**Title:** Specifications of policies and strategies for distributed and multi-level adaptation

**Authors:** INRIA, CNR, FBK, UniDue, TUW, UOC, SZTAKI

**Editor:** Erwan Daubert, Guillaume Gauvrit (INRIA)

**Reviewers:** Zsolt Nemeth (SZTAKI)  
Salima Benbernou (Université de Paris Descartes)

**Identifier:** Deliverable # CD-JRA-2.3.8

**Type:** Deliverable

**Version:** 1

**Date:** 07 March 2012

**Status:** Final

**Class:** External

---

**Management Summary**

This deliverable aims to present the research progress of the project partners about distributed and multi-level (also known as cross layer) adaptation. Previous works from the deliverables CD-JRA-1.2.2, CD-JRA-1.2.3, CD-JRA-2.3.2, CD-JRA-2.3.4, CD-JRA-2.3.6 provide the bases for local adaptation and monitoring. Local adaptations are done on a part of a system taken in isolation from the rest. In this deliverable the adaptation process is extended to take into account the distribution and all the layers (all the part) of a system (hardware, networks, operating systems, middleware, services, workflows...) in order to provide coherent and concurrent adaptations.

The research results are presented through the summaries of joint papers that are classified according to the solutions they provide to manage distributed and multi-level adaptation.
Members of the S-Cube consortium:

University of Duisburg-Essen (Coordinator) Germany
Tilburg University Netherlands
City University London U.K.
Consiglio Nazionale delle Ricerche Italy
Center for Scientific and Technological Research Italy
The French National Institute for Research in Computer Science and Control France
Lero - The Irish Software Engineering Research Centre Ireland
Politecnico di Milano Italy
MTA SZTAKI – Computer and Automation Research Institute Hungary
Vienna University of Technology Austria
Université Claude Bernard Lyon France
University of Crete Greece
Universidad Politécnica de Madrid Spain
University of Stuttgart Germany
University of Hamburg Germany
Vrije Universiteit Amsterdam Netherlands

Published S-Cube documents

All public S-Cube deliverables are available from the S-Cube Web Portal at the following URL:
http://www.s-cube-network.eu/results/deliverables/
The S-Cube Deliverable Series

Vision and Objectives of S-Cube

The Software Services and Systems Network (S-Cube) will establish a unified, multidisciplinary, vibrant research community which will enable Europe to lead the software-services revolution, helping shape the software-service based Internet which is the backbone of our future interactive society.

By integrating diverse research communities, S-Cube intends to achieve world-wide scientific excellence in a field that is critical for European competitiveness. S-Cube will accomplish its aims by meeting the following objectives:

- Re-aligning, re-shaping and integrating research agendas of key European players from diverse research areas and by synthesizing and integrating diversified knowledge, thereby establishing a long-lasting foundation for steering research and for achieving innovation at the highest level.
- Inaugurating a Europe-wide common program of education and training for researchers and industry thereby creating a common culture that will have a profound impact on the future of the field.
- Establishing a pro-active mobility plan to enable cross-fertilisation and thereby fostering the integration of research communities and the establishment of a common software services research culture.
- Establishing trust relationships with industry via European Technology Platforms (specifically NESSI) to achieve a catalytic effect in shaping European research, strengthening industrial competitiveness and addressing main societal challenges.
- Defining a broader research vision and perspective that will shape the software-service based Internet of the future and will accelerate economic growth and improve the living conditions of European citizens.

S-Cube will produce an integrated research community of international reputation and acclaim that will help define the future shape of the field of software services which is of critical for European competitiveness. S-Cube will provide service engineering methodologies which facilitate the development, deployment and adjustment of sophisticated hybrid service-based systems that cannot be addressed with today’s limited software engineering approaches. S-Cube will further introduce an advanced training program for researchers and practitioners. Finally, S-Cube intends to bring strategic added value to European industry by using industry best-practice models and by implementing research results into pilot business cases and prototype systems.

Contents

1 Introduction 6
  1.1 Background ................................................................. 6
    1.1.1 Connections to the JRA-WP-2.3 research architecture .......... 6
    1.1.2 Relationship to other deliverables and workpackages ......... 7
  1.2 Deliverable Structure .................................................. 8

2 Cross-Layers Adaptations 9
  2.1 Cross-layer adaptation .................................................. 9
    2.1.1 Adaptation process .................................................. 9
    2.1.2 Adaptation capabilities and layers ................................ 10
    2.1.3 Adaptation layers .................................................... 10
  2.2 The needs of cross layer adaptations ................................ 11
  2.3 How to manage multi-level/cross layer adaptations ................. 12

3 Contributions 13
  3.1 Overview of the contributions ....................................... 13
  3.2 A Chemical Based Middleware for Workflow Instantiation and Execution .................................................. 15
  3.3 The chemical machine: an interpreter for the Higher Order Chemical Language ................................................. 15
  3.4 Distribution and Self-Adaptation of a Framework for Dynamic Adaptation of Services .............................................. 16
  3.5 Combining SLA Prediction and Cross Layer Adaptation for Preventing SLA Violations ............................................... 16
  3.6 ECMAF: An Event-Based Cross-Layer Service Monitoring and Adaptation Framework ................................................ 17
  3.7 A QoS Assurance Framework for Distributed Infrastructures .... 17
  3.8 A Self-Adaptable Approach for Easing the Development of Grid-Oriented Services ............................................... 17
  3.9 CLAM: Cross-layer Adaptation Manager for Service-Based Applications ................................................................. 18
  3.10 Stepwise and Asynchronous Runtime Optimization of Web Service Compositions ...................................................... 18
  3.11 FCM: an Architecture for Integrating IaaS Cloud Systems .... 19
  3.12 Integrated Monitoring Approach for Seamless Service Provisioning in Federated Clouds ........................................... 19

4 Conclusion 21

Bibliography 24

5 Attached Papers 25
  5.1 A Chemical Based Middleware for Workflow Instantiation and Execution .................................................. 26
  5.2 The chemical machine: an interpreter for the Higher Order Chemical Language ................................................. 38
  5.3 Distribution and Self-Adaptation of a Framework for Dynamic Adaptation of Services .............................................. 48
  5.4 Combining SLA Prediction and Cross Layer Adaptation for Preventing SLA Violations ............................................... 54
  5.5 ECMAF: An Event-Based Cross-Layer Service Monitoring and Adaptation Framework ................................................ 60
  5.6 A QoS Assurance Framework for Distributed Infrastructures .... 75
  5.7 A Self-Adaptable Approach for Easing the Development of Grid-Oriented Services .... 83
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.8</td>
<td>CLAM: Cross-layer Adaptation Manager for Service-Based Applications</td>
<td>90</td>
</tr>
<tr>
<td>5.9</td>
<td>Stepwise and Asynchronous Runtime Optimization of Web Service Compositions</td>
<td>97</td>
</tr>
<tr>
<td>5.10</td>
<td>FCM: an Architecture for Integrating IaaS Cloud Systems</td>
<td>105</td>
</tr>
<tr>
<td>5.11</td>
<td>Integrated Monitoring Approach for Seamless Service Provisioning in Federated Clouds</td>
<td>111</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This document is part of the deliverable series of S-Cube, the Software Services and Systems Network. Self-* Service Infrastructure and Service Discovery Support (WP-JRA-2.3): Work on service infrastructures will define policies, monitoring and redeployment techniques, for self-adaptive and self-healing services. This will strongly rely on input from WP-JRA-2.2 to design policies for adaptive service composition. In addition, this workpackage will specify and develop registry support mechanisms for service metadata, QoS attributes, service composition, and a federation of service registries.

