The general structure of our design is shown for an individual service A in Figure 3. Service A is extended with one component for each phase of the MAPE loop and converted into a *Managed Service* A, indicated by dashed lines. The original "service" and "reference" interfaces of service A are promoted to the corresponding interface of

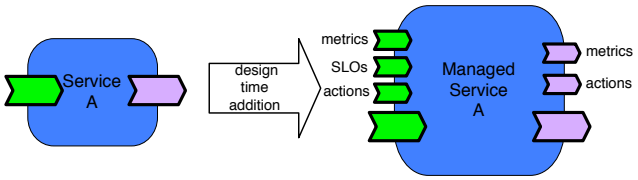

Figure 2. SCA component A extended at design time with management interfaces

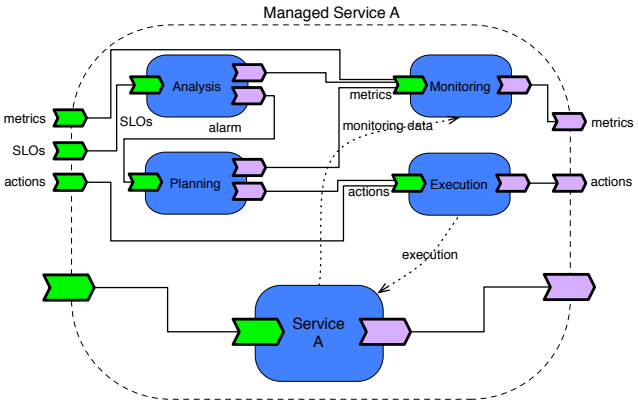

Figure 3. SCA component A with all its attached monitoring and management components

Managed Service A so that, from a functional point of view, the *Managed Service* A can be used in the same way as the original Service A.

The general functioning of the framework is as follows. The *Monitoring* component collects monitoring data from service A using the specific means that A may provide. Using the collected monitoring data, the *Monitoring* component provides access to a set of metrics through the *metrics* interface. The computation of metrics may involve communication with the *metrics* interface of other managed services. The *Analysis* component provides an interface for receiving and storing SLOs expressed as conditions. At runtime, the *Analysis* component checks the SLOs using the metrics that it obtains from the *Monitoring* component. Whenever an SLO is not fulfilled (a faulting condition), the *Analysis* component sends an alarm signal that activates the *Planning* component. The *Planning* component uses a pre-stored strategy to create an adaptation plan, described as a sequence of actions, that will be the response of the autonomic system to the faulting condition. If the adaptation strategy requires additional monitoring information, it can be obtained from the *Monitoring* component. The sequence of actions created by the *Planning* component are sent to the *Execution* component, which executes the actions on the service using the specific means that the service allows and, if needed, it can delegate the execution to the *Execution* component of other services. This way, the loop is completed.

Although simple, this component view of the autonomic control loop has several advantages.

- First, by separating the control loop from the component implementation, we obtain a clear **separation of concerns** between functional content and non-functional activities; meaning that the programmer of the application does not need to explicitly deal with management activities or with autonomic behaviour.
- Second, the component-based approach allows separate implementations to be provided for each phase of the loop. As each phase may require complex tasks, we abstract from their implementation, that may be specific for each service, and allow them to interact only through predefined interfaces, so that each phase may be implemented by different experts.
- Third, as each phase can be implemented in a separate way, we may consider components that include, for example, multiple sensors, condition evaluators, planning strategies, and connections to concrete effectors as required. This way we allow multiple autonomic control loops running over the same system, taking care of different concerns.

Regarding the genericity or the approach we have described it in a way as technology-independent as possible. However, every implementation that intends to manage a concrete service has, at some point, to use the specific means that the service admits either for obtaining information from it, or for modifying it. Our design is generic until the point that we must define the concrete sensors and actuators that must interact with the managed service. Actually, the amount of information that we can collect from the service and the kind of actions that we can execute over it, will be limited by the methods that the service makes available. We consider, however, that this limitation is given by the technology that provides access to the services (in this case, a component middleware) instead of the service programmer itself. In Figure 3, the service implementation dependent parts are indicated by the dashed arrows between the Service A and its respective *Monitoring* and *Execution* components.

The framework allows the addition and removal at runtime of different components of the loop, which means that, for example, a service that does not need monitoring information extracted, does not need 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. This way, a service may be modified at runtime to have a major or minor level of autonomicity according to the needs.

As a simple example, consider a component that represents a storage service, and provides some basic operations to read, write, search and delete files. In order to get information about the performance of the storage service, a *Monitoring* component can be added and expose metrics about the average response time for each operation, and the amount of free space. As an evolution, some non-functional maintenance actions can be exposed to compress, index, or tune the periodicity of backups. These actions can be exposed by adding an *Execution* component that can execute them over the storage service. Now the managed storage service exposes some metrics, and exposes an interface for executing maintenance actions. However, the storage service is still not autonomic and the reading of metrics and execution of maintenance actions are invoked by external entities. A next evolution can consider adding an autonomic behaviour to avoid filling the capacity of the storage service. An *Analysis* component can be added and include a condition that checks the amount of free space, and in case it is less than, for example, 2%, it triggers an action oriented to increase the amount of free space. The decision about what action to take can be delegated to a *Planning* component, which will create the list of actions to be carried on by the *Execution* component.

Depending on the management needs, any evolution of the storage service can be used. If the autonomic behaviour described is not needed anymore, then the *Analysis* and *Planning* components can be removed and return to the simple version of the storage service. The three versions mentioned of the storage service are shown in Figure 4.

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.

A. Monitoring

The *Monitoring* task consists of collecting information from a service, and computing a set of indicators or *metrics* from it. The *Monitoring* component includes sensors specific for a service or, alternatively, supports the communication with sensors provided by the target service. This way, the *Monitoring* component can be effectively attached to the service.

In the presence of a high number of services, the computing and storage of metrics can be a high-demanding task, specially if it is done in a centralized manner. Consequently, the monitoring task must be as decentralized and low-intrusive as possible. For this, our design considers one *Monitoring* component attached to each monitored service, that collects information from it, and exposes an interface to provide the computed metrics. This approach is decentralized and specialized with respect to the monitored service. On the other side, some metrics may require additional information from other services: for example, to compute the cost of running a composition, the *Monitoring* component would require to know the cost of all the services used while serving some request. To address this situation in a decentralized way, the *Monitoring* component is capable

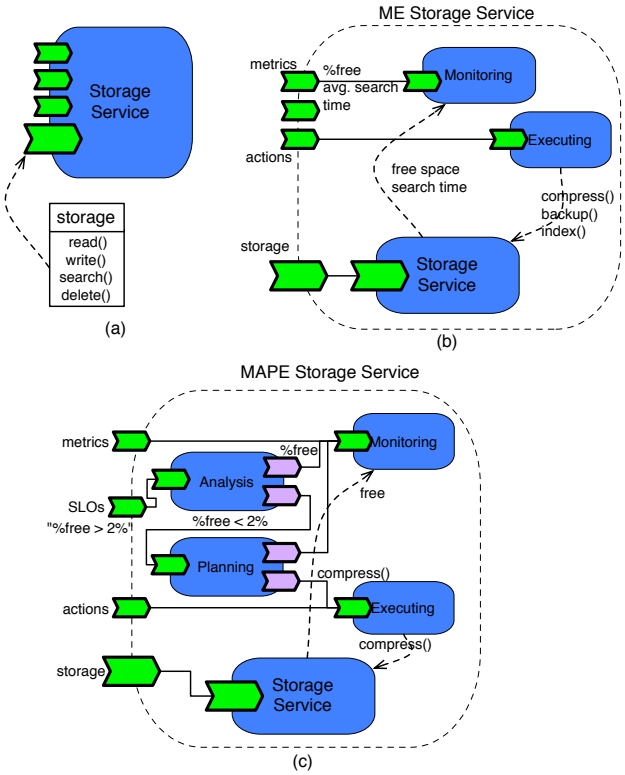

Figure 4. (a) Storage service in its basic version, (b) with Monitoring and Executing components, (c) with all the MAPE components and providing an autonomic behaviour

of connecting to the *Monitoring* components of other services. The set of *Monitoring* components are inter-connected forming an architecture that reflects the composition of the monitored service and forming a "monitoring backbone" as shown in Figure 6.

Figure 5 shows the methods of the *metrics* interface. Metrics are referenced by a *metricName* string. The method *getMetric(metricName)* is used by another component, or by an external tool to fetch the current value of the metric *metricName* in a *pull* mode. It is also possible to read the values in a *push* mode by using the *subscribe(metricName)* and *unsubscribe(metricName)* methods, so that the *Monitoring* component notifies the receptor of any changes in the value. The method *getMetricList()* allows the caller to verify which metrics are available from the *Monitoring* component, and the *insertMetric(metric, metricName)* and *removeMetric(metricName)* methods allow the caller to manipulate the available metrics by inserting or removing the code that actually computes the values. An actual implementation of this interface is permitted to extend it as needed.

