Enabling SLA monitoring for component-based SOA applications

Cristian Ruz, Françoise Baude
INRIA Sophia Antipolis, CNRS, I3S, Univ. de Nice Sophia Antipolis
2004 Route des Lucioles, BP93, F-06902 Sophia-Antipolis CEDEX, France
{Cristian.Ruz,Francoise.Baude}@inria.fr

***Abstract*—This work contributes to the research of how to define a framework that can be plugged to a component-based SOA application to provide SLA monitoring for performance. We define a low intrusive Monitor Controller that listen to asynchronous notifications about the performance of the monitored application, and stores that data in a scalable way. The approach allows flexibility as it defines the Monitor Controller as a non-functional component bound to the monitored entities, and exposing the monitored data through a non-functional interface. A proof-of-concept implementation is described, showing the feasibility of this approach over a middleware that provides asynchronous communication between consumers and providers over a grid environment, and which also serves as a platform for a large-scale SOA.**

***Index Terms*—grid computing; SLA; SOA; monitoring; SCA; QoS;**

I. Introduction

In service provisioning relationships, contracts relating to Quality of Service (QoS) are agreed in the form of a Service Level Agreement (SLA). This agreement establishes parameters that are to be met during the service. To watch the fulfillment of this contract, these parameters must be monitored through techniques that allow for SLA monitoring [1].

SLA monitoring has several implications for the management of a runtime service. One of them is that it allows to determine if the provider is behaving accordingly to the contract, and in case it does not, expose the required information to take corrective actions. Another use for the monitored information is when performing dynamic recomposition. Monitored data can feed the information required during the dynamic recomposition of a service, based on a desired QoS [2].

Given a previoulsy agreed SLA, we are interested in being able to monitor the compliance of the provider to this agreement. Although there are several aspects related to SLA monitoring, like provenance and security, we point specifically to performance monitoring, that is, the time that a given service takes to answer the requests that it receives.

One way to implement SLA monitoring in an architecture, is through the insertion of probes or counters in critical parts of the architecture, to measure the time that the service takes to execute a request. This information can be complemented with rules defined to be executed in the case that the monitored parameters do not behave accordingly to the desired values, to provide autonomic management [3], [4].

In this work we present a framework to add SLA monitoring capabilities to a component-based application built on the principles of SOA. Our solution enables to monitor the performance of a component, and obtain information that can be used by another, possibly autonomic, component to take decisions about the compliance of the monitored component to an SLA, or allow an external application to provide meaningful information to a human agent. While we focus in performance monitoring, the approach can also be applied to other aspects of SLA monitoring like, for instance, provenance and security.

We demonstrate the feasibility of this SLA monitoring framework, through a proof of concept implementation in a middleware that provides asynchronous services over a grid environment, and also serves as a platform for large-scale SOA.

The following sections are organized as follow. Section II describes the context and positions our work. Section III describes our proof of concept implementation and the platform used. Section IV mentions some examples of how our solution can be used, and discusses performance issues. Section V presents related work, and finally section VI draws the conclusions.

II. Positioning of Contribution

Service Oriented Architectures (SOA) allow to separate the different concerns in large-scale enterprise applications, through the definition of independent and loosely coupled services that can be accessed in an standardized way. Services in a SOA environment inter-operate using a formal definition of interfaces. This way, services can be defined in a manner that is independent of the underlying platform or programming language. Individual services can be dynamically composed into workflows to form more complex applications.

Common technologies used for implementing a SOA environment are Web Services described using Web Services Description Language (WSDL), and Business Process Execution Language (BPEL) for describing business processes and workflows. UDDI for discovery of services, and SOAP for communication.

In this work, we consider an SOA application implemented according to the Service Component Architecture (SCA) [5]. SCA is a specification for designing and building SOA applications, in which services and workflows are modeled as components, called *SCA components*. SCA defines a hierarchical

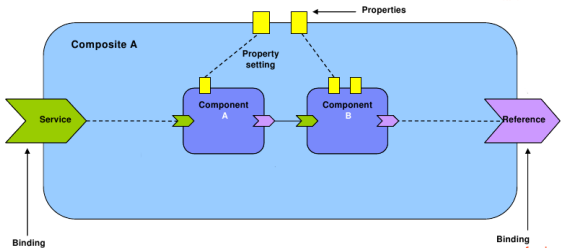

Fig. 1. SCA composite component embedding two components

component model that is independent of the technology used to implement these components.

SCA components can have any number of interfaces, named *SCA Services*, and dependencies, called *SCA References* to other SCA services. Dependencies between two SCA components are called *SCA bindings*, and several SCA components can be grouped into bigger *SCA composites* that hide their inner structure from the outside, and which can be handled in the same way as any other SCA component. One example and the common notation is shown in Fig.1.