This deliverable named “Specifications of policies and strategies for distributed and multi-level adaptation” reports on our findings about the reasonable policies to use in multi-level (also called cross-layer) adaptation when applied to distributed systems like Grids and clouds.

1.1 Background

As it was stated in CD-JRA-1.2.1 [5]: “The dynamic nature of the business world highlights the continuous pressure to reduce expenses, to increase revenues, to generate profits, and to remain competitive. This requires Web services to be highly reactive and adaptive. It should be equipped with mechanisms to ensure that their constituent component Web services are able to adapt to meet changing requirements. In fact, services are subject to constant changes and variations. Services can evolve due to changes in structures (attributes and operations), in behavior (when services are interacting) and policies.” There is a strong consensus on the fact that services and service infrastructures must be able to perform (self-) adaptation which is the topic of WP-JRA-2.3.

This deliverable focuses on the way to adapt large scale systems taking into account the requirements of the users as well as the constraints of the providers of each part of the systems such as services, middlewares, hardwares, networks... As it is needed to take into account all these elements, it is obviously mandatory to be able to monitor and adapt them. That’s why the works described here complete, extend or at least use the existing solutions presented on previous deliverable of the scube project (see the list 1.1.2).

1.1.1 Connections to the JRA-WP-2.3 research architecture

Research work in WP-JRA-2.3 is driven by the Work Package vision that structures the research work internally. Figure 1.1 illustrates the overall research architecture of WP-JRA-2.3: research on service infrastructures is comprised in three threads, Service Discovery, Service Registries and Service Execution. Orthogonally different approaches are separated in three layers.

- Service Discovery Thread (A) - Service discovery is a fundamental element of service-oriented architectures, services heavily rely on it to enable the execution of service-based applications. Novel discovery mechanisms must be able to deal with millions of services. Additionally, these
discovery mechanisms need to consider new constraints, which are not prevalent today, such as Quality of Experience requirements and expectations of users, geographical constraints, pricing and contractual issues, or invocability.

- Service Registry Research Thread (B) - Service registries are tools for the implementation of loosely-coupled service-based systems. The next generation of registries for Internet-scale service ecosystems are emerging, where fault tolerance and scalability of registries is of eminent importance. Autonomic registries need to be able to form loose federations, which are able to work in spite of heavy load or faults. Additionally, a richer set of metadata is needed in order to capture novel aspects such as self-adaptation, user feedback evaluation, or Internet-scale process discovery. Another research topic is the dissemination of metadata: the distributed and heterogeneous nature of these ecosystems asks for new dissemination methods between physically and logically disjoint registry entities, which work in spite of missing, untrusted, inconsistent and wrong metadata.

- Runtime Environment Research Thread (C) - There is an obvious need for automatic, autonomic approaches at run-time. As opposed to current approaches we envision an infrastructure that is able to adapt autonomously and dynamically to changing conditions. Such adaptation should be supported by past experience, should be able to take into consideration a complex set of conditions and their correlations, act proactively to avoid problems before they can occur and have a long lasting, stabilizing effect.

![Figure 1.1: WP-JRA-2.3 Research Architecture](image)

In alignment with the lifecycle aspect of the Integrated Research Framework, the presented work, is related mostly to runtime activities hence, to research topics in thread C in Figure 1.1. All these works target the ”Multi-level and self-adaptation” research challenge.

### 1.1.2 Relationship to other deliverables and workpackages

Some previous deliverables provides various information that fits with the content of this deliverable:

- CD-JRA-1.2.2 [7] untitled “Taxonomy of Adaptation Principles and Mechanisms” provides a comprehensive and holistic overview of the knowledge and concepts in the field of adaptation and monitoring. Those concepts are used and referred to in this CD-JRA-2.3.8 deliverable.
• PO-JRA-1.2.3 [30] untitled “Baseline of Adaptation and Monitoring Principles, Techniques, and Methodologies across Functional SBA Layers” provides some information about the requirements for the multi-level adaptation and monitoring.

• CD-JRA-2.3.2 [8] untitled “Basic requirements for self-healing services and decision support for local adaptation” captures the requirements for the design and realization of self-healing services within the S-Cube architecture. These foundations are used in the following deliverables of the “Infrastructure Mechanisms for the Run-Time Adaptation of Services” research thread.

• CD-JRA-2.3.4 [9] untitled “Decision support for local adaptation” documents research results related to making services adaptable and self-aware while adhering to a set of given specifications and mechanisms; it also investigated the applicability of certain policies to trigger local adaptation mechanisms. This decision support is extended to enable distributed adaptation in the CD-JRA-2.3.8 deliverable.

• CD-JRA-2.3.6 [10] untitled “policies and strategies for adaptation from the infrastructure” describes the elements of infrastructure mechanisms that enable or support the specification of policies and strategies able to be used to adapt the infrastructure. The works presented in the CD-JRA-2.3.8 deliverable build upon and extend these mechanisms to make distributed and multi-level adaptations possible.

As you can see with the relationship to the other deliverables, the works described in this deliverable is clearly related to the works proposed on the WP-JRA-1.2 that define methodologies to build cross-layer adaptation and monitoring systems. It is also related with WP-JRA-1.3 that define “principles, techniques and methods to ensuring end-to-end quality provision and SLA conformance by taking a holistic view on service infrastructure, service composition and coordination, and business process management and by employing SLAs as key elements to guide the integrated, cross-layer monitoring, and adaptation”.

1.2 Deliverable Structure

Firstly, section 1.1.2 describes the relationship with other workpackages and deliverables. In section 2.1, we remind the definition of the adaptation, the layers, the multi-level adaptation also known as cross layer adaptation (see the end of section 2.1) and provide some details about each of them. Then, in section 2.2, we provide some concrete example which highlights the need of cross-layer adaptation and especially on distributed systems. In section 2.3, we present very basic schemes to manage cross layer adaptation. Finally, in chapter 3, we present the work done in this workpackage to manage distributed and cross layer adaptation.
Chapter 2

Cross-Layers Adaptations

2.1 Cross-layer adaptation

2.1.1 Adaptation process

To define cross-layer adaptation, we previously need to define what is adaptation. A self-adaptive software makes decisions on its own, using high-level policies; it will constantly check and optimize its status and automatically adapt itself to an evolving context. The figure 2.1 describes an adaptation framework following the autonomic manager described by Kephart and Chess [21].