Figure 6 shows an example of a *metric* named "energy consumption" ($e(i)$) for each component i. Each *Monitoring* component M_i is in charge of computing its value $e(i)$ as the sum of its own energy metric, and those of its references. In the case of the composite service C, the value $e(C)$ is

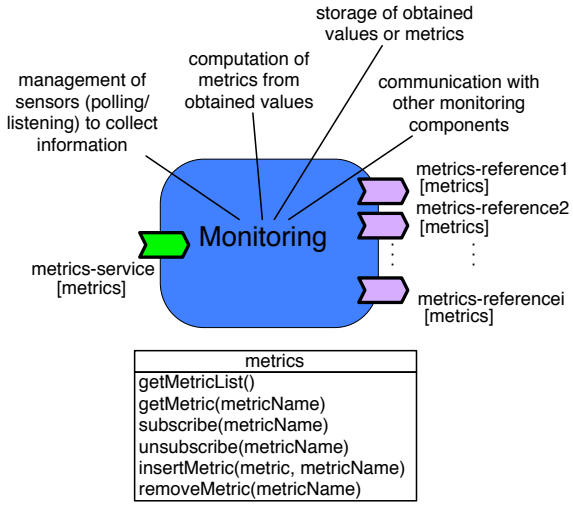

Figure 5. The *metrics* interface of the *Monitoring* component

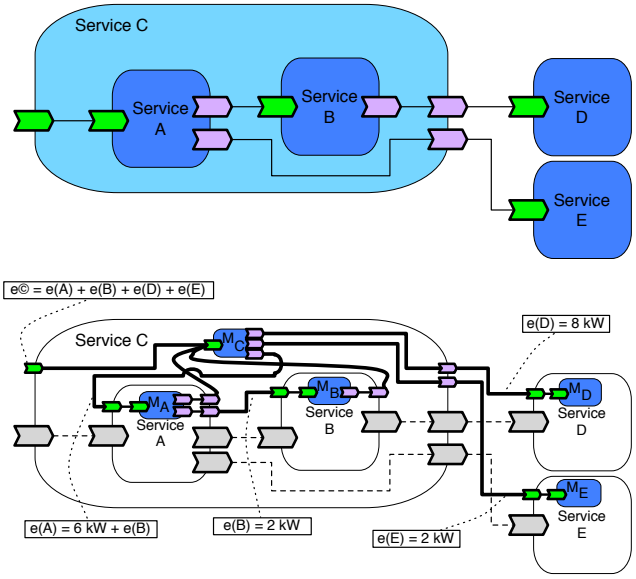

Figure 6. An SCA application, and the inner "monitoring backbone"

the sum of the values of both internal components, $e(A)$ and $e(B)$, and of its references $e(D)$ and $e(E)$. Using the connection between the different *Monitoring* components, the total value $e(C)$ is computed by M_C and exposed through its *metrics* interface. Note that the means for computing the energy metric for each component may be different, depending on the characteristics of the implementation; however, once the value is computed in the corresponding *Monitoring* component, it becomes accessible in a uniform way by the other *Monitoring* components.

Figure 6 also shows a characteristic of our design with respect to the number of monitoring interfaces. In order to connect to monitoring interfaces of other components, each *Monitoring* component includes one reference to the

Monitoring component of each component to which the managed component is bound. This is done so that we can properly identify the monitoring information coming from each managed component. It is possible to see, for example, that M_C includes three references: one for communicating with M_A because the service interface of Service C is bound to Service A; and two reference interfaces for M_D and M_E because Service D and Service E are referenced by Service C. In this particular case, M_C is not bound to M_B because its service interface is not bound to any service interface of Service C.

B. Analysis

The *Analysis* component checks the compliance to a previously defined SLA. An SLA is defined as a set of simpler terms called SLOs, which are represented by conditions that must be verified at runtime.

One of the challenges of the *Analysis* component is to be able to understand the conditions that need to be checked. There exist several languages proposed for representing SLOs and the metrics they require [4], [5], [6], [7]. Using a component-based approach inside the *Analysis* component it should be possible to embed an interpreter for these languages into the *Analysis* component.

For illustrative purposes, we can consider a very simple description of conditions using triples $\langle metric, comparator, value \rangle$ expressing, for instance, "$respTime \leq 30sec$"; or more complex expressions involving other metrics or operations on them like "$cost(weatherService) < 2 \times cost(mappingService)$", where the metrics used by different services are required.

The *Analysis* component obtains the values of the metrics it needs from the *Monitoring* component and, thanks to the interconnected *Monitoring* components, it can obtain metrics from other services as well.

The *Analysis* component receives a set of conditions (SLOs) to monitor through the *SLOs* interface, and it checks the compliance of all the stored SLOs according to the metrics reported by the *Monitoring* component. In case some SLO is not fulfilled, the *Analysis* component sends an alarm notification through a reference *alarm* interface. The consequences of this alarm are out of the scope of the *Analysis* component and will be mentioned in the next section.

The *Analysis* component can also be configured in a proactive way to detect SLA violations not only after they happened, but instead to generate the alarm before the violation happens (with a certain probability). This predictive capability may be useful in many contexts, as it can avoid incurring into penalties as a consequence of the ocurrence of the violation [8]. Of course, a tradeoff between the precision of the prediction and the cost of the prevention must be made.

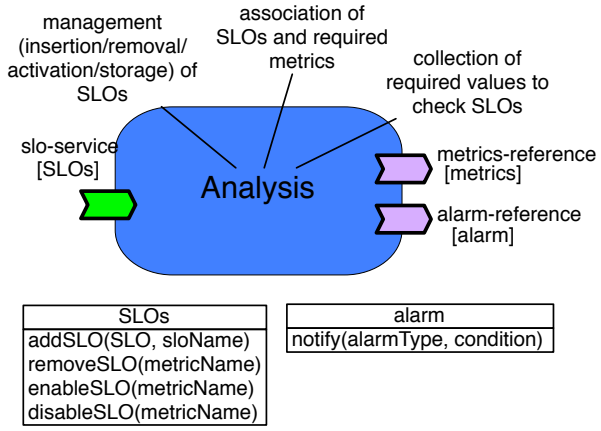

Figure 7. The SLOs interface of the *Analysis* component

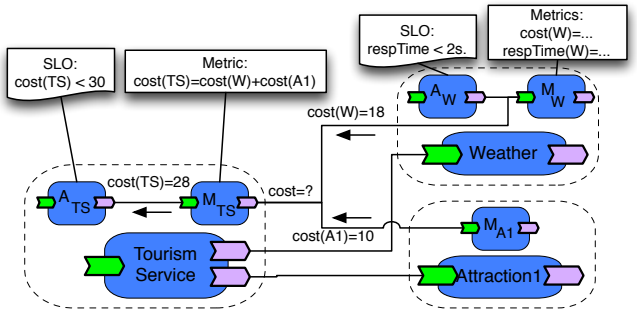

Figure 8. SCA components with Analysis (A_i) and Monitor (M_i) components. *Tourism Service* and *Weather* have different SLAs. The metric *cost* is computed in *Tourism Service* by calling the monitors of *Weather* and *Attraction1*.

By having the *Analysis* component 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.

Figure 7 shows the methods of the *SLOs* interface. The methods allow the caller to manipulate the list of SLOs that are checked by the *Analysis* component by inserting or removing the object that contains the SLO description and referencing it through the *sloName* string. The enable/disable methods permit the caller to enable or disable the verification of a particular SLO. The precise manner in which the *Analysis* component reads and stores the SLO objects, checks the compliance of the SLOs, and obtains the information from the *Monitoring* component are left as an implementation concern. One way to implement it is described in Section IV-D.

Figure 8 shows an example where Service *TourismService (TS)* has an *Analysis* component A_{TS}, and a *Monitoring* component M_{TS}; services *Weather (W)* and *Attraction1 (A1)* are referenced by *TS*. Service *W* includes an *Analysis* component A_W and a *Monitoring* component M_W; service

A1 only includes a *Monitoring* component M_{A1}.

The *Analysis* component of *TS* must check the SLO "$\langle cost, <, 30\rangle$" over Service *TS*. For checking that condition, it requires the value of the metric *cost* from M_{TS}. In M_{TS}, the computation of the metric *cost* requires the value of the metric *cost* from both services W and $A1$. M_{TS} obtains this information from the corresponding *Monitoring* components M_W and M_{A1} and is able to deliver the response to A_{TS}. It is worth noting that A_{TS} is not aware that the computation of M_{TS} actually required additional requests to M_W and M_{A1}, as this logic is hidden into M_{TS}. At the same, the *Analysis* component A_W works independently to check a condition related to the response time (*respTime*) metric from service W, which requires to read the appropriate metric from M_W.

C. Planning

The objective of the *Planning* phase is to generate a sequence of actions, called *plan*, that can modify the state of the service in order to restore some desired condition. In general, we want to restore the condition (the *SLO*) that has been violated.

The computation of a plan is triggered when a notification is received indicating that a condition is not being fulfilled, through the *alarm* interface. For creating such a plan, the *Planning* component must execute a planning algorithm that can determine that sequence of actions. This logic can be implemented in a number of ways. On the more simple side, a strategy may be a notification to a human agent (email, SMS, etc.) who would be responsible of taking any further action; another alternative could rely on a table of predefined actions, like ECA (Event-Condition-Action) triggers, such that if some conditions hold, then the corresponding action is generated. On a more complex side, numerous strategies and heuristics, in particular from the artificial intelligence area have been proposed for planning a composition or recomposition of services that complies with certain desired QoS characteristics. The aim of our *Planning* component is to be capable of supporting the implementation of such existing strategies.

The *alarm* interface is shown in Figure 9. It only considers one method *notify(alarmType, condition)* that includes the condition that is triggering the reaction, and optionally a level indicator called *alarmType* that permits the caller to assign priorities or levels of gravity of the notification.

Given the wide range of different solutions for generating a plan, it does not seem easy to find an interface that is uniform across all the possible strategies. However, most of the strategies require as input information the current state of the service in order to guide the possible solutions. Consequently, our *Planning* component considers one interface for obtaining information about the state of the service, connected to the *Monitoring* component.