Some examples of SCA implementations are Apache Tuscany [6], IBM Websphere Application Server [7], and FraSCAti [8].

An SCA-based SOA can potentially be composed of a wide number of services forming a complex workflow. These services can themselves be widely spread on a distributed environment like a grid or the Internet. We consider in our design to we have a large-scale SCA-based SOA.

There are several concerns we considered when designing our monitoring framework for large-scale SCA-based SOA.

- *Dynamicity*. SOA environments are highly dynamic. Provisioning relationships can change during the lifetime of a service or workflow. Therefore, tools must be able to adapt to these dynamicity. Our framework monitors a component-based SOA application that can adapt to changes in the runtime architecture.
- *Scalability*. In a large-scale application, to be able to monitor the performance of individual components, and make that information available to make decisions, we must store the collected data in a scalable way that also makes it easy to access. For that concern we use distributed storage of monitoring information in the involved components.
- *Low intrusiveness*. Monitoring has a cost in terms of performance, as some fraction of the computational power must be dedicated to collect and process the data. In our approach we use an asynchronous environment, where the management of the monitored data can be done in a way that does not block unnecessarily the service. Moreover, the environment is flexible enough to allow that, given the necessity, the data can be processed remotely.
- *External entities*. Existing works [9] analyze the situation of component-based applications where an SLA can be split among all components, managing non-functional concerns at different levels. That work considers that all components of the application, which can be seen as services in a SCA-based SOA, can be managed by means of autonomic managers attached to the components. Our approach aims to extend this situation by considering the existence of services implemented by components over which we do not have control, because they are provided by external entities. In that case we also need to take performance measures, to be able to track a perceived response time in order to watch the compliance to the SLA.

In the next section we describe the solution that we have designed, where these concerns are taken into account.

III. Design and Implementation of the SOA Monitoring Solution

We describe the design of out SOA monitoring solution. For the implementation we have chosen the GCM/ProActive middleware, so we also give the necessary background on this tool.

A. ProActive

The ProActive Grid Middleware [10] 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 invocation and futures.

ProActive provides means of notifying events by providing an asynchronous and grid enabled JMX connector [11]. Active objects are instrumented with MBeans which provide notifications about events at the implementation level. This way, monitoring information from the active objects, and from the grid infrastructure levels is exposed. We rely upon this JMX-ProActive mechanism, as it can handle grid-scale, large-scale monitoring information collection and notifications.

B. GCM

The Grid Component Model (GCM) [12] is a component model for grids, that takes the Fractal component model [13] as a base for its specification. Fractal defines a component model where components can be hierarchically organized, reconfigured, and controlled. GCM extends that model providing the components the possibility to be remotely located and deployed in a grid environment.

In GCM it is also possible to have a componentized membrane [14] that allows the existence of non-functional components. Having non-functional components allows to have a more flexible control of non-functional concerns, and develop more complex implementations [15]. Non-functional components can be accesed through NF server interfaces and can bind to other components through NF client interfaces.

GCM/ProActive is the reference implementation of GCM. Being built over an active object environment, the GCM/ProActive platform provides asynchronous communications, which poses more challenges for monitoring events.

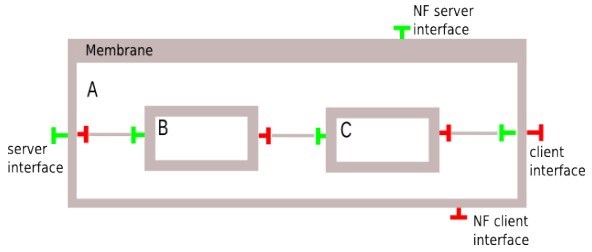

Fig. 2. Example GCM component A, including subcomponent B and C. The membrane takes care of non-functional concerns, and can include NF-components

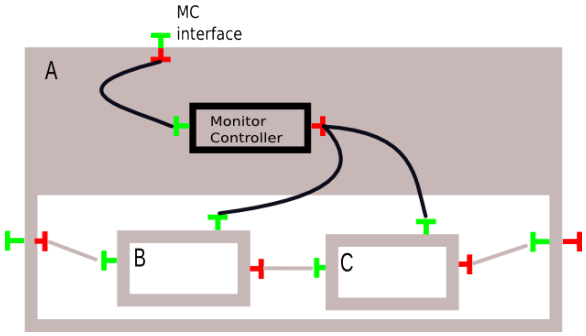

Fig. 3. Location of Monitor Controller in the membrane of a GCM composite A. The MC is connected to MC interfaces of subcomponents through NF-bindings

Fig.2 shows an example of a GCM component, with a composite component including two subcomponents. The composite appears to other external components as any other primitive component, hiding the details of its internal composition. When receiving a request through a server interface, the composite forwards the request to the internal component bound to that interface. In a similar way, when an internal component wants to communicate to an external component, it must do it through a call to the composite, which forwards the request to the bound component. This way, the composite abstract the internal implementation. This fact is used by our solution when describing how to store the monitored data.