In this framework, we identify 4 phases called Monitoring (described in [7], Analysis (also called Decision [9, 10], Planning and Execution (MAPE). Each phase triggering the next one: the monitoring gather contextual information used by the analyzing part to decide whether an adaptation is needed or not; from this need, the planning part builds an execution plan to be executed by the execution part.

The contextual information is gathered by probes through events and measures. Events can trigger adaptation while measures are done on demand by the analyzing part when complementary information is needed. The monitoring is not only platform specific, it can also be application or domain specific when adaptation is not due to resources. The application itself can be monitored, for self-healing purpose for example, by a machine learning software, the user or by using ad-hoc metrics.

When receiving an event, the analysis phase chooses if an adaptation is needed by following a specific decision policy and using some information requested to the system. By using decision policy, the analysis phase is generic and can be used for different systems; only the policy is specific to each system adaptation.

Once an adaptation is chosen, the analysis sends a strategy to the planning part. The planning has to work out how to apply the strategy to the system to adapt; the implementation of the planning refers to a guide. This guide provides some information about how to decompose the strategies into elementary tasks to be executed. Like the decision policies, the planning guide may be specific to each adaptation system and so the planning phase is generic.

Then the plan of actions is sent to the executing part. Its role is to execute each action specified in the plan, taking into account the execution of the system to adapt. To do so, the execution phase interact with the system to adapt and may intercept the execution flow to execute adaptation actions.

The system described on the framework may represent something different according to points of view. In the case of Service-Based Applications (SBA), the system is most of time viewed as the application itself but it can also encompass the underlying architecture that host the services. Indeed this architecture is composed by a set of heterogeneous machines. A machine is able to communicate with some others in order to allow service(s) interoperation (communication). A machine can range from small devices like smartphones to more powerful ones like clouds, including personal computers and laptops. Moreover machines are running specific operating systems and some middlewares providing all the necessary core services needed such as communication facilities and resource brokering and they are...
distributed which means that operating systems and middlewares do not provide seamless data sharing. Indeed, delays can vary while communicating between machines and there will be some failures among machines. At last, applications are not only composed by services and business processes but may also include human beings.

2.1.2 Adaptation capabilities and layers

Every layer composing a system can be adapted in various ways. These ways can be classified according to several criteria (some of them are also described on [8]). The simplest one, “parameter adaptation” consists in changing the value of a parameter, for example to modify the frame rate on a video or to change the time between harvesting and processing in the Scube Wine production case study [6]. “Functional adaptation” that consists in changing a piece of code without visible logical consequences for the outside (the users). This means that this adaptation must not modify the interface for the service. For instance, one can change the bytecode of a Java service to improve its performance. Oppositely, “behavioural adaptation” consists in changing the algorithm of an element while having some visible effect for the outside. Then, “structural adaptation” concerns the modification of a composition of elements (such as services), by changing one or several links between them. For example, this can be used to replace a service by another.

All these adaptation capabilities may require various information and may also be executed with different techniques depending on what is adapted. Indeed, the adaptation of a parameter of the Operating System (the scheduling algorithm for example) doesn’t use the same operations as the adaptation of a parameter of a service, because these capabilities don’t have the same providers and impact different layers.

2.1.3 Adaptation layers

There exist various definition of layers. For example a layer may be defined for each level of the software stack (hardware, OS, middleware, services, workflow) but it is also possible to define layers according to the properties of the distribution of the system (one machine, one cluster, a set of connected computers on a same network, on different networks, ... ) because it can provide different adaptation capabilities. Moreover some layers may represent a set of layers due to the heterogeneity of the top layer. For example, it is possible to use different middlewares to host services and these middlewares may have different adaptation capabilities. At last, in the deliverable 2.3.2 [8], the authors define 3 other layers with the introduction of “human (or not) in the loop” properties (a human may replace the analysis phase in the adaptation process) and the need of coordination (e.g. services should be able to agree before any adaptation) before to adapt a set of related services.

Each layer is not clearly independent of the others because an adaptation at a specific layer may...
impact some others. For example, migrating a service from one server to another can impacts at least to layers: at the service layer, the composition between services has to take into account the new address of the service and the infrastructure layer may have to deploy a host (virtual machine or middleware) for the service. Such an adaptation can be useful to do load balancing and thus improve the QoS.

The cross layer adaptation can be defined in multiple ways. First of all, it can be seen as a set of adaptation that are applied at the same time on different layers. In that case, there are multiple adaptation process (MAPE) where each have access to the information of a single layer. It can also be seen as a single adaptation that impact multiple layers. In that case, there is only one adaptation process but each part of it have access to information of multiple layers.

We always speak about cross layer adaptation instead of multi-level adaptation because cross layer adaptation explicits that adaptations on different layers can have impacts and/or interactions with the others while multi-level only explicits there are some adaptations at different layers.

### 2.2 The needs of cross layer adaptations

Each level can and must adapt itself to fulfill the SLA constraints. Thus the cross layer adaptation is a strong issue to ensure the consistency and the coordination between all the adaptations. Indeed, when there are multiple adaptation systems, the consequences of an adaptation at a specific level may have impacts on the others. For example, the modification of the scheduling algorithm to promote a specific software (the one that are the most used) may decrease the CPU time allowed to others and so damage their quality of services. Moreover, the cross layer adaptation allows to optimize the adaptation itself by reducing the number of actions to execute or by selecting more appropriated actions at different levels.

The figures 2.2 to 2.4 details a usecase showing the difference between clearly independent adaptations and cross layer adaptation.

At the beginning, there are three applications that are executed on top of two machines. Each of these applications are service-based and execute on top of middlewares (middleware 1 and 2 are equivalent). App2 is made up of three services called Core which provides the set of functionality of the application, MM1 which is an active monitoring manager that looks at some sensors on a network and allows to get aggregate values of them, and Cache that is used as a cache between the Core and MM1 because MM1 needs times to compute an aggregate value whereas this value doesn’t evolve to much.

![Figure 2.2: State of the usecase before adaptation](image)

During the execution of the applications, the adaptation system at the middleware level decides to migrate the core service (Figure 2.3(a)) to reduce bandwidth used and so increases the response time between core and MM1 (or cache). Meanwhile, the App3 is stopped on the resource 2 and the adaptation system at the resource level decides to migrate the middleware 2 on resource 1 (Figure 2.3(b)).