Although a simple implementation would embed only one specific strategy, our approach considers that several

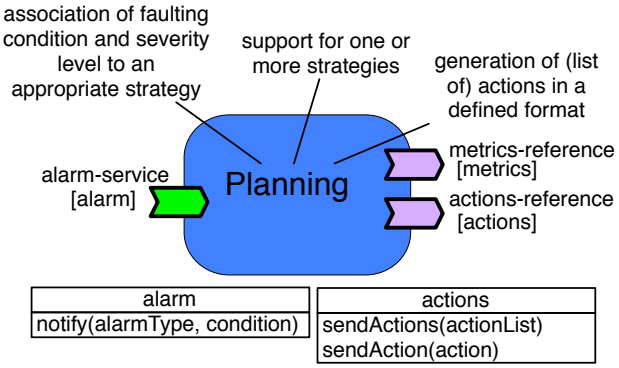

Figure 9. The *alarm* interface of the *Planning* component

conditions may be supported by the *Analysis* component. Consequently, several conditions may need to be checked and, if it is necessary to take some actions, different strategies may be applied upon each case. That is why we think that a component-based approach applied to the *Planning* component should be able to support different planning strategies that would be activated depending on the condition that needs to be restored.

It is also a concern that these strategies may be replaced at runtime. For example, an application may be driven by a cost-saving strategy and, at some point the administrator may need to change the requirements and enforce an energy-saving strategy. In that case, a replacement of the corresponding strategy should be performed inside the *Planning* component. However, this task is not an autonomic task of the framework itself and is, instead, driven by an administrator of the management layer.

Figure 10 shows an example where service *Tourism-Service (TS)* uses two services *Weather (W)* and *Mapping (MP)*. The *Planning* component of *TS*, P_{TS} receives an alarm from the *Analysis* component A_{TS} indicating that the condition $\langle cost, <, 30\rangle$ has been violated, and that an action should be taken. P_{TS} executes a very simple strategy, which intends to replace the component with the higher cost. For obtaining the *cost* of both components *W* and *MP*, P_{TS} uses the *Monitoring* component M_{TS}, which communicates with M_W and M_{MP} to obtain the required values. As *MP* has the higher cost, the strategy determines that this component must be replaced. P_{TS} uses an embedded reference to a discovery service, to obtain an alternative service, called *MX*, which provides the same functionality as *MP* (this is neccesary to not interfere with the functional task of the application) and whose *cost* is expected to satisfy the condition $\langle cost, <, 30\rangle$. With all this information, P_{TS} is able to produce a single action $replace(MP, MX)$ as output.

It is worth to notice that all the logic of the planning algorithm is encapsulated inside P_{TS}, and that M_{TS} is only used to obtain the values of the metrics that the strategy may need.

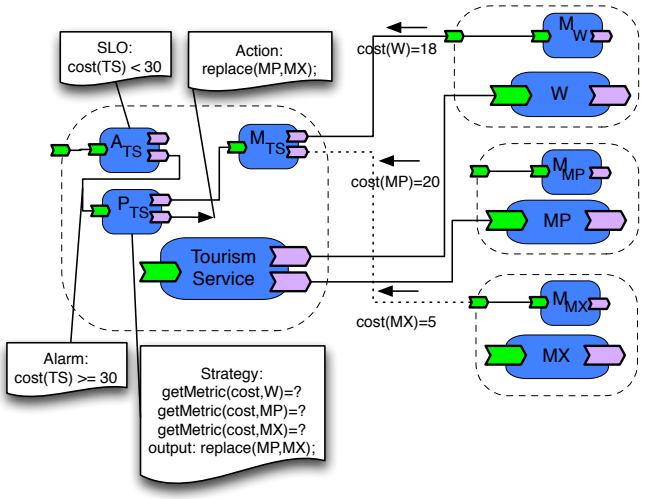

Figure 10. Example for the Planning component.

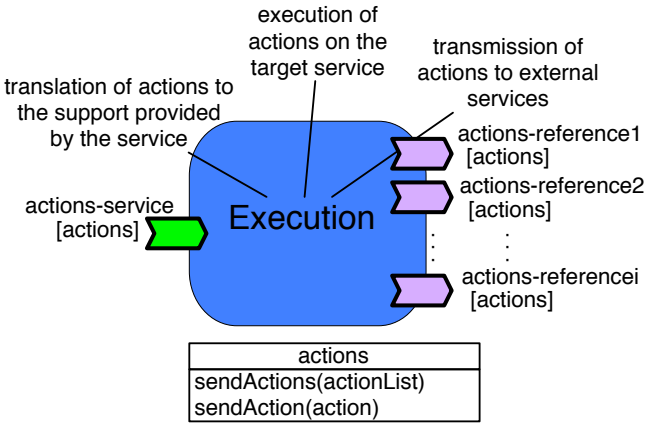

Figure 11. Example for the Execution component.

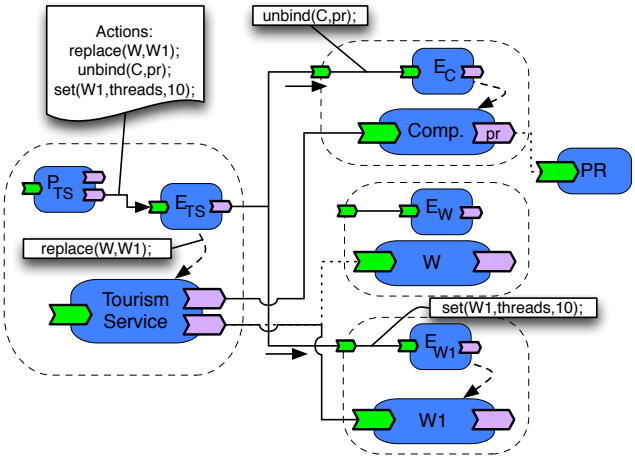

Figure 12. Example of propagation of actions through *Execution* components

D. Execution

The *Execution* component carries out the sequence of actions that have been determined by the *Planning* component.

Although it seems reasonable that once the actions have been decided, those be executed immediately, the *Execution* component has more importance than just executing actions. One of the reasons for having a different component is to separate the description of the actions from the specific way to execute them. In the same sense that the *Monitoring* component abstracts the way to retrieve information from the target service and provides a common interface to access the metrics it collects, the *Execution* component abstracts the communication with the target service to provide a uniform way to execute actions on the service. This also implies that, like the *Monitoring* component, the *Execution* component must be implemented according to the specific characteristics of the service on which the actions must be executed.

The set of actions demanded may involve not only the managed service, but also different services. For this reason, the *Execution* component is also able to communicate with the *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 distribute the execution of the actions.

The *Execution* component receives the sequence of actions to execute from the *actions* interface, which is shown in Figure 11. The interface has two methods that permit the caller to send either a list of *action*s, or a single *action* to the *Execution* component. The proper definition of the *action* object will depend on the implementation. In any case, the *Execution* must be able to read this object and interpret it as an action that can be executed on the service.

Figure 12 shows an example where three actions are generated by the *Planning* component of *TourismService (TS)*: one to replace the service *Weather (W)*, one to unbind the service *Printer (PR)*, and the third one to set a parameter on the reference to service *Mapping (MP)*. In the example, the *Planning* component P_{TS} has sent the list of actions to the *Execution* component E_{TS}. The action of replacing component *W* by W_1 is executed locally at *TS*. However, the unbinding of reference *pr* on service *Composer (C)* must be executed by E_{TS}; and the setting of the parameter "threads" on service W_1 must be executed by E_{W_1}. By using the connections between the different *Execution* components, the actions can be delegated to the appropriate place.

IV. Implementation

This section describes our prototype implementation over a middleware that implements a particular component model. We describe the pieces of the framework that have been implemented according to the design guidelines presented in Section III and exemplify how they can be used to provide self-adaptability in the context of the scenario described in Section II.

A. Background: GCM/ProActive

The ProActive Grid Middleware [9] 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 method invocations and futures. 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 [10].

The Grid Component Model (GCM) [11] is a component model for applications to be run on computing grids, that extends the Fractal component model [12]. Fractal defines a component model where components can be hierarchically organized, reconfigured, and controlled offering functional server interfaces and requiring client interfaces (as shown in Figure 13). GCM extends that model providing to the components the possibility to be remotely located, distributed, parallel, and deployed in a grid environment, and adding collective communications (multicast and gathercast interfaces). In GCM it is possible to have a componentized membrane [13] that allows the existence of non-functional (NF) components, also called *component controllers* that take care of non-functional concerns. NF components can be accessed through NF server interfaces, and components can make requests to NF services using NF client interfaces (shown respectively on top and bottom of A in Figure 13).

The use of NF components instead of simple object controllers as in the Fractal reference implementation, allows a more flexible control of NF concerns and to develop more complex implementations, as the NF components can be bound to other NF components within a regular component application. This notion of defining a componentized membrane has been used in previous works to manage an define structural reconfigurations [13], [14]. In this work we use these notions to address self-adaptability concerns in service-oriented contexts.

GCM/ProActive is the reference implementation of GCM, within the ProActive middleware, where components are implemented by Active Objects, which can be used to implement new services using Java, or wrap existent legacy applications like C/Fortran MPI code, or a BPEL code.

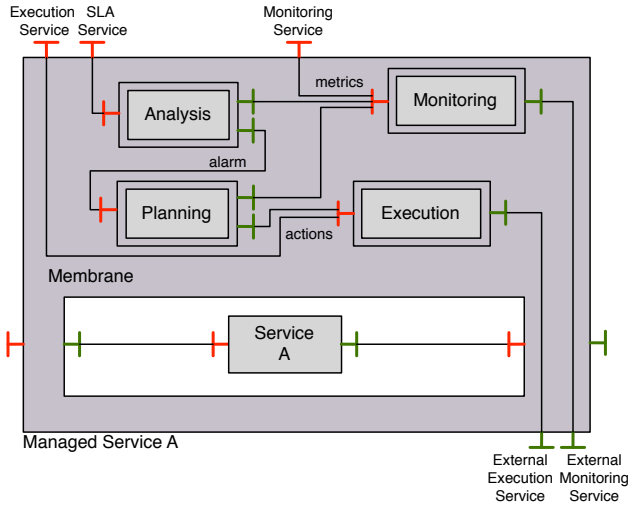

Figure 13. Framework implementation weaved to a primitive GCM component A. The MAPE components are isolated from the functional part in the *membrane* of the component.