Current work exists for giving an SCA personality to GCM components, so that GCM/ProActive can serve also as a runtime SCA infrastructure, being a valid platform to support our SLA monitoring solution.

C. Monitor Controller

At the implementation level of GCM/ProActive we already have some information available through the JMX-ProActive connector. We aim to leverage this information at the level of components, storing it, and make it available for other components or applications.

For this objective, we rely on *non-functional components*, also referred to *component controllers*, introduced in the membrane of GCM components. These component controllers give more flexibility, as they can be easily reused, and be assembled with other component controllers to further customize the monitoring process [15].

At the component level, we implement a specialized component controller, called the *Monitor Controller* (MC). Being a component controller, the MC is inserted in the membrane of a GCM component as shown in Fig.3. The mission of the MC is:

- Collect monitoring information by subscribing to the required MBean, and listening to appropriate events, and computing statistics.
- Store the important information of the events.
- Expose this information for later consulting and use.

D. Collecting information and statistics

We are interested in detecting events that allow to determine the performance of the services. In GCM/ProActive, we must consider that components are implemented by active objects, which provide asynchronous communication. Because of this, each component has its own request queue where incoming requests are stored for being processed according to a defined policy. Moreover, asynchronism is provided through the use of futures; that means that when a request is sent to a service, the response received is not the actual response, but a placeholder where the response is to be stored once the service effectively has taken place. Because of this, care must be taken to detect the moment when the real answer for a request has been received and not just a partial update.

We collect two kinds of events. Events that happen in the caller side, this is, the component that issues the request, and in the provider side, this is, the component that receives and serves the request. By monitoring both sides of the service we can determine, on the caller side, the service time perceived by the caller, where not only the effective time of service takes part, but also network latency can be involved. On the provider side, whenever that component can be monitored (as could be an external service over which we do not have any control), it allows to determine the effective time taken by the service, associate chains of requests, and trace service paths, where several components may be involved.

We detect events on both sides by subscribing to the appropiate MBeans and listening the JMX notifications generated. For each notification, the timestamp and attached information are stored. These timestamps are used to determine the timing data for each request.

At the provider side, we detect the following events:

- t_{arr}, the moment when a request is received by the component and stored in its request queue for later processing.
- t_{serve}, the moment when a request is taken from the request queue, and begins to be served.
- t_{reply}, the moment when the provider sends a reply to the caller, regarding a finished request, if any reply was expected. For void requests, this is not issued.

At the caller side, we detect:

- t_{send}, the moment when the caller sends a request to the provider

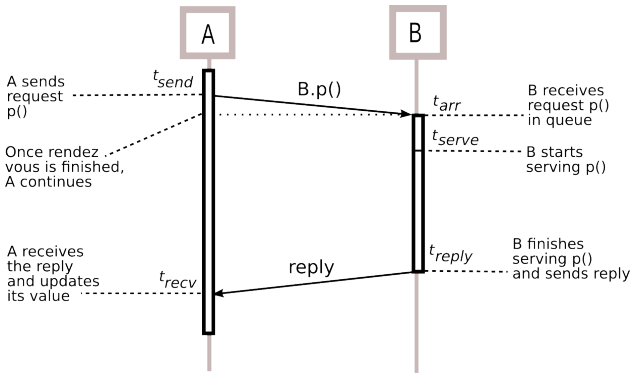

Fig. 4. Events captured during the asynchronous service of request call `p()` from component `A` to `B`

- t_{recv}, the moment when the caller receives a response from the provider, if a reply was expected. As we deal with asynchronous requests, the caller may have been doing some other work when the reply arrives. Also we have to be sure that this answer is a definitive reply, in the sense that by using it we don't have to block for another reply.

In GCM, components can be located in different places in a transparent way. Even a composite component can have some or all of their subcomponents located remotely. Given this situation, it is worth noting that the timestamps compared are always related to events that happen in the same pyhsical node, so the differences are meaningful.

Fig.4 shows the events that are detected during the service of a request.

As well as the events are collected, the Monitor Controller computes statistics like average, maximum or minimum response time, availability or throughput. These statistics are updated during the event collection, so they are readily available.