If the two adaptations are seen as independent, then the core will be migrated on the middleware 2 and this middleware 2 will be migrated on the resource 1 (Figure 2.4(a)) while the migration of the middleware 2 would have been enough. This shows why taking into different layers in an adaptation is so important. Furthermore many possibilities exist to do these two adaptations and maybe one of them
is more efficient than the others. For example, if a service migration between middlewares is less time consuming than middleware migration on top of resources, it will be better to migrate MM1 and Cache to the middleware 1 instead of migrate the complete middleware 2 (Figure 2.4(b)).

2.3 How to manage multi-level/cross layer adaptations

Managing multi-level/cross layer adaptation can be done with different solutions according to the behavior of each adaptation system and the layers. All these approaches have some common requirements to be able to manage cross-layer. These requirements are described on [30]. One of the solutions consists to manage all the layers with only one adaptation system that is able to get all the needed information of all the elements in each layers to adapt the overall system. The complexity of this kind of solutions lies in the way to build such a global description of the system to have enough information to be able to take decisions.

Another solution consists to coordinate adaptation systems. Thus each adaptation system must ask validation and agreement to the other adaptation systems before to enact its adaptations. The complexity of this kind of solutions lies in the way to coordinate the adaptation systems. Indeed this solution is interesting when already existing adaptation systems must be used and when these adaptation systems have different implementations and different communication languages that are difficult to make interoperable.
Chapter 3

Contributions

3.1 Overview of the contributions

The core of this deliverable is constituted by a collection of published papers attached as appendices. These papers summarize research work carried out in or related to JRA-WP-2.3 according to its long-term goals, with a focus on distributed and multi-level adaptation. Mainly two complementary paradigms emerge to manage cross-layer adaptations: the approach based on chemical programming and another one based on distributed frameworks. This section presents the contributions in this order, followed works either using a different perspectives or targeting a specific problem.

Table 3.1 summarizes the contributed papers by the problem they address and the global approach taken to this end.
<table>
<thead>
<tr>
<th>Problem addressed</th>
<th>Approach</th>
<th>Ref.</th>
<th>Paper</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic workflow instantiation and execution</td>
<td>Chemical metaphor</td>
<td>Section 3.2</td>
<td>C. Di Napoli, M. Giordano, J.L. Pazat, and C. Wang. A chemical based middleware for workflow instantiation and execution. Towards a Service-Based Internet, 2010</td>
</tr>
</tbody>
</table>
3.2 A Chemical Based Middleware for Workflow Instantiation and Execution

The paper [13] proposes a middleware architecture for dynamic workflow instantiation and execution. The middleware integrates late binding service selection mechanisms with a workflow execution module. The selection mechanisms are carried out at two levels. The first occurs at the composition request level and it allows for the selection of abstract services by taking into account global QoS requirements on the composition of services, such as the overall delivery time and the total price. An abstract service is an interface specifying a required functionality of the workflow, and the corresponding QoS offered by a provider for that functionality without referring to a specific service implementation. The global QoS requirements usually concern the application and/or business levels.

The second selection mechanism takes place at the concrete binding level, and it allows to select actual service implementations, i.e. concrete services, based on some local QoS dimensions (i.e. referring to a single service), such as performance, reliability. These requirements usually concern system and/or middleware levels. The middleware architecture implementation is based on the chemical programming model, so that the dynamic selection of both abstract and concrete services is realized as an autonomous and always running process that can react to environmental changes affecting system behaviour in the form of chemical perturbations.

The middleware can be used as a component of an adaptation architecture making it possible to process adaptation needs coming from different levels. In the present work strategies for each level are not specified, since the main contribution is to model the dynamic selection of abstract and concrete services as a chemical process always running and always ready to process any change in the environment. In such a way adaptation needs coming from different levels can be continuously processed and the corresponding adaptation strategies can be inserted dynamically in the chemical system for each coming need, so providing support at the infrastructure level for multi-level adaptation.

3.3 The chemical machine: an interpreter for the Higher Order Chemical Language

Previous and ongoing research work has proven the viability of non-conventional approaches, particularly the chemical modeling to service composition and enactment. One of the research questions is how the chemical model described in the Higher Order Chemical Language (HOCL) can be realized and executed. This work focuses solely on creating an interpreter for HOCL that enables native execution of the entire language; the applicability of the HOCL to service composition has been investigated and published in other works, e.g. [12] [4] and reported in [10]. The major obstacle of realizing the chemical model is its intrinsic semantic difference from von Neumann architectures and imperative languages.

Such an unconventional computing model requires a full reconsideration of implementation means: we completely rejected all techniques related to imperative languages instead, carefully analyzed related models and distilled similar features that can be re-used in a chemical framework. In this way the concept of the chemical interpreter (i) follows the notion of abstract interpreter engines of declarative languages such as Prolog, Lisp, where HOCL is transformed into an intermediate language that is interpreted by the engine; (ii) is based on a production system that lends its state-of-the-art pattern matching mechanism nevertheless, needs modifications to support the hierarchical notion of knowledge base of the chemical semantics and fulfill other technical challenges; and (iii) several additional mechanism ensure the proper chemical behaviour.

A proof-of-concept implementation of the interpreter was realized that supports the entire HOCL language and has a graphical user interface and a basic support for tracing and debugging. The interpreter with its additional tools is able to support more complex experiments in the realm of the chemical model.
3.4 Distribution and Self-Adaptation of a Framework for Dynamic Adaptation of Services

The paper [1] presents a generic framework for self-adaptation of services and service based applications. The basic steps of an adaptation framework are Monitoring, Analysis, Planning and Execution, following the MAPE model proposed in [21] and presented in section 2.1.1. This work intend to improve this basic framework by refining each step of the MAPE model, in particular by providing elements that cope with the distribution of the application and the infrastructure layer. The adaptation system can itself be distributed for the purpose of scalability or to better match the heterogeneity of the environment. Moreover, it can be adaptable, allowing to take into account unforeseen situations.

This system called SAFDIS for Self-Adaptation For Distributed Services fully exploits the advantages of the framework concept [32]. It gives a frame, paradigms and rules to develop and implement adaptation mechanisms, as well as the liberty and the flexibility for the developer to specialize its system according to its specific needs. Using this framework, the task of developing concrete adaptation systems for some applications, services or infrastructures will be facilitated as many of the different elements that may be composed in adaptation systems are exposed, their interfaces clearly defined, the relationships between them coherently specified.

The SAFDIS framework is build as a set of services, providing functionalities useful to build an adaptation system. Not all functionalities are necessarily needed for each instantiation of an adaptation system. For instance we provide a negotiator service to negotiate the adaptation decisions when SAFDIS is distributed on several nodes; this service is not useful when SAFDIS is build as a unique centralized adaptation system.