The GCM/ProActive platform provides asynchronous communications with futures between bound components through GCM bindings. GCM bindings are used to provide asynchronous communication between GCM components, and 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. These controllers have been used to allow GCM to act as an SCA compliant platform, in a similar way as achieved by the SCA FraSCAti [15] platform, which however bases upon non distributed components (Fractal/Julia).

B. 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 or from the membrane of any GCM component, which becomes a managed service of the application.

We have designed a set of predefined components that implement each one of the elements we have described in Section III. This is just one of possible implementations, and particularly this has been designed to provide self-adaptable capabilities to the composition.

The general implementation view for a single GCM component is shown in Figure 13 (using the GCM graphical notation [11]), and resembles the design presented in Figure 3, however now the components that implement the MAPE control loop are inserted in the membrane and they are structurally isolated from the functional part. The framework is weaved in the GCM component A by inserting NF components in its membrane. Monitoring and management features are exposed through the NF server interfaces *Mon-*

itoring Service, *SLA Service* and *Execution Service* (top of Figure 13). NF components can communicate with the NF components of other GCM components through the NF client interfaces *External Monitoring Service* and *External Execution Service* (bottom of Figure 13). The sequence diagram of the self-adaptability loop is shown in Figure 14.

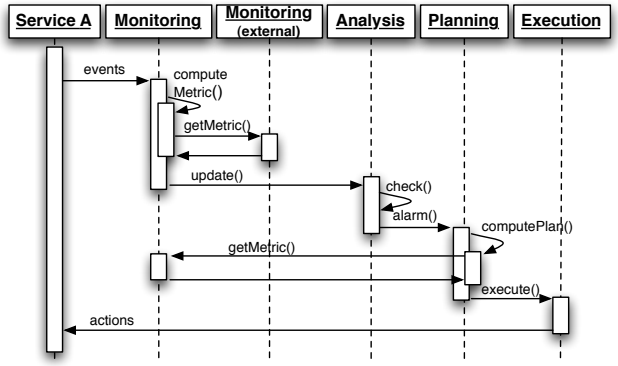

Figure 14. Sequence diagram for the autonomic control loop

C. Monitoring

We have designed a set of probes for CPU load and memory use, and incorporated them along with the events produced by the GCM/ProActive platform. Over them, we provide a *Monitoring* component, shown on Figure 15, which includes (1) an *Event Listener* that receives events from 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* that stores objects that we call *Metrics*, 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.

The *Monitor Manager* receives a Metric that, in our implementation, 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). For example, a simple *respTime* metric to compute the response time of requests, requires to acess the *Record Store* for retrieving the events related to the start and finish times of the service of a request.

Consider, for instance, that the Tourism Service needs 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 can ask the *requestPath* to the *Monitoring* components of all the services involved while serving r_0, which can repeat the process themselves; when no more calls

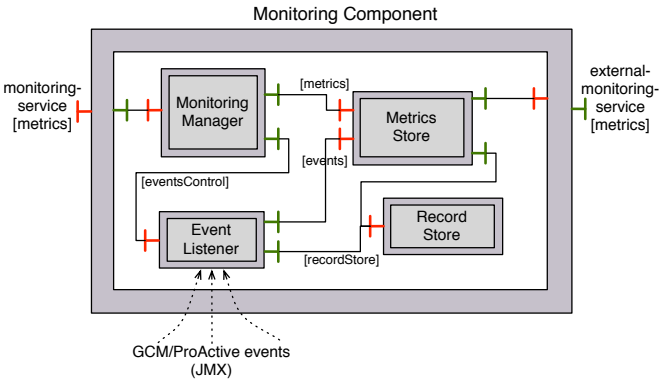

Figure 15. Internal Composition of the Monitoring component

are found, the composed path is returned 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 Service, the complete path is built and it is possible to identify the time spent in each service.

D. SLA Analyzer

The *SLA Analyzer* is implemented as a component that queries the *Monitoring* component. The *SLA Analyzer* consists in (1) an *SLO Analyzer*, which transforms the SLO description to a common internal representation, (2) an *SLO Store* that maintains the list of SLOs, (3) an *SLO Verifier* that collects the required information from the *Monitoring* interface and generates alarms, and (4) an *SLA Manager* that manages all the process.

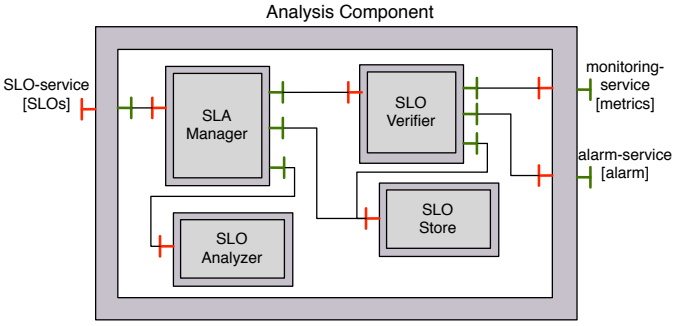

Figure 16. Internal Composition of the Analysis component

In this implementation, an SLO is described as a triple $\langle metricN, comparator, value \rangle$, where *metricN* is the name of a metric. The SLA Monitor subscribes to the *metricN* from the *Monitoring* component to get the updated values and check the compliance of the SLO.

For example, the Tourism Service includes the SLO: "All requests must be served in less than 30 secs", described as $\langle respTime, <, 30 \rangle$. 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 including the description of the faulting SLO.

E. Planning

The *Planning* component, shown on Figure 17, includes a *Strategy Manager* that receives an alarm message and, depending on the content of the alarm, it triggers one of several bound *Planner* components. Each one of the *Planner* components implements a planning algorithm that can create a plan to modify the state of the application. Each *Planner* component can access the *Monitoring* components to retrieve any additional information, they may need; the output is expressed as a list of actions in a predefined language.

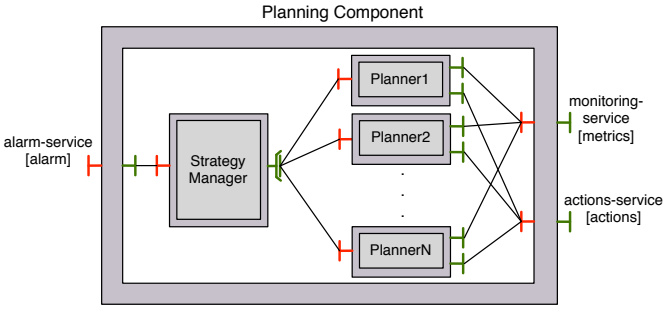

Figure 17. Internal Composition of the Planning component

In our implementation we profit by the selective 1-to-N communications provided by GCM to decide the *Planner* component that will be triggered. For example, if the SLO violated is related to response time, we may trigger a planner that generates a performance-oriented recomposition; or if a given cost has been surpassed, we may trigger a cost-saving algorithm. The decision of what planner to use is taken in the *Strategy Manager* component. However, the possibility of having multiple strategies might be a source for conflicting decisions; while we do not provide a method to solve these kind of conflicts, we assume that the conflict resolution behaviour, if required, is provided by the *Strategy Manager*.

We have implemented a simple planning strategy that, given a particular request, asks to compute the *requestPath* for that request, then finds the component most likely responsible for having broken the SLO, and then creates a plan that, when executed, will replace that component for another component from a set of possible candidates. Applied to the Tourism Service, suppose a request has violated the SLO $\langle respTime, <, 30 \rangle$. The *Strategy Manager* activates the *Planner* component that obtains the *requestPath* for that request along with the corresponding response time, selects the component that has taken the highest time, then obtains a set of possible replacements for that component, and obtains for each of them the *avgRespTime* metric. The output is a plan expressed in a predefined language that aims to replace the slowest component by the chosen one.

Clearly this strategy does not intend to be general, and does not guarantee an optimal response in several cases. Even, in some situations, it may fail to find a replacement and, in that case, the output is an empty set of actions. However, this example describes a planning strategy that can be added to implement an adaptation for self-optimizing and that uses monitoring information to create a list of actions.

F. Execution

The *Execution* component, shown on Figure 18, includes a *Reconfiguration Engine*. This engine uses a domain specific language called PAGCMScript, an extension of the FScript [16] language (designed for Fractal), which supports GCM specific features like distributed location, collective communications, and remote instantiation of components.

The *Execution* component receives actions from the *Planning* component. As many strategies may express actions using different formats, a component called *Execution Manager* may require a transformation to express the actions in an appropriate language for the *Reconfiguration Engine*, using a *Translation* component. The *Execution Manager* may also discriminate between actions that can be executed by the local component, or those that must be delegated to external *Execution* components.

For example, if a planner determines that the *Weather* service must be removed from the composition, it can be unbound from the *Tourism Service* by using a PAGCMScript command like the following

```
unbind($tourism/interface::"weather")
```

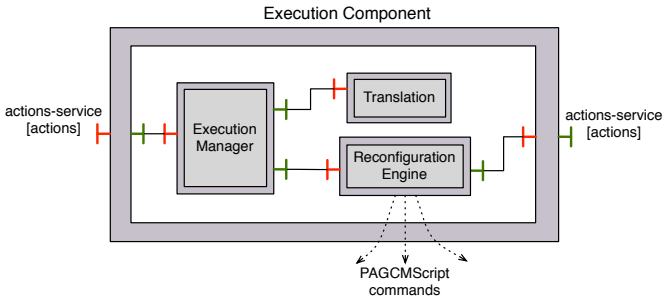

Figure 18. Internal Composition of the Execution component

G. Generalization and Dynamic Insertion

The GCM-based framework shown in Figure 13 has been presented as an instantiation of the SCA version shown in Figure 3. Indeed, the SCA design of Figure 3, presented only in terms of SCA elements, can be realized for any SCA runtime platform. The deployment of the framework may be done by injecting the required SCA description in

the SCA ADL file. This way, the application is deployed with all the needed elements of the framework attached.