E. Storing information

For each of the events, we store the name of both the receiver and sender components, the interface and method called. This information is included in the notification generated by the middleware implementation, so that the Monitor Controller can receive it.

The Monitor Controller keeps two logs: the *Request Log* and the *Call Log*.

- The *Request Log* stores tuples that represent a request that arrives to the component through a server interface. It includes an identifier for the incoming requests, an identifier of the sender component, the interface and method called, and the timestamps for the events t_{arr}, t_{serve} and t_{reply}. There is one entry on this log for each request received by the component.
- The *Call Log* stores tuples that represent a new request called by the component on a client interface. It includes the identifier of the request that was being served when the new request was issued, this is, the parent request; a new identifier for the request that has been called, the identifier for the called component, the interface and method called, and the timestamps for the events t_{send} and t_{recv}.

TABLE I
STATE OF A'S CALL LOG, AND B'S REQUEST LOG AFTER EXECUTING THE EXAMPLE OF FIG.4. COMPONENT A IS ASSUMED TO BE SERVING A REQUEST WITH IDENTIFIER r_0 DURING THE CALL TO B

Call Log A

parent	current	dest	interface	method	t_{send}	t_{recv}
r_0	r_1	B	p	p1	5	25

Request Log B

id	caller	interface	method	t_{arr}	t_{serve}	t_{reply}
r_1	A	p	p1	4	7	18

By keeping both logs, it is possible to construct the path followed by the request. Each time that a new request is issued, an identifier for the new request is created, stored in the Call Log and transmitted. This way a causality relationship can be established.

Table I displays an example of the content of the logs, after the execution of the example from Fig.4. In this case, we assume that component `A` was serving a request with identifier r_0 while it made the call to component `B`, with identifier r_1.

In that example, the component `A` perceived a response time from `B` of 20ms. Component `B` received the request, kept it in the request queue during 3ms, and took 11ms to serve it. The total service time taken by `B` was 14ms. The 6ms. difference can be caused by network propagation. Note that each component stores its logs in an independent way.

Following the hierarchical nature of the composition of GCM components, the MC of the composite stores the aggregated information of all the requests that go through the composite. This means, the requests that have entered by a server interface, and the requests that have been generated through a client interface. All data related to the internal serving of the request is stored in the MC of the involved inner subcomponents. Whenever those data are required, the MC of the composite can make non-functional calls to the MC of the subcomponents like can be pointed in Fig.3. This strategy avoids the problem of having the same information stored several times and improves scalability. Even more, the MC can be located in a different place than the monitored component, relying on the asynchronous JMX-ProActive notifications to receive the events.

F. Exposing information

The information stored through the MCs must be exposed to be of utility. The interested recipients for the collected data can be other MCs, which can belong to inner components of a composite, or to external components; and external applications that may want to do further analysis.

The MC must provide access to the raw logs stored in the component, as well as more detailed information that could

require additional processing. For example, the average, maximum or minimum service time for this component, availability percentage, or throughput. For all these statistics, a filter can be applied to refer to an specific interface, a defined window of time, or a set of requests. Section IV-B gives more detail about that.

This information is exposed in two ways. The first is through a non-functional server interface. This allows that another component, which can be the MC of a composite to which the current component belongs (Fig.5), or well another external non-functional component (Fig.6), can connect to it to obtain the data it requires.

The second way to expose is for another external application, not necessarily component-based which can be used to do further analysis over the data collected, or expose it in a more meaningful, possibly graphical way. For this, a specialized MonitorController MBean is created, which can be connected through JMX standard means by using the JMX-ProActive connector.

IV. Monitored data usage

From the basic monitored data stored by the Monitor Controllers, more complex data can be obtained. We describe two possible uses, and show an example of a more complete situation.

A. Service Invocation Tracking

Monitored data can be used to track the service invocation path of a specific request, and obtain the time spent in each component involved in the service. This kind of decomposition, typical in profiling applications, can be used to detect critical points in the service of the request where performance could be failing, or to determine services utilised in a request for pricing goals.

Using the Monitor Controllers of the components, tracking information can be obtained. By using the information stored in the logs, the path of the request and its child request can be followed, forming a calling tree. This calling tree can be enriched by statistics associated to each request.

However, the logs are distributed through several components, which is an advantage from the scalability point of view, but poses a challenge for path reconstruction, as the MC of different components must communicate. We solve this by benefiting of NF bindings allowed by GCM [15]. In this context two situations arise.

When a component receives a request, it can serve it by itself using internal means, or by calling subcomponents in the case it is a composite. For tracking that request, the MC of the composite must talk to the MC of the inner subcomponents. This can be done using internal NF bindings as shown in Fig.5.