In the context of the S-Cube infrastructure, the work presented in the paper addresses the Multi-level Adaptation research challenge of WP2.3, since it proposes a framework for building self-adaptable services that is aware of the different layers. Moreover, it enables users of this framework to propose and fine-tune adaptation policies and strategies for their service-based applications.

3.5 Combining SLA Prediction and Cross Layer Adaptation for Preventing SLA Violations

Service-based Applications (SBA) are deployed in highly dynamic and distributed settings, where various parts of the constituent components – services and their infrastructure – are controlled by different third parties. In such a loosely coupled environment, adaptation capabilities are needed to manage deviations and unforeseen situations which might lead to negative consequences (e.g. contractual penalties). Current approaches either focus on cross-layer-adaptation or the prevention of SLA violations. In contrast to this, the approach presented in this paper combines both. The paper presents an architecture as a generic framework for the management of arising problems during service execution. Multiple adaptation mechanisms are available to react on adaptation needs, acting on different layers of the SBA (including e.g. the composition layer and the infrastructure layer). The final goal of the cross-layer adaptation capability is to avoid the violation of agreed Service Level (in SLAs) and thus ensure the benefits of SBAs for both customers and providers. The novelty of the approach is the exploitation of all SBA layers (BPM, SCC and SI) for the prevention of SLA violations. The identification of adaptation needs is based on SLA prediction, which uses assumptions on the characteristics of the running execution context. Multiple adaptation mechanisms are available to react on the adaptation need, acting on different layers of the SBA. The adaptation strategy chooses the right adaptation mechanism, coordinated by a multi-agent community. In the future, we plan to further generalize the mechanism for handling the adaptation management cycle of detection, selection and enactment, and to incorporate more components, event types and adaptation capabilities.
3.6 ECMAF: An Event-Based Cross-Layer Service Monitoring and Adaptation Framework

The paper [34] outlines a cross-layer monitoring and adaptation framework that is based on monitored events. In addition, a layer-based taxonomy of adaptation-related events and a meta-model describing the dependencies among components of a cross-layer system are introduced to pinpoint the need for such a type of framework.

Although several techniques have been proposed towards monitoring and adaptation of Service-Based Applications (SBAs), few of them deal with cross-layer issues. This paper proposes a framework, able to monitor and adapt SBAs across all functional layers. This is achieved by using techniques, such as event monitoring and logging, event-pattern detection, and mapping between event patterns and appropriate adaptation strategies. In addition, a taxonomy of adaptation-related events and a meta-model describing the dependencies among the SBA layers are introduced in order to capture the cross-layer dimension of the framework. Finally, a specific case study is used to illustrate its functionality.

3.7 A QoS Assurance Framework for Distributed Infrastructures

Maintaining the promised QoS properties is a major concern for service providers in order to avoid losses and penalties. On the one hand, most research on QoS assurance in SOAs targets composite services, where managing QoS typically involves replacing services by other services more suitable for the composition [18]. However, such work does not address how individual, atomic services guarantee QoS properties, which unavoidably requires controlling the underlying infrastructure. On the other hand, further work that addresses QoS assurance of atomic services such as [2, 19, 17], fails to address the SLA life-cycle. This paper proposes the Qu4DS framework which addresses QoS assurance for atomic services by taking into account the complete SLA life-cycle. In particular, atomic services that build on large-scale distributed infrastructures, such as clusters, grids or clouds. Importantly, the framework supports dynamic adaptation, i.e., support for monitoring and automatically modifying service behavior and resource usage.

Currently, the Qu4DS framework supports providers in enforcing quality properties in distributed environments. The framework has three main features. First, Qu4DS provides flexible support for dynamic adaptation, necessary for maintaining agreed SLAs in the face of fluctuating environmental conditions. Second, the framework tackles the complete SLA life-cycle, from contract negotiation to service termination. Lastly, the framework lies on a modular design, extensible structure, cleanly separating the different QoS management functions and service implementations. In order to increase its applicability, Qu4DS builds on the standard SAGA API that provides a uniform and consistent interface to the most commonly used distributed functionality. A prototype of the framework has been developed; this prototype uses the XtreamOS [11] grid and leverages the MAPE Autonomic Computing control loop [21]. Furthermore, the paper presents initial experimental results that provide evidence that Qu4DS can effectively reduce SLA violations in dynamic environments.

3.8 A Self-Adaptable Approach for Easing the Development of Grid-Oriented Services

The Service-Oriented Architecture (SOA) proposes a support which addresses service composition as well as basic and high-level service management features. However, the SOA was not conceived to take into account non-trivial computational capabilities for services that require high-performance computing. On the other hand, grid computing [15] leverages low-cost and heterogeneous resources in order to provide a distributed infrastructure for high-performance computing in large-scale. Grids precisely define resource access through the management of Virtual Organizations (VO) in a dynamic fashion.
Thereby, grids offer an infrastructure suitable for supporting services that have high-performance computing needs. In order to use the grid, such services rely on the grid job abstraction by submitting and managing them through the grid interface. However, in spite of interesting efforts as the Simple Grid API (SAGA) [16] that addresses to ease the use of grids, grid usage still remains complex. This drawback becomes an inconvenience for the development of grid-oriented services.

This paper proposes a self-adaptable support for easing the conception of grid-oriented services. The self-adaptive aspect leverages Dynaco (Dynamic Adaptation for Components) [3] by following its component-based design. The architecture proposed takes advantage of the iPOJO [14] component model owing to its dynamic and flexible capabilities for developing and maintaining services on top of OSGi [29] platforms. iPOJO allows to separate the functional code from non-functional concerns which are modularly addressed in handlers. Thereby, this paper proposes an iPOJO handler which automatically addresses the management of jobs on grids. Moreover, the XtreemOS [11] grid platform is used as case study.

3.9 CLAM: Cross-layer Adaptation Manager for Service-Based Applications

CLAM [35] is a cross-layer adaptation management platform where adaptation actions are analyzed and validated for their consistency with the overall service-based system (SBS). It relies on two pillars:

1- Cross-layer system meta model: It is basically a directed graph where nodes represent the system elements from different layers and the edges represent the relations between these elements. Apart from system elements and their relationships, we have the tools associated with the elements. They are the analysis and adaptation mechanisms that are available for the SBS: (i) Analyzers check the compatibility of a new adaptation for a system element that they are associated with. (ii) Solvers get an incompatibility problem triggered by an analyzer and try to propose an adaptation to handle the problem. (iii) Enactors implement final adaptation strategies validated by CLAM.

2- Rules and the algorithm: We perform the entire adaptation analysis as a continuous execution of predefined rules. These rules and the overall algorithm determine how to navigate the SBS meta model to identify the system elements affected by an adaptation, and subsequently how to decide which analyzers and solvers to invoke, and finally how to gradually construct a cross-layer adaptation tree upon receiving results from those external tools. Cross-layer adaptation tree is basically the output of the analysis process where the branches correspond to alternative adaptation strategies validated by CLAM.