In our implementation, however, we allow the insertion of the components that provide the autonomic behaviour to occur at runtime. We have provided a console application that can use the standard NF API of GCM components to insert or remove at runtime the required components of the framework.

The console, while not being itself a part of the framework, shows that an external application can be built and connected to the NF interfaces of the running application and handle at runtime the composition and any subsequent reconfiguration, if needed, of the monitoring and management framework itself. In the use case that we present in Section V-B we use this console application, for instance, to interact with the *Monitoring* interface and obtain the value of certain metrics.

V. Use Case and Evaluation

This section shows the experimentation we have made with the implementation of our framework over the GCM/ProActive middleware. The experimentation is divided in two parts. First we execute some micro-benchmarks to analyze the overhead incurred by the execution of the MAPE components concurrently with the functional application in our particular implementation. Then, we describe from a working point of view the use of the framework to insert and modify a set of MAPE components into a concrete application, showing the practicality of our proposition.

A. Performance

We have built a sample application with several components that interchange messages. Each execution performs a distributed computation through all the components to compose a return message, so that each execution generates a communication that ultimately reaches every other component.

1) MAPE Execution Overhead: We run a repetition of n messages in two versions of the application: one with no MAPE components inserted, and another with a version of each MAPE component inserted in all the membranes. This is, a complete MAPE cycle in each component. The *Monitoring* component computes metrics related to response time; the *Analysis* component checks an *SLO* that compares the response time in a *push* mode (subscription) upon each update of the *respTime* metric and, in case it is bigger than 1 second, it sends an alarm to a *planner* component. The *planner* only checks the last value obtained for the *respTime* metric from the *Monitoring* component, but does not generate actions. In order to isolate the execution of the application respect to network communication, in this experiment all the components are deployed in a single node.

The times obtained for each execution depending on the number of requests, and the overhead obtained for the total execution is shown in Table I. The "Base" column shows the execution time without any MAPE component inserted, and the "w/MAPE" columns shows the execution with all the MAPE components inserted and running in the membranes of each functional component.

#msgs	Base (sec)	w/MAPE (sec)	Diff.	%Overhead
1000	6.98	8.00	1.02	14.6
2500	17.20	19.29	2.09	12.2
5000	34.39	39.18	4.79	13.9
10000	68.57	77.55	8.98	13.1
20000	140.38	158.91	18.53	13.2

Table I
Execution Overhead in non-distributed application with MAPE components executing in the membrane of functional components

We observe that the overhead incurred stabilizes around 13% of the initial time. Although it seems important, we must highlight that this case represents one of the worst cases of an execution, as the only thing that this application does is to send requests to other components, while little functional work is done by each individual service. In a more general situation, an application would be expected to do some other activity than only sending requests. However, this experiment allows us to test the behaviour of our framework implementation under a high load and still obtaining acceptable results.

2) MAPE Execution and Communication Overhead: In this experiment we use a distributed version of the application, where each component is deployed in a different node in a grid environment. In this case, in addition to the overhead caused by the execution of the MAPE components, we expect to have and additional overhead caused by the communication between the membranes of the different functional components.

The results are shown in Table II. The "Base" column shows the execution time of the distributed application without any MAPE component inserted, and the "w/MAPE" columns shows the execution with all the MAPE components inserted and running in all the membranes, and in the same node of their corresponding managed functional component.

In this case, the overhead reaches around 15% of the "Base" execution time. This is not a big increment with respect to the previous situation, while the amount of network communication is bigger. Once again, we must mention that this particular experiment reflects a situation where the components spent most of the time sending and receiving requests, which consequently triggers reactions over the application. The node where each component runs must support the execution of both the original functional node, and the activity of the additional NF components.

#msgs	Base (sec)	w/MAPE (sec)	Diff.	%Overhead
1000	29.66	33.69	4.03	13.6
2500	72.20	82.18	9.98	13.8
5000	138.72	156.74	18.02	13.0
10000	271.45	314.20	42.75	15.7
20000	539.26	624.27	85.01	15.8

Table II
EXECUTION OVERHEAD IN A DISTRIBUTED APPLICATION WITH MAPE COMPONENTS EXECUTING IN THE MEMBRANE OF FUNCTIONAL COMPONENTS

Overall, the insertion of the MAPE components in this implementation implies a bigger load in the execution of the managed component, which is natural. In a worst-case scenario, the overhead incurred does not account for more than 15% of the not-managed execution. This measure however, is not completely accurate, as the actual overhead incurred by the MAPE components may depend on many additional factors. For one, the specific logic applied to the *metrics* implementation, and to the *planner* strategies may require much more additional processing. Moreover, the *planner* strategy may require to (it is not forbidden to) temporarily stop the functional execution of the component if some computation needs to be performed in an isolated way, introducing more overhead in the execution. However, we must remember that the planning activity should be executed mainly for resolving undesired situations and not become the main activity of the application.

Another factor is the supporting implementation. In our case we have conducted our experiments over a distributed environment supported by the GCM/ProActive middleware. This particular implementation profits of asynchronism to allow the concurrent execution of the MAPE components. Each implementation of the framework, however, may profit of their particular characteristics and optimize the implementation.

B. Use Case

We implement the application described in Section II using the GCM/ProActive implementation of our framework. The application is presented as an example of use of the framework to add progressively autonomic behaviour to an application.

The application is initially designed without any monitoring or management activity. However, in order to be able to insert some MAPE components later, it is necessary that the required interfaces be previously declared. In the context of our implementation, this is achieved by introspecting the functional interfaces defined for the component and, before instantiating the component, declaring the monitoring and management interfaces. This extension of the originally declared interfaces is done in an automatic way by our implementation prior to deploy the components. The components are, thus, deployed without any MAPE components inserted, however they are prepared to receive them and gradually support autonomic behaviour.

The design from Figure 1 is shown using the GCM notation in Figure 19 for the *Tourism Service* composite.

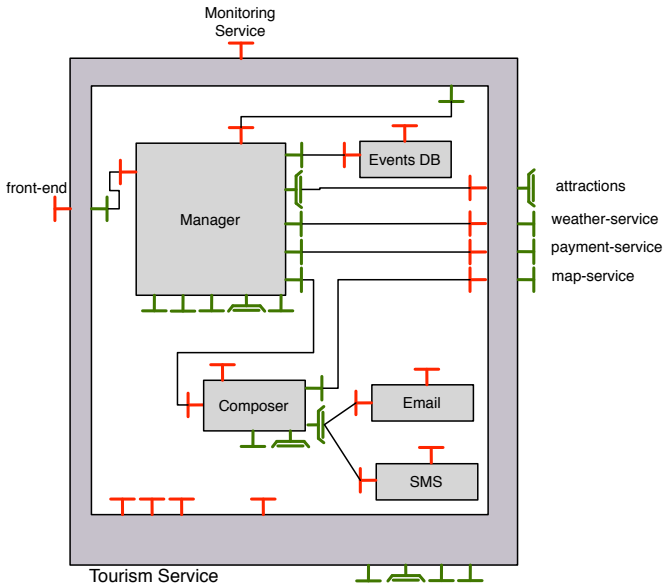

Figure 19. GCM description of the Tourism Service composite. NF interfaces are available but no NF Component is in the membrane

1) Inserting Monitoring activity: In order to monitor the application, it is possible to insert a *Monitoring* component as the one described in Section IV-C. Figure 20 shows the *Tourism Service* composite once the *Monitoring* component has been inserted in its membrane, and in each one of its subcomponents. The NF bindings are shown as solid lines inside the membrane, and as dashed lines in the functional part.

Using this configuration, it is now possible to connect to the *Monitoring* interfaces of each component and insert, query, or remove some metrics. Among others, we have implemented a metric called *respTime*, which computes the response time on the server side of a binding, a metric called *avgRespTime* that keeps an average of response time on each interface, and another one called *requestPath* that uses the previous one to trace the tree of calls generated by a request including the response time on each component. Our console application includes commands to connect and interact with the monitoring interfaces, providing an interaction like it is shown in Listing 1.

Listing 1. Request Path computation by invoking a metric from the console. Numbers in parenthesis are unique request identifiers

```
> addMetric TourismServ requestPath rp
Metric rp (type: requestPath) added to TourismServ
...
> addMetric MappingServ requestPath rp
Metric rp (type: requestPath) added to MappingServ
```

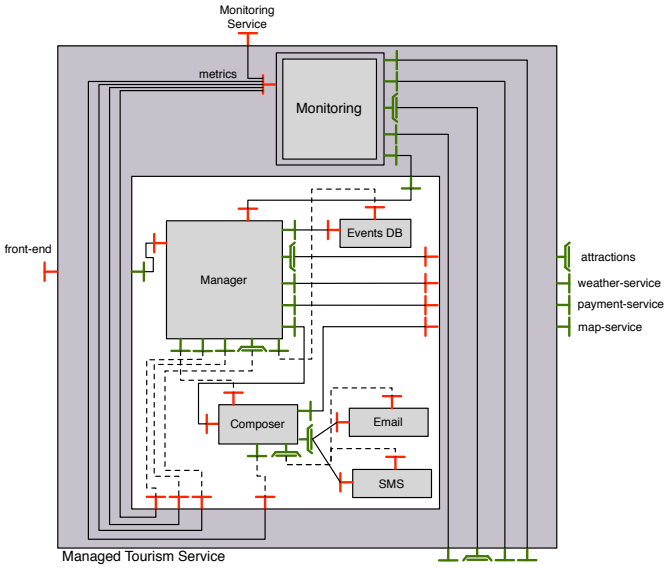

Figure 20. GCM description of the Tourism Service composite, once the *Monitoring* component has been inserted in the membrane of all components and its NF Interfaces are bound

```
> runMetric TourismServ rp 1131284383
Path from TourismServ, for request 1131284383
Request Path from request 1131284383
* (1131284383) TourismServ.reqs.buildDoc:
client: 7943 server: 7646
*     (-516789329) Manager.events.getEvent:
client:  410 server:  398
*     (-516789328) Manager.weather.getWeather:
client: 2224 server: 2118
*        (1131284384) TourismServ.weather.getWeather:
client: 2011 server: 1841
*     (-516789327) Manager.attr3.getTicktData:
client: 3019 server: 2867
*        (1131284385) TourismServ.attr3.getTicktData:
client: 2860 server:  702
*     (-516789326) Manager.composer.buildDoc:
client: 5066 server: 5002
*        (1278875256) Composer.mapping.getLocn:
client: 3200 server: 3109
*           (1131284385) TourismServ.mapping.getLocn:
client: 3006 server: 2955
*        (1278875257) Composer.email.send:
client: 1434 server: 1137
>
```

2) Automating the monitoring: By connecting to the *Monitoring* interface, it is possible to introduce metrics and request their values. However, this still requires to explicity ask for the values and interpret them in an external way from the application as shown on Listing 1.