An example of how stored logs can be used to reconstruct the service invocation path of a request is presented in Section IV-C.

If the component requires another external component to serve a request, then it is necessary to connect to the MC interface of this external component. To solve this situation, a

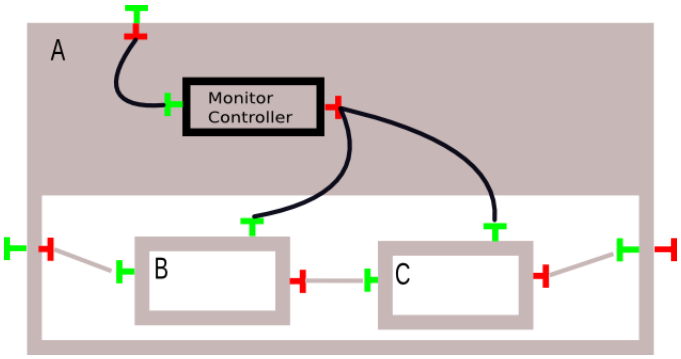

Fig. 5. MC of composite `A` connects to MC interfaces of inner subcomponents.

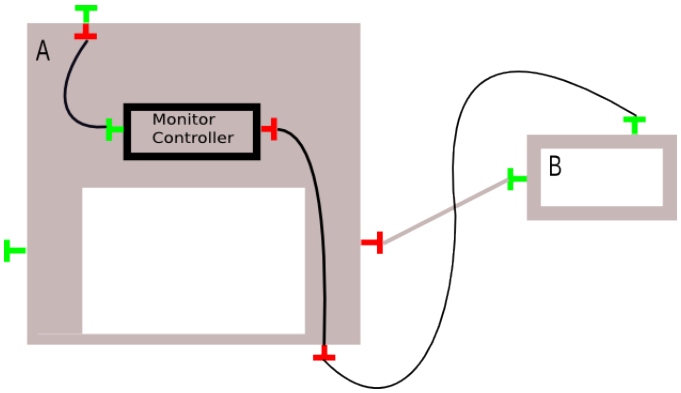

Fig. 6. Component `A` uses a NF client interface to connect to the MC interface of external component `B`

client NF interface of the caller component is used to connect to the MC interface of the called component. We define that such NF binding is done each time that a regular functional binding is set between two components. This situation can be seen in Fig.6.

After the execution path has been retrieved, the aggregated information can be stored inside the MC for later use, so it remains ready to be required again from another component or external application, avoiding to perform the tracking several times.

B. Filtering Statistics

In case when more complex processing is required from the statistics, we can profit of the flexibility provided by the non-functional components present in the membrane of GCM components.

Further processing could include filtering. As already stated, the Monitoring Controller provides, through an interface, basic statistics like average, maximum, and minimum service time that are computed and updated while new request are processed and the related events are detected. Another application could require filtering these statistics to include only a specific interface or method inside an interface, a defined window of time, or a set of request according to some property.

The Monitor Controller can be extended to include these filter capabilities while exposing the appropriate non-functional

TABLE II
STATE OF THE CALL AND REQUEST LOGS AFTER EXECUTING THE EXAMPLE OF FIG.7. WHILE SERVING r_0, COMPONENT B MAKES TWO CALLS TO D. TIMES ARE DISPLAYED IN SECONDS.

Call Logs

CallLog	parent	id.	dest	intf.	meth.	t_{send}	t_{recv}
Z	-	r_0	A	p	p1	0	11.515
A	r_0	r_1	B	p	p1	0.000741	11.510
A	r_3	r_4	E	u	u1	3.006	4.509
A	r_6	r_7	F	v	v1	8.008	9.511
B	r_1	r_2	D	s	s1	1.503	4.510
B	r_1	r_5	D	s	s1	6.505	9.512
D	r_2	r_3	A	u	u1	3.006	4.509
D	r_5	r_6	A	v	v1	8.007	9.512

Request Logs

RLog	id	caller	intf.	meth.	t_{arr}	t_{serve}	t_{reply}
A	r_0	Z	p	p1	0.000048	0.000133	0.000845
A	r_3	D	u	u1	3.006	3.007	3.008
A	r_6	D	v	v1	8.008	8.008	8.009
B	r_1	A	p	p1	0.000784	0.000841	11.507
D	r_2	B	s	s1	1.503	1.504	3.007
D	r_5	B	s	s1	6.505	6.505	8.008
E	r_4	A	u	u1	3.006	3.007	4.508
F	r_7	A	v	v1	8.008	8.008	9.511

interface. If required, another non-functional component can be bound to the Monitor Controller to execute this post-processing, thus avoiding to add too much processing task to the Monitor Controller.