3.10 Stepwise and Asynchronous Runtime Optimization of Web Service Compositions

Providers of service compositions often guarantee important customers so called service level agreements (SLAs), which are essentially collections of target qualities (service level objectives, SLOs) and monetary penalties that go into effect if the promised target quality cannot be achieved. Hence, providers of service compositions have strong incentives to prevent cases of SLA violation. One promising approach to achieve this is predicting violations at runtime, before they have actually occurred, and using adaptation to prevent these violations [27, 20]. An important part of runtime adaptation is deciding which adaptations to apply, which we have modeled as an optimization problem in other work [25]. However, a limitation of the approach presented in that papers is that it is inherently assumed that the decision problem can be solved in time, before the first adaptation has to be applied. Even using fast meta-heuristics this is not guaranteed, especially so for shorter service compositions.

In this paper, we improve on this by proposing an asynchronous and step-wise optimization model, in which we do not generate a decision for all adaptations at once. Instead, we run the optimization in parallel to the execution of the composition. At so-called decision points we make a decision for only
those adaptations that absolutely need to be decided at that moment, update the optimization problem according to the made decisions, and continue optimizing for all remaining possibilities for adaptation. Note that the contribution presented here is not specific to the approach presented in [25]. Much more, the same ideas are applicable for other runtime adaptation approaches facing similar problems as well, e.g., [20].

### 3.11 FCM: an Architecture for Integrating IaaS Cloud Systems

Highly dynamic service environments require a novel infrastructure that can handle the on demand deployment and decommission of service instances. Cloud Computing offers simple and cost effective outsourcing in dynamic service environments and allows the construction of service-based applications extensible with the latest achievements of diverse research areas, such as Grid Computing, Service-oriented computing, business processes and virtualization. Virtual appliances (VA) encapsulate metadata (e.g., network requirements) with a complete software system (e.g., operating system, software libraries and applications) prepared for execution in virtual machines (VM). Infrastructure as a Service (IaaS) cloud systems provide access to remote computing infrastructures by allowing their users to instantiate virtual appliances on their virtualized resources as virtual machines.

Nowadays, several public and private IaaS systems co-exist and to accomplish dynamic service environments users frequently envisage a federated cloud that aggregates capabilities of various IaaS cloud providers. These IaaS systems are either offered by public service providers (like Amazon or RackSpace) or by smaller scale privately managed infrastructures. Therefore there is a need for an autonomic resource management solution that serves as an entry point to this cloud federation by providing transparent service execution for users. The following challenges are of great importance for such a mediator solution: varying load of user requests, enabling virtualized management of applications, establishing interoperability, minimizing Cloud usage costs and enhancing provider selection.

To address these issues, this paper proposes a layered architecture that incorporates the concepts of meta-brokering, cloud brokers and automated, on-demand service deployment. The meta-brokering component allows the system to interconnect the various cloud brokers available in the system. The cloud broker component is responsible for managing the virtual machine instances of the particular virtual appliances hosted on a specific infrastructure as a service provider. Our architecture organizes the virtual appliance distribution with the automatic service deployment component that can decompose virtual appliances to smaller parts. With the help of the minimal manageable virtual appliances the Virtual Machine Handler rebuilds these decomposed parts in the IaaS system chosen by the meta-broker. As a result, the cloud broker component uses the VM Handler to maintain the number of virtual machines according to the demand.

### 3.12 Integrated Monitoring Approach for Seamless Service Provisioning in Federated Clouds

Cloud Computing offers simple and cost effective outsourcing in dynamic service environments, and allows the construction of service-based applications using virtualization. By aggregating the capabilities of various IaaS cloud providers, federated clouds can be built. Managing such a distributed, heterogeneous environment requires sophisticated interoperation of adaptive coordinating components. However, user demands are frequently overextending the boundaries of a single cloud system. In these cases, they need to handle the differences between the various cloud providers and have to negotiate their requirements with multiple parties. Federated clouds aim at supporting these users by providing a single interface on which they can transparently handle the different cloud providers as they would do with a single cloud system.
This paper proposes an architecture to construct federated cloud systems that not only offers a single interface for its users but it automatically manages their virtual machines independently from the currently applied cloud system. We argue that efficient cloud selection in federated clouds requires a cloud monitoring subsystem that determines the actual health status of the available IaaS systems. We present an architecture that incorporates the concepts of on-demand service deployment, cloud brokering and meta-brokering, supported by an integrated monitoring solution. The meta-brokering component allows the system to interconnect the various cloud brokers available in the system. It is also responsible for selecting a proper execution environment managed by a cloud broker. This selection process relies on a sophisticated monitoring component, which provides up-to-date service availability and infrastructure reliability based on specific monitoring metrics. The cloud broker component is responsible for managing the virtual machine instances of the particular virtual appliances hosted on a specific infrastructure as a service provider. Our architecture also organizes virtual appliance distribution with its automatic service deployment component that can decompose and deliver virtual appliances in smaller parts. We also provided a simplified scenario for exemplifying the operation of our proposed solution using a minimal metric monitoring service. Our future work targets the evaluation of the proposed architecture with ordinary services deployed at different cloud providers.
Chapter 4

Conclusion

This deliverable presented our results related to elements and aspects of infrastructure level mechanisms to specify adaptation policies and strategies for distributed and multi-level adaptation. These mechanisms are targeting the objectives of the “Infrastructure Mechanisms for the Run-Time Adaptation of Services” research thread of the work package. As the last deliverable of this research thread, the proposed mechanisms build upon its previous deliverables, mainly the deliverable CD-JRA-2.3.6 “Specifications of policies and strategies for adaptation”.

The deliverable is a collection of scientific papers published in conference proceedings and organized along the research directions of WP-JRA-2.3. The papers all have been peer reviewed which ensures that the papers represent significant contributions to service-based system research and they demonstrate progress in the WP. The positioning of the papers within the adaptation framework, their relationship to the WP-JRA-2.3 research goals and vision and to other research WPs are exposed in Section 3. These research directions are focused around various solutions to manage cross layer and distributed. First, we have applied the chemical metaphor in 3.2 to reveal the possibilities of composing services while taking into account the multi-level or cross-layer properties and the distribution of the infrastructure. Next, we have demonstrated that framework-based approach [1, 34, 23, 24, 35, 28, 22] can be extended to manage and enact cross-layer and distributed dynamic adaptation of services based applications.

The papers presented in this deliverable proposed mechanisms that focused on distributed and multi-level aware components of service based infrastructures. They demonstrate that viable approach exists to build self-adaptive and self-healing services in the dynamic environment of the future internet.
Bibliography


[8] Basic requirements for self-healing services and decision support for local adaptation. (available on Scube website).