A next level of autonomic behaviour is achieved by automating the monitoring. The *Analysis* component can be dynamically inserted in the membrane and bound to the *Monitoring* component to check periodically certain metrics. In the example, an *Analysis* component like that described in Section IV-D is inserted in the *Tourism Service* component, and made available through the *SLA Service* interface. This interface allows to insert SLOs according to the format described in Section IV-D and associate them to the metrics provided through the *Monitoring* interface. In the example shown in Figure 21, the *Analysis* component uses the *avgRespTime* metric to check the average response time on the *frontEnd* interface.

In case a condition is not met, the *Analysis* component is expected to throw an alarm through its *alarm* interface. This notification must be logged and produce some action in order to be useful. A simple way to handle it is to insert a *Planning* component like that described in Section IV-E that implements a *planner* whose only action is to send a notification email about the faulty condition. This simple activity is described in Figure 21.

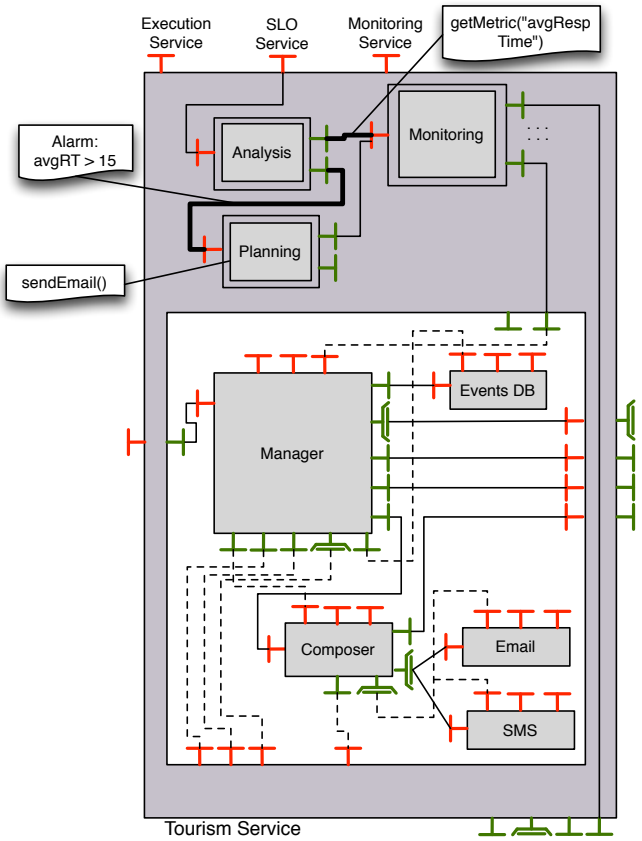

Figure 21. *TourismService* with *Monitoring*, *Analysis*, and simple *Planning* inserted. The basic action is to check a metric and notify in case a threshold is reached. The complete set of monitoring bindings is not shown for clarity.

3) Providing a self-optimizing autonomic loop: At this moment the autonomic control loop is not complete, as the final action is still dependent on a human administrator. In order to provide a complete autonomic behaviour, a more complex *planner* can be added to the *Planning* component and associated to the SLO that checks the *avgRespTime* metric, and an *Execution* component must be inserted in each component where an action may be carried on.

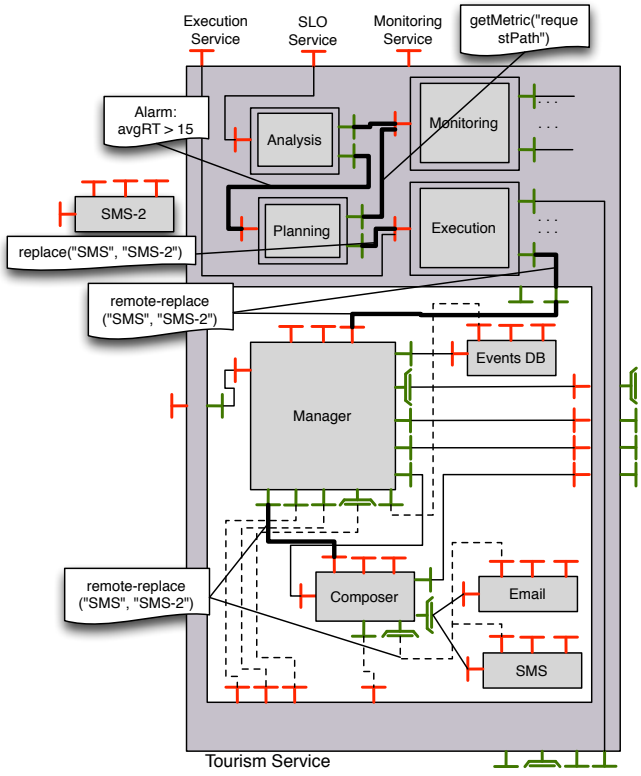

Figure 22. *TourismService* with all MAPE components, providing a self-optimizing behaviour. If the average response time is not met, the slowest component is identified and replace by an equivalent one. The complete set of monitoring bindings is not shown for clarity.

A simple self-optimizing autonomic behaviour may consist of reacting when the desired average response time is not obtained in the *frontEnd* interface, and replacing the component that takes the most time to execute by an equivalent quicker component.

To implement this kind of action, the new *planner* component must implement a behaviour slightly more complex that the old component. The *planner* first needs to identify the "faulty" component, which in this case is defined as the one that takes the biggest slice of the total time to serve a request. This information is obtained from the *requestPath* metric that can be obtained from the *Monitoring* component. Once the component to be replaced is identified, the *planner* must find a proper replacement. The discovery process is not shown in the example, however we assume that an alternative, more efficient component can be found (if that is not possible, the *planner* can safely fail without producing an action). Finally, the replacement action must be carried on in the appropriate binding. By using the connections between the *Execution* components, the action can be propagated and the binding can be updated. The sequence of the propagation of actions, and the application with all the MAPE components inserted is shown in Figure 22.

This particular implementation is a concrete implementation of an effective autonomic self-optimizing behaviour built through our framework and dynamically inserted in a running application.

4) Providing a self-healing behaviour based on infrastructure: As we have mentioned before, the implementation of sensors and actuators are the only parts of our framework that are heavily dependent on the particular implementation. In the previous example, we have relied in sensors that detect JMX events produced by the GCM/ProActive implementation of the functional code, and actuators that rely on the PAGCMScript scripting language to describe reconfiguration actions.

A different implementation of sensors can be oriented to measure characteristics of the running infrastructure like CPU or memory utilization, by using operating system calls, or communicating with a virtual machine manager. Once these sensors are implemented, their values can be fetched by the *MetricsStore* and they are available for the rest of the components of the framework as any other metric value. These kind of sensors are particularly useful in a Cloud computing environment, where the introduction of autonomic behaviour in the application seems like a promising way to benefit of the elasticity of the running infrastructure.

Infrastructure-based sensors may be used to provide a simple self-healing behaviour in which a metric called *avgLoad* is used to determine the average load of the node where a component is runnning. In case the load surpasses a threshold, a *planner* is activated, which determines the node with the highest load, and migrates one component from that node to another newly acquired node, expecting to achieve a better balance.

Figure 23 shows an example of this behaviour.

5) Integrating adaptation on a cloud infrastructure: We have also integrated the infrastructure monitoring capability of our framework to provide adaptation through the lifecycle of an SOA-based application running on a cloud environment [17].

Figure 24 shows a simplified version of the *TourismService* application where the *Composer* component is duplicated and each component is located in a different node of a cloud infrastructure. The integration of our monitoring capabilities through our framework allows the collection of information both from the infrastructure sensors, and from the runtime levels, and made it available at a higher level view.

From the unified view, it is possible to interact through the *Execution* interfaces and introduce modifications both at the component runtime architecture level as we showed on Figure 22, or by acquiring new nodes from a cloud infrastructure and migrate a component to that node, in order to balance the load of the application. Such example is shown on Figure 25, where component $C2$ is migrated from node C to a newly acquired node D.