C. Example

Consider the example from Fig.7. A client application represented by component Z makes a call on server interface p of component A, and generates a possible execution path displayed in Fig.8.

The path is implicitly available as displayed in Fig.8, thought from that view it is not clear how the calls r_3 and r_6 are related to r_2 and r_5. It may have happened that while serving r_2, D made both requests r_3 and r_6, and while serving r_5, it did not make any additional calls ; or it may have happened that while serving r_2, D made one of the requests, and while serving r_5 it made the other.

The final state of logs, is shown in Table II. All tables are grouped for displaying purposes, but it must be noted that each Monitor Controller keeps a copy of its own log, and no particular Monitor Controller has the grouped information of all logs at once.

By following the path from the Call Log, which can be followed through NF bindings as explained in Section IV-A, it is possible to find the actual sequence of calls. With this information it is possible to create the effective call tree as displayed in Fig.9.

The timing information is shown in Table II. From these

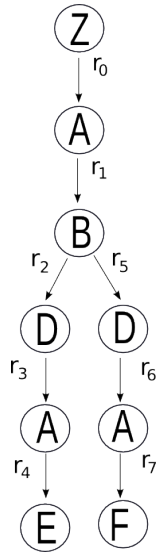

Fig. 9. Calling tree representing the path of the request r_0 initiated by component Z in Fig.8. The actual sequence of calls can be obtained from the logs shown in Table II

timestamps it is possible to determine the time that each component spent while serving a request. For example, the time that component A took to serve request r_0, as perceived by caller Z is 11.515s. This time includes the time required by all invocation that were generated while serving r_0.

Consider request r_5, which triggered the invocation of r_6 to composite A with timestamp 8.007. The composite forwards it as a new request r_7 to component F, with timestamp 8.008. The answer from component F is received by the composite A at 9.511, so r_7 was served in $1.503s$. That reply is forwarded to D and received at 9.512, which means that r_6 was served in $1.505s$.

When considering the total service time for a request involving calls to other components, the time of each invocation from the component that made the original call must be taken into account. Consider request r_1, which triggers two calls from component B: r_2 and r_5. The service time for r_1 must be at least the sum of the service time for r_2 and for r_5. In turn, the service time for r_5, as exemplified above, includes at least the service time for r_6, which must include the service time r_7.

From the times displayed in the Request Log it is possible to check that components E and F take around $1.5s$ to serve a requests r_4 and r_7. The time reported by component B to serve r_1 includes the sum of both request r_2 and r_5 that were made to component D. The difference with the time reported by r_1 in component B accounts for the time that the component was performing internal work.

D. Discussion

As it has been pointed in section II, monitoring must be as low intrusive as possible. This affirmation is also acknowledged by other works [16], [17]. A valid question is the impact that this framework has over the performance of the monitored