Chapter 5

Attached Papers
A Chemical Based Middleware for Workflow Instantiation and Execution

Claudia Di Napoli², Maurizio Giordano², Jean-Louis Pazat¹, and Chen Wang¹

¹ Université Européenne de Bretagne, INSA/INRIA/IRISA, F-35708 Rennes, France
{chen.wang, jean-louis.pazat}@irisa.fr
² Istituto di Cibernetica CNR, Via Campi Flegrei 34, 80078 Pozzuoli, Naples, Italy
{c.dinapoli, m.giordano}@cib.na.cnr.it

Abstract. Service-Oriented Architecture (SOA) is widely adopted today for building loosely coupled distributed applications. With this approach, new applications can be built by creating a business process that defines a concrete workflow composed of different partner services available via the network. In this scenario, the bindings between the business process and partner services are predefined by statically referencing the corresponding endpoints. This paper proposes a middleware architecture for dynamical workflow instantiation and execution. Using this middleware, partner services are selected and bound in the run-time and the aggregated QoS values are ensured to satisfy the requester’s end-to-end QoS requirement. The selection is based on both non-functional requirement (such as price and response time) of the global composition, and Quality of Service (QoS) performance of each candidate service. The implementation is based on chemical computing, a parallel and autonomic computing paradigm that allows to model workflow instantiation and execution as an evolving and adaptable process.

1 Introduction

Today, the need for designing loosely-coupled distributed applications requires service collaboration across enterprise boundaries. Service-Oriented Architecture (SOA) brings us standards-based, robust and interoperable solutions [1]. From the viewpoint of SOA, new applications can be built by composing a set of independent software modules called services. Within the scope of this paper, a service is a software system designed to support interoperable machine-to-machine interaction over computer networks. It can be seen as a black box providing certain functionality to its clients through a set of interfaces.

The traditional approach to build a composite application is to use an executable language (such as WS-BPEL [2]) to create a business process by defining a workflow that specifies business logic and execution order. In this context, the bindings between the business process and all its partner services are statically

* The research leading to these results has received funding from the European Community’s Seventh Framework Programme [FP7/2007-2013] under grant agreement 215483 (S-CUBE).
predefined. However, it is very likely that different service providers can provide a given functionality under different quality levels related to some non-functional characteristics. These characteristics may change in time depending on both provider policies and consumer requirements. For example, the cost associated to a service can be different according to market conditions (driven by demand-supply mechanisms), or the response time may vary according to the workload of the provider. Given the dynamic nature of service non-functional characteristics, the possibility of configuring service composition dynamically becomes a crucial requirement when taking into account the dynamic execution environment.

This paper addresses the problem of on-demand selecting and binding partner services in response to dynamic requirements and circumstances. A middleware architecture is proposed for dynamic workflow instantiation and execution. In our approach, a workflow is described in an abstract manner by specifying the functional requirement of each workflow activity and the dependencies among them. Each execution request is associated with a global QoS requirement (such as total price and overall response time) specified by the client. Then, the instantiation process is carried out to construct an instantiated workflow by mapping each workflow activity to a candidate service provider. The instantiated workflow ensures that the aggregated QoS value of all the selected service providers can match the client’s end-to-end requirement. The implementation is based on chemical programming [3]. It is a parallel and autonomic programming paradigm that models computation as chemical reactions controlled by a set of rules and all molecules involved represent computing resources.

This paper is organized as follows: in Section 2, some of the main existing approaches for dynamic service selection are introduced. In Section 3, we propose a middleware architecture for workflow instantiation and execution. Section 4 presents the implementation of this middleware based on chemical computing. Finally, conclusions and future works are addressed in Section 5.

2 Related Work and Motivations

Recently, the problem of dynamically selecting partner services that meet user’s requirements in terms of both functional and non-functional characteristics has gained wide attention. Some research works have concentrated on the design of languages and ontologies to represent these characteristics. In this case, automatic mechanisms are implemented to select the appropriate services to build service composition. Mukhija et al. [4] present an approach for QoS specification and service provider selection. The selection algorithm takes into account both the trustworthiness of the provider and the relative benefit offered by a provider with respect to the requester-specified QoS criteria. In [5] the authors propose an extension to the ontology framework based on OWL-S, which enables defining the composite services at the abstract level and then at runtime filtering and plugging in the candidate services that are chosen from a service instance pool.

Other research works have studied the development of frameworks to dynamically select service implementations. The Sword project [6] explores techniques
for composing services using logical rules to express the inputs and outputs associated with services. A rule-based expert system is used to automatically determine whether a process could be implemented with the given services. It also returns a process plan that implements the composition. Maximilien et al. [7] propose a framework and ontology for dynamic Web Service selection based on software agents coupled with a QoS ontology. With this approach, participants can collaborate to determine each other’s service quality and trustworthiness. Keidl et al. [8] propose the serviceGlobe environment that implements dynamic service selection using UDDI’s notion of a tModel.

Other solutions aim at integrating service selection and execution. In [9] the authors present an architecture for dynamic Web service selection within a workflow enactment process. The dynamic selection of services is performed through a Proxy Service interacting with a Discovery Service and an Optimization Service. In [10] the eFlow system is proposed that allows nodes (activities) to have service selection rules. When the eFlow engine tries to execute an activity it calls a service broker which executes the service selection rules and returns a list of services (with ranking information).

Within this context, the present work proposes a middleware architecture that integrates late binding service selection mechanisms with a workflow execution module. The selection mechanisms are carried out at two levels. The first occurs at the composition request level and it allows selecting abstract services by taking into account global QoS requirements on the composition of services, such as the overall delivery time and the total price. Then, the second selection mechanism takes place at the concrete binding level. It allows further selecting suitable service implementation to forward the invocation request. The selection is based on some local QoS dimension (such as performance, reliability) so that each abstract service still meets the constraints derived by the global requirements. Using this middleware, dynamical selection of service implementations is an autonomous process that can react to environmental changes.

3 Middleware for Workflow Instantiation and Execution

As shown in Figure 1, the middleware is built upon the Web service implementation layer. All components in the proposed middleware are organized in two levels: the Service Representation level and the Workflow Execution level. The former resolves the abstract representation of concrete services in the middleware layer while the latter takes charge of workflow instantiation and execution.

3.1 Middleware Architecture

In this middleware, we assume that all the available functionalities are provided by a set of Abstract Service (AS). An AS is a description of functional interfaces related to a certain Web service rather than the relative implementation. Thus, in order to perform the real calculation, it groups a set of Concrete Services (CS) in order to forward the invocation requests (see AS1 in Figure 1). An AS can deliver a given functionality with different quality levels by publishing various
offers in the Registry component. The Registry acts as a directory maintaining the information of all currently available offers. An offer specifies both functional and non-functional characteristics of a service delivery.