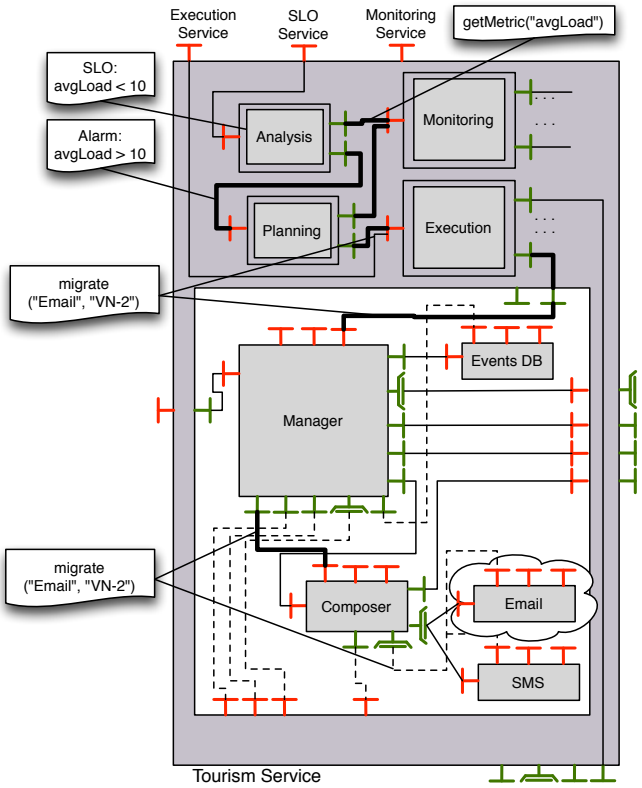

Figure 23. *TourismService* with all MAPE components, providing a self-healing behaviour using sensor over the infrastructure. When a component runs in a node that exhibits a high load. One component is migrated to a newly acquired node, illustrated as a cloud provided node. The complete set of monitoring bindings is not shown for clarity.

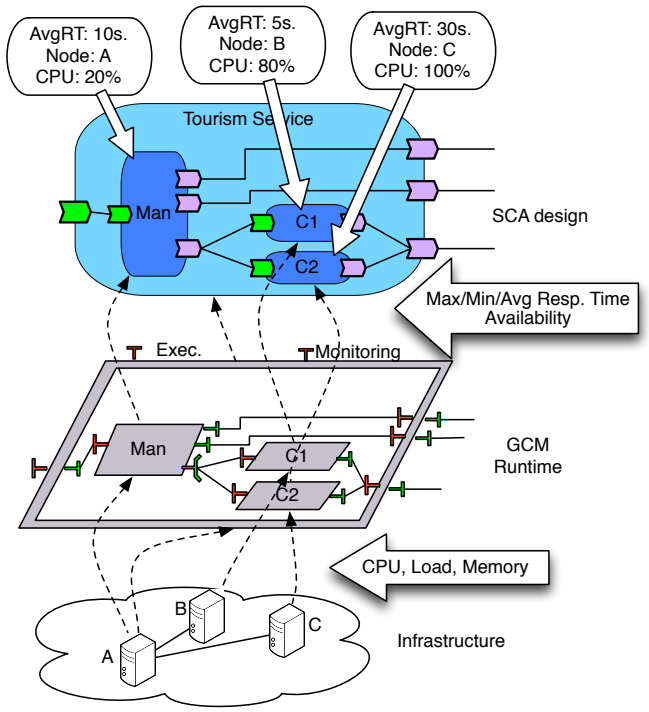

Figure 24. *TourismService* with Monitoring and Execution interfaces. The GCM implementation allows to retrieve information from the infrastructure and the runtime middleware, and associate it to the SCA design.

VI. Related Work

Several works exist regarding monitoring and management of service-oriented applications and about the implementation of autonomic control loops.

A set of works tackle the implementation of each phase mostly in a separate way. We can find infrastructures for monitoring components and services [18], [19], [20], and tools for monitoring grid and cloud infrastructures [21], [22], [23]. The work of Comuzzi et al. [24] 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).

Regarding the Analysis phase, several works integrate SLA monitoring and analysis [25] with SLA fulfillment [8], [26]. For representing the conditions to verify, several languages have been proposed [4] like SLAng [5], WSLA [6] and WS-Policy [7], which are mostly oriented to specify the agreement conditions between providers and consumers. Our claim is that our component based approach allows the integration of one of these languages, specifically in the *SLO Analyzer* shown in Section IV-D to represent the conditions.

On the area of planning strategies for adaptation, several planning algorithms can be found using different techniques. Some of them try to solve the problem of dynamically selecting a set of services that accomplish some determined QoS characteristic [27] using techniques from the genetic algorithms area [28], [29] or using linear and integer programming [30]. Other common way to separate the workflow composition from the selection of services is to rely on *abstract services* with some optionally defined QoS constraints, and bind them to proxies or *brokers* that are in charge of collecting information from a set of *candidate services* and performing the selection to bind *concrete services* to them [31], [32], [33]. Those works intend to compose a service, previous to execution, that complies with the required QoS characteristics. On the other side, other works address the problem of dynamically adjusting a composition at runtime [34], which is closer to the autonomic control loop that we provide, although it makes encapsulation harder as they require a closer integration between the monitoring and analysis phases with the planning phase. The runtime nature of these approaches imposes restrictions on the time spent for

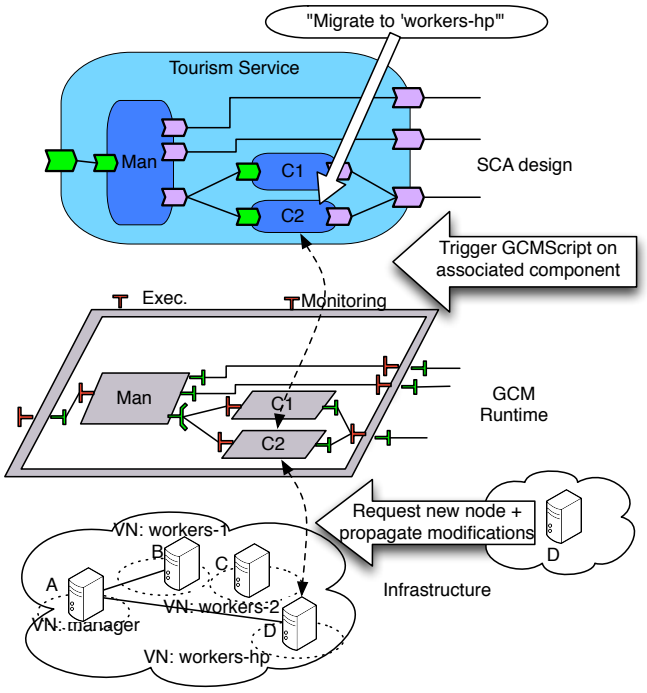

Figure 25. By sending commands through the *Execution* interface, reconfigurations can be enacted both at the runtime level, as well as in the infrastructure level.

computing the necessary rebinding. Some heuristics include K-means clustering of candidate services [35], and filtering of services that combine local and global optimizations [36] and skyline selections [37].

Regarding the *Execution* phase, recent component systems have been designed to take into account support for executing reconfigurations. Among them, works like FraSCAti [15] and SAFRAN [38] include methods for dynamically modifying the composition of an application. FScript [16] is a scripting language closely related to Fractal [12] based applications to describe such reconfigurations, and is the base for our own scripting language PAGCMScript.

Our approach, however, provides support for dynamically building complete autonomic control loops through a meaningful integration of the previous phases. The existing works that provide complete frameworks for the MAPE loop include Rainbow, and architecture-based approach providing a single autonomic control loop [39] that uses a model of the managed architecture to analyze and generate adaptations, which are later mapped to the effective system using a set of sensors and actuators. Another similar work to ours [40] proposes a generic context-aware framework that separates the steps of the MAPE control loop to provide self-adaptation; their work allows the implementation of self-adaptive strategies, though not much is mentioned about runtime reconfigurability, or the possibility to have multiple strategies. Also, we do not necessarily consider that all services require the same level of autonomicity.

CEYLON [41] is a service-oriented framework for integrating autonomic strategies available as services and using them to build complex autonomic applications. They provide the managers that allow the integration and adaptation of the composition of the autonomic strategies according to evolving conditions. In CEYLON, autonomicity is a main *functional* objective in the development of the application, while in our case, we aim to provide autonomic QoS-related capabilities to already existing service based applications. Also, we take benefit of the business-level components intrinsic distribution and hierarchy to split the implementation of monitoring and management requirements across different levels, thus enforcing scalability.

VII. Conclusion and Perspectives

We have presented a generic component-based framework for supporting monitoring and management tasks of component-based SOA applications.

The strengths of our approach include a clear separation of concerns between the functional content and the management tasks, relieving the programmer of the functional application to integrate the managment activities. The framework is generic in the sense that most of its components can be implemented in an independent way from the supporting technology of the application. The necessary implementation-dependent elements, such as sensors and actuators are encapsulated in components, and made available through a common interface to the rest of the framework. Finally we provide two levels of flexibility as we can dynamically insert or remove sensors, conditions, planning strategies and actuators in a previously existent skeleton that provides the autonomic control loop; and we also allow the modification of the composition of the control loop by including phases like analysis and planning only when they are needed and providing different degrees of autonomicity to each component.

We have provided an implementation of our framework as a self-adaptation loop for component-based services, thanks to the composition of appropriate monitoring, SLA management, planning and reconfiguration components. This prototype has been developed in the context of an SCA compliant platform that includes dynamic reconfiguration and distribution capabilities.

This approach provides a high degree of flexibility as the skeleton we have provided for the autonomic control loop can be personalized to support, for example, different planning strategies, and leverage heterogeneous monitoring sources to provide the input data that these strategies may need (for example, performance, price, energy consumption, availability).

One point not targeted by our proposition is the problem of conflict resolution. Indeed we may think about two kinds

of conflicts: one when two or more different planners generate opposite actions, or actions that invalidate each other; and the other situation where the result of an action triggers a chain of autonomic reactions that does not converge to a stable state resulting in a *livelock* situation. Both types of conflicts must be eventually dealt with, and they may arise as a consequence of the fact that conditions are inserted in the system in a way that they may be unaware of each other. For that matter we can consider an additional component that collects the output of each planner involved and that is capable of resolving these kind of conflicts inside the planning component. The specific implementation of a conflict resolution mechanism is not a concern of this work. Nevertheless, its integration is a promising perspective that goes in the direction of improving the autonomic capabilities that can be added to an application.