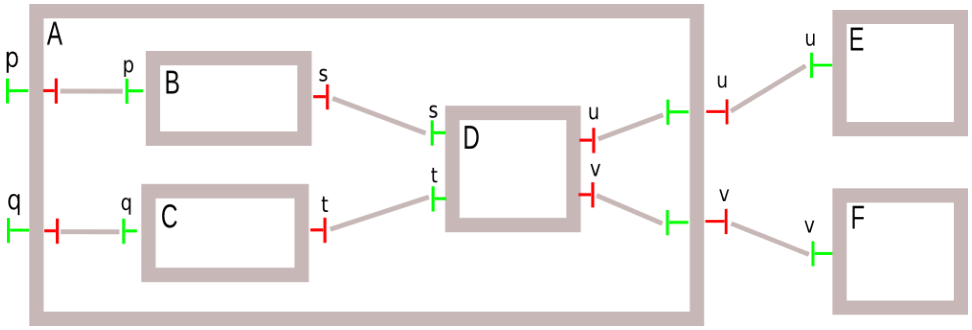

Fig. 7. Example GCM application

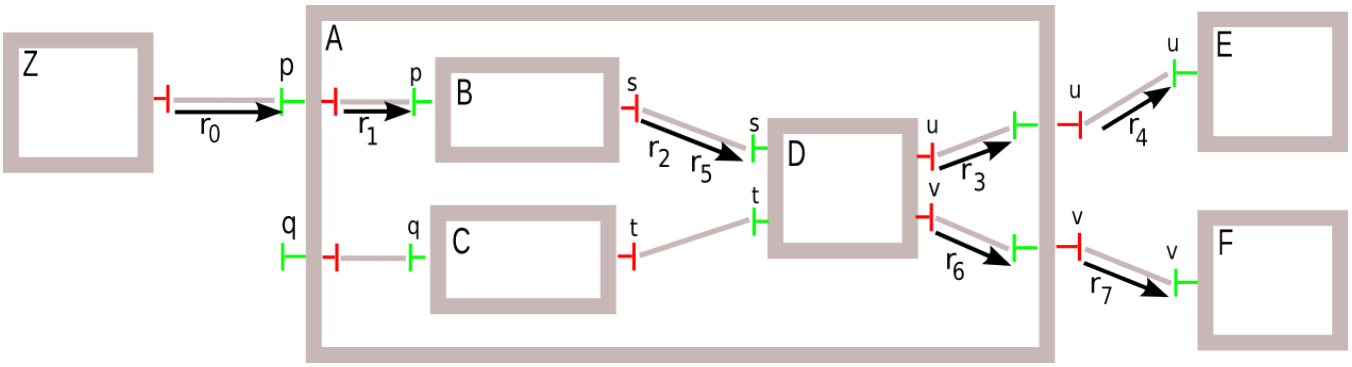

Fig. 8. Request path. Client `Z` calls `p` on composite `A`, which performs internal computation, and requires services from external components `E`, and `F`

application. Although we have not yet conducted experiments to effectively measure this impact in a large-scale application, we believe that performance would not suffer much harm, as the framework relies on asynchronous notifications, and it is implemented over a middleware that has been proved proficient in demanding large-scale scenarios [10].

Moreover, by implementing the Monitor Controller as a component controller, the monitoring tasks are delegated to other thread: the thread of the active object associated to the component controller. The functional service is therefore, not blocked by the monitoring task.

Also, if the monitoring task becomes too demanding, the Monitor Controller can be located remotely in another physical node, in a transparent manner for the application. This way, the only taks that remains in the monitored application is to generate the notifications, which is already done asynchronously.

V. Related Work

Schmid and Kroeger [1] describe a decentralised architecture for SCA, where a Manager component is associated to each workflow and to each service component, which is responsible of monitoring the behaviour of that workflow or service component with regard to its previously defined QoS requirements. This way a logically layered architecture for Service Level Monitoring is created, where the managers associated to workflow components communicate with the managers of the services participating in the worflow in order to enforce the QoS requirements that have been defined.

Aldinucci et al. [9] present an approach for managing NF concerns through Autonomic Managers in a hierarchical setting, which are attached to the individual software modules of the hierarchy, and by splitting the SLA into sub-contracts for the lower level managers. Managers at the higher levels can take more autonomous decisions than those at lower levels, which will behave according to the decisions taken at the higher levels. The top level manager receives from the user an SLA, which is split in sub-contracts, which will be given to the lower level managers.

Parsons et al. [17], [18] present and evaluate several approaches for extracting component-level interactions (CLI) in component-based Java applications. These approaches allow to extract runtime paths inside a complex application, and provide information that can be used to better understand and tune the targeted systems. Among them, the COMPAS tool allows to perform adaptive monitoring in a complex J2EE application. Like our framework, the information extracted can be used to feed an autonomic management system that can self-adapt the runtime configuration, and also can be used to perform problem diagnosis.

Application Response Measurement (ARM) [19] is a standard for monitoring and diagnose application response time in business applications. It provides C and Java bindings. By using the ARM API, applications can be instrumented with calls to ARM Agents, which make it available to other management and analysis applications. Its principles have been also applied to Web Services [20], [21] and multi-tier

environments [22] to correlate calls and monitor performance. Neither of these approaches, though, targets the scalability and flexibility of our solution.

VI. Conclusions

In this work, we presented a framework to collect performance information from components in an SCA architecture, and describe our current prototype implementation through a middleware that implements a component model with asynchronous invocations, which can be used as an SCA compliant platform.

The information is collected by listening to notifications of events, which are transmitted asynchronously by the middleware, with minimal intrusion in the application functional activity. The information collected is stored in a distributed way in each component improving the scalability of the approach. By implementing the monitor entities as components, which can be remotely located by the middleware in a transparent way, we aim for better flexibility and dynamicity. Finally, the information stored is made available for other components or external applications by using non-functional interfaces, and asynchronous JMX MBeans as provided by the middleware.

For the moment we are experimenting and taking measures for a small application as a proof of concept, but we are in the process to extend it to an application comprising a larger number of services, to demonstrate the feasibility of our approach.

By applying the appropriate probes, it is possible to modify the MC design to collect information for other aspects of an SLA, not only performance, but also provenance and security, for instance.

By exposing the information collected, this framework can provide the data required to enable SLA monitoring, which can feed mechanisms for QoS-aware service composition [2]. It can also provide the information required for an autonomic system to take decisions and, if needed, reconfigure the application to ensure compliance to an SLA.