REFERENCES

[1] C. Ruz, F. Baude, and B. Sauvan, "Flexible Adaptation Loop for Component-based SOA applications," in *IARIA 7th International Conference on Autonomic and Autonomous Systems (ICAS 2011)*, 2011, pp. 29–36.

[2] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," *IEEE Computer*, vol. 36, no. 1, 2003.

[3] (2007, Mar.) Service Component Architecture Specifications. OASIS. Last accessed: April 2011. [Online]. Available: http://oasis-opencsa.org/sca

[4] A. Sahai, V. Machiraju, M. Sayal, A. van Moorsel, and F. Casati, "Automated SLA Monitoring for Web Services," in *Management Technologies for E-Commerce and E-Business Applications*, ser. Lecture Notes in Computer Science, M. Feridun, P. Kropf, and G. Babin, Eds. Springer Berlin / Heidelberg, 2002, vol. 2506, pp. 28–41.

[5] J. Skene, A. Skene, J. Crampton, and W. Emmerich, "The Monitorability of Service-Level Agreements for Application-Service Provision," in *Proceedings of the 6th international workshop on Software and performance*, ser. WOSP '07. New York, NY, USA: ACM, 2007, pp. 3–14.

[6] A. Keller and H. Ludwig, "The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services," *Journal of Network and Systems Management*, vol. 11, pp. 57–81, 2003.

[7] (2007, Sep.) Web Services Policy 1.5 - Framework (WS-Policy). W3C. Last accessed: June 2012. [Online]. Available: http://www.w3.org/TR/ws-policy/

[8] P. Leitner, B. Wetzstein, F. Rosenberg, A. Michlmayr, S. Dustdar, and F. Leymann, "Runtime prediction of service level agreement violations for composite services," in *Proceedings of the 2009 international conference on Service-oriented computing*, ser. ICSOC/ServiceWave'09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 176–186.

[9] ProActive Parallel Suite. Last accessed: June 2012. [Online]. Available: http://proactive.inria.fr/

[10] F. Baude, V. Contes, and V. Lestideau, "Large-Scale Service Deployment–Application to OSGi," in *IARIA 3rd International Conference on Autonomic and Autonomous Services (ICAS 2007)*. IEEE Computer Society Press, Jun. 2007, pp. 19–26.

[11] F. Baude, D. Caromel, C. Dalmasso, M. Danelutto, V. Getov, L. Henrio, and C. Prez, "GCM: a grid extension to Fractal for autonomous distributed components," *Annals of Telecommunications*, vol. 64, pp. 5–24, 2009.

[12] E. Bruneton, T. Coupaye, M. Leclercq, V. Quma, and J.-B. Stefani, "The FRACTAL component model and its support in Java," *Software: Practice and Experience*, vol. 36, no. 11-12, pp. 1257–1284, 2006.

[13] F. Baude, L. Henrio, and P. Naoumenko, "Structural Reconfiguration: An Autonomic Strategy for GCM Components," in *IARIA 5th International Conference on Autonomic and Autonomous Systems (ICAS 2009)*. IEEE Computer Society, 2009, pp. 123–128.

[14] 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 structured parallel programming environment design," in *Parallel Computing - Software Technology, Algorithms, Architectures and Applications*, ser. Advances in Parallel Computing. North-Holland, 2004, vol. 13, pp. 617 – 624.

[15] L. Seinturier, P. Merle, D. Fournier, N. Dolet, V. Schiavoni, and J.-B. Stefani, "Reconfigurable SCA Applications with the FraSCAti Platform," in *Proceedings of the 2009 IEEE International Conference on Services Computing*, ser. SCC'09. IEEE Computer Society, 2009, pp. 268–275.

[16] 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*, vol. 64, pp. 45–63, 2009.

[17] C. Ruz, F. Baude, B. Sauvan, A. Mos, and A. Boulze, "Flexible SOA Lifecycle on the Cloud Using SCA," in *Proceedings of the 2011 IEEE 15th International Enterprise Distributed Object Computing Conference Workshops*, ser. EDOCW'11. IEEE Computer Society, 2011, pp. 275–282.

[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, last accessed: June 2012. [Online]. Available: http://www.informatik.uni-kiel.de/uploads/tx_publication/vanhoorn_tr0921.pdf

[19] I. Garcia, G. Pedraza, B. Debbabi, P. Lalanda, and C. Hamon, "Towards a Service Mediation Framework for Dynamic Applications," *2006 IEEE Asia-Pacific Conference on Services Computing*, pp. 3–10, 2010.

[20] P.-C. David and T. Ledoux, "Wildcat: a generic framework for context-aware applications," in *Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, ser. MPAC'05. ACM, 2005, pp. 1–7.

[21] M. L. Massie, B. N. Chun, and D. E. Culler, "The ganglia distributed monitoring system: design, implementation, and experience," *Parallel Computing*, vol. 30, no. 7, pp. 817 – 840, 2004.

[22] Hyperic. CloudStatus Monitoring. Last accessed: June 2012. [Online]. Available: http://www.hyperic.com/products/cloud-status-monitoring

[23] LogicMonitor. Last accessed: June 2012. [Online]. Available: http://www.logicmonitor.com/

[24] M. Comuzzi and G. Spanoudakis, "A Framework for Hierarchical and Recursive Monitoring of Service Based Systems," in *4th International Conference on Internet and Web Applications and Services, 2009. (ICIW'09)*, may 2009, pp. 383 –388.

[25] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar, "End-to-End Support for QoS-Aware Service Selection, Binding, and Mediation in VRESCo," *IEEE Transactions on Services Computing*, vol. 3, pp. 193–205, 2010.

[26] I. Foster, "Globus toolkit version 4: Software for service-oriented systems," *Journal of Computer Science and Technology*, vol. 21, pp. 513–520, 2006.

[27] C. Ghezzi, A. Motta, V. Panzica La Manna, and G. Tamburrelli, "QoS Driven Dynamic Binding in-the-many," in *Research into Practice – Reality and Gaps*, ser. Lecture Notes in Computer Science, G. Heineman, J. Kofron, and F. Plasil, Eds. Springer Berlin / Heidelberg, 2010, vol. 6093, pp. 68–83.

[28] S. Liu, Y. Liu, N. Jing, G. Tang, and Y. Tang, "A Dynamic Web Service Selection Strategy with QoS Global Optimization Based on Multi-objective Genetic Algorithm," in *Grid and Cooperative Computing – GCC 2005*, ser. Lecture Notes in Computer Science, H. Zhuge and G. Fox, Eds. Springer Berlin / Heidelberg, 2005, vol. 3795, pp. 84–89.

[29] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani, "A framework for QoS-aware binding and re-binding of composite web services," *J. Syst. Softw.*, vol. 81, pp. 1754–1769, Oct. 2008.

[30] L. Zeng, B. Benatallah, A. H.H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, "QoS-Aware Middleware for Web Services Composition," *IEEE Trans. Softw. Eng.*, vol. 30, pp. 311–327, May 2004.

[31] T. Yu and K.-J. Lin, "A broker-based framework for QoS-aware Web service composition," in *Proceedings of the 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service*, ser. EEE '05. IEEE Computer Society, 2005, pp. 22–29.

[32] D. A. D'Mello, V. Ananthanarayana, and S. Thilagam, "A QoS Broker Based Architecture for Dynamic Web Service Selection," *2nd Asia International Conference on Modelling & Simulation*, vol. 0, pp. 101–106, 2008.

[33] M. Serhani, R. Dssouli, A. Hafid, and H. Sahraoui, "A QoS broker Based Architecture for Efficient Web Services Selection," in *Proceedings of the 2005 IEEE International Conference on Web Services (ICWS 2005)*, 2005, pp. 113 – 120 vol.1.

[34] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani, "QoS-aware replanning of composite Web services," in *Proceedings of the 2005 IEEE International Conference on Web Services*, ser. ICWS '05. IEEE Computer Society, 2005, pp. 121–129.

[35] N. B. Mabrouk, S. Beauche, E. Kuznetsova, N. Georgantas, and V. Issarny, "QoS-aware service composition in dynamic service oriented environments," in *Proceedings of the ACM/IFIP/USENIX 10th international conference on Middleware*, ser. Middleware'09. Springer-Verlag, 2009, pp. 123–142.

[36] M. Alrifai and T. Risse, "Combining global optimization with local selection for efficient QoS-aware service composition," in *Proceedings of the 18th International Conference on World Wide Web*, ser. WWW'09. ACM, 2009, pp. 881–890.

[37] M. Alrifai, D. Skoutas, and T. Risse, "Selecting skyline services for QoS-based web service composition," in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW '10. ACM, 2010, pp. 11–20.

[38] P.-C. David and T. Ledoux, "An Aspect-Oriented Approach for Developing Self-Adaptive Fractal Components," in *Software Composition*, ser. Lecture Notes in Computer Science, W. Löwe and M. Südholt, Eds. Springer Berlin / Heidelberg, 2006, vol. 4089, pp. 82–97.

[39] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure," *IEEE Computer*, vol. 37, pp. 46–54, 2004.

[40] F. André, E. Daubert, and G. Gauvrit, "Towards a Generic Context-Aware Framework for Self-Adaptation of Service-Oriented Architectures," *5th International Conference on Internet and Web Applications and Services (ICIW'10)*, vol. 0, pp. 309–314, 2010.

[41] Y. Maurel, A. Diaconescu, and P. Lalanda, "CEYLON: A Service-Oriented Framework for Building Autonomic Managers," *7th IEEE International Conference and Workshops on Engineering of Autonomic and Autonomous Systems*, pp. 3–11, Mar. 2010.