This work opens the way to more distributed and scalable supports for SCA-based SOA, as also targeted by [8]. It leverages the GCM model and the GCM/ProActive implementation as feasible technical solutions to implement largely distributed and dynamically monitorable, and thus adaptable, SCA applications, using SLA monitoring techniques that are also distributed and scalable.

References

[1] M. Schmid and R. Kröger, "Decentralised QoS-Management in Service Oriented Architectures," in *DAIS*, 2008, pp. 44–57.

[2] M. Alrifai and T. Risse, "Combining global optimization with local selection for efficient QoS-aware service composition," in *WWW*, 2009, pp. 881–890.

[3] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *IEEE Computer*, vol. 36, no. 1, 2003.

[4] M. Schmid, K. Geihs, and W. Allee, "Self-Organisation in the Context of QoS Management in Service Oriented Architectures," in *Proceedings of the 13th Annual Workshop of HP OpenView University Association, Hosted by University of Nice at Cote dAzur*. Citeseer, 2006, pp. 153–164.

[5] "Service Component Architecture Specifications," March 2007. [Online]. Available: http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications

[6] "Apache Tuscany." [Online]. Available: http://tuscany.apache.org/

[7] "IBM Websphere Application Server." [Online]. Available: http://www-01.ibm.com/software/webservers/appserv/was/

[8] L. Seinturier, P. Merle, D. Fournier, N. Dolet, V. Schiavoni, and J.-B. Stefani, "Reconfigurable SCA Applications with the FraSCAti Platform," in *6th IEEE International Conference on Service Computing (SCC'09)*. Bangalore Inde: IEEE, 2009, pp. 268–275, IST FP7 IP SOA4All.

[9] M. Aldinucci, M. Danelutto, and P. Kilpatrick, "Autonomic management of non-functional concerns in distributed and parallel application programming," in *Proc. of Intl. Parallel and Distributed Processing Symposium (IPDPS)*. Rome, Italy: IEEE, May 2009.

[10] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici, *Grid Computing: Software Environments and Tools*. Springer-Verlag, January 2006, ch. Programming, Deploying, Composing, for the Grid, ISBN 978-1-85233-998-2.

[11] F. Baude, V. Legrand-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, june 2007, pp. 19–26.

[12] 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*, vol. 64, no. 1-2, pp. 5–24, 2009.

[13] 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), special issue on "Experiences with Auto-adaptive and Reconfigurable Systems*, vol. 36, 2006.

[14] F. Baude, D. Caromel, L. Henrio, and P. Naoumenko, *A Flexible Model And Implementation Of Component Controllers*, ser. Coregrid. Springer, 2008, ISBN 978-0-387-78447-2.

[15] F. Baude, L. Henrio, and P. Naoumenko, "Structural reconfiguration: an autonomic strategy for GCM components," in *5th International Conference on Autonomic and Autonomous Systems(ICAS 2009)*. IEEE Xplore, 2009.

[16] S. Agarwala, Y. Chen, D. Milojicic, and K. Schwan, "QMON: QoS- and utility-aware monitoring in enterprise systems," in *The 3rd IEEE International Conference on Autonomic Computing*. Citeseer, 2006, pp. 124–133.

[17] T. Parsons, A. Mos, M. Trofin, T. Gschwind, and J. Murphy, "Extracting interactions in component-based systems," *IEEE Trans. Software Eng.*, vol. 34, no. 6, pp. 783–799, 2008.

[18] T. Parsons, A. Mos, and J. Murphy, "Non-intrusive end-to-end runtime path tracing for J2EE systems," *IEE Proceedings Software*, vol. 153, no. 4, p. 149, 2006.

[19] M. Johnson, "Monitoring and diagnosing applications with ARM 4.0," in *Int. CMG Conference, Computer Measurement Group*. Citeseer, 2004, pp. 473–484.

[20] J. Turner, D. Bacigalupo, S. Jarvis, D. Dillenberger, and G. Nudd, "Application Response Measurement of Distributed Web Services," *International Journal of Computer Resource Measurement*, vol. 108, pp. 45–55, 2002.

[21] J. Schaefer, "An Approach for Fine-Grained Web Service Performance Monitoring," *Lecture Notes in Computer Science*, vol. 4025, p. 169, 2006.

[22] M. Schmid, M. Thoss, T. Termin, and R. Kroeger, "A generic application-oriented performance instrumentation for multi-tier environments," in *IEEE Intl. Symposium on Integrated Network Management*. Citeseer, 2007, pp. 304–313.