A group of Abstract Business Processes (ABP) act as service consumers. An ABP, similar to a WS-BPEL business process, is a collection of interrelated tasks and activities that function in a logical sequence to achieve the ultimate business goal. The business logic and execution sequence are defined by an Abstract Workflow (AW). It is described by a directed acyclic graph whose nodes represent the functional requirements of the workflow activities and edges specify the dependency among those activities.

Each request to an ABP is associated with a global QoS constraint specified by the requester (such as how much he can offer to purchase a service delivery with the response time less than 10s). Before the execution, the abstract workflow is forwarded to the Workflow Instantiation (WI) component. This component is responsible for building a possible Instantiated Workflow (IWF) by assigning each workflow activity to a suitable AS. The selection of adequate AS is based on all currently available offers in the Registry and the global QoS constraints specified by the client. Our system aims at searching a feasible IWF that can satisfy the client’s requirement rather than the optimal one.

Once a feasible IWF is found, it is passed to the Execution Component which then takes charge of workflow execution. As proposed in [11], the execution of a workflow can be decentralized. IWF information is separated into several blocks and then forwarded to the relative abstract services. A partial IWF block for a partner service includes the information about its predecessor(s), successors(s) as well as the QoS value to be delivered. An abstract service works as an autonomic system: it receives the request from the predecessor service(s), calculates and then forwards the results to the successor service(s). The calculation recurs to the selection of a concrete service to forward the service invocation request. This selection is based on the historical invocation information.
Service provider can join and leave the middleware freely. The management of an AS is easy by using the User Console (UC), a user interface tool. To join the middleware system, a service provider needs to pass the URL address of the relative WSDL file as input, UC generates the code for the corresponding abstract service (including the interface descriptions and the connection to the implementation), and finally it deploys the code on the default or user specified server. Provider can also publish offers, monitor the execution by using UC.

The following sub-sections will highlight the service selection mechanisms performed in two stages: one is responsible for selecting abstract services to build an instantiated workflow, the other is to select a service implementation to carry out the real computation.

### 3.2 Selection of Abstract Service

A client requests the execution of a Service-Based Application (SBA) by specifying an abstract workflow. A request is composed of: 1) the functionality of each execution activity, 2) the dependence constraints among those activities and 3) user QoS requirements for the entire application. It is assumed that the global QoS can be computed by combining the non-functional parameters associated to each abstract service of the workflow. To instantiate a workflow, a WI component requires the information of all currently available offers from the registry. For each workflow activity, multiple offers may be available since the same service can be provided by different providers or under different QoS levels.

The WI component maps the currently available offers to each activity and then it checks the possibility to construct a fully instantiated workflow. Initially, each offer mapped to an activity represents a chain. The instantiation is a recursive process: smaller chains mapped to the abstract workflow are concatenated to form larger ones, or assembled around the split and merge nodes (representing workflow branches). The workflow instantiation process is driven by the objective of fulfilling the client’s global QoS request on the entire workflow. This is possible only when a QoS of an entire workflow can be expressed by combining the QoS of the single service offers. The instantiation process is not a sequential one that begins from the start node of the abstract workflow. Instead, workflow chains are aggregated and instantiated in a parallel and concurrent way.

The execution state of the Workflow Instantiation component at a given time is represented by the set of available service offers together with the partial results of the workflow instantiation process. The set of offers may change when new offers become available or others disappear from the system. With this design, the result of the instantiation process is not determined only by the initial state of the component, but also by its state evolution in time since the availability of new offers during the process may produce alternative instantiations of the abstract workflow.

### 3.3 Selection of Concrete Service

Similar to a letting agent, through which a tenancy agreement is made between a landlord and a tenant for the rental of a residential property, an abstract service
A Chemical Based Middleware for Workflow Instantiation and Execution

takes charge of making agreement between requester and a service implementation, either short-term or long-term. Thus, it has to connect to a number of concrete services in the similar way as a letting agent manages various residential properties. The concrete service can join and leave freely. To achieve this, it composes several *invokers*, each *invoker* takes the responsibility of communicating with a certain concrete service, such as forwarding the invocation request and retrieving the result. As a rule, only one invoker can be active and capable of forwarding the request message to the relative concrete service at a time. As a result, a concrete service is selected by activating the corresponding invoker.

Besides the invokers, an abstract service also defines three functional modules. If an invocation demand arrives to the abstract service, the *Evaluation Module* checks whether the concrete service connected by the currently active invoker can satisfy the user’s requirement. If not, it deactivates this invoker and informs the *Selection Module* to select another concrete service to avoid the penalty. The selection criteria is based on the historical invocation information. Some non-functional characteristics are monitored by the *Monitoring Module* for each service invocation. After the invocation, monitored results are stored as a record in the relative *invoker*. The *Selection Module* evaluates a concrete service according to these records information (i.e. the response time can reflect the workload of a concrete service as well as network condition). If a qualified concrete service is found, the relative invoker is activated so that the invocation demand can be forwarded to it.

4 Implementation

The implementation of this middleware architecture is based on the chemical programming paradigm inspired by the chemical metaphor. All the computing resources are represented by molecules and the computation can be seen as a series of chemical reactions controlled by a set of rules. Analogous to a chemical equation which reflects nature laws by specifying the reactants and resultants of a chemical reaction, a *rule* defines the law of calculation by specifying the input, output as well as the condition.

Each component in the middleware architecture in Figure 1 is implemented by an HOCL program, defined later on. An HOCL program defines a *multi-set* acting as a chemical reaction container where objects inside represent molecules. Multi-set extends the concept of “set” with “multiplicity”: an element can be present only once in a set whereas many times in a multi-set. A multi-set contains elements and rules to perform the computation. As an example, an abstract service defines a multi-set containing some invokers as molecules along with a set of rules to perform the functionalities of three functional modules. This implementation is highly distributed. All the multi-sets are implemented in a decentralized way and they have the ability to talk with each other by writing elements into remote multi-sets.

The chemical-based implementation enables the middleware to adapt to the dynamic environment. A multi-set is implemented as an autonomic system and
it works independently and concurrently to the others. Computation in a multi-set can be seen as a perpetual motion that never needs to stop or intervene. If some unexpected incident happens, due to the dynamic and unpredictable characteristic of distributed computing, certain rules are activated to take actions.

4.1 Higher Order Chemical Language

HOCL stands for Higher Order Chemical Language which implements chemical programming model. It is based on the $\gamma$-calculus [12] which extends the Gamma language with higher-order property [14]. In Figure 2, an HOCL program is defined to calculate the maximum number of a set of given integers.

```plaintext
let max = replace x::int, y::int by x if x>y in
< Max, 13, 45, 65, 54, 25, 2, 5, 4, 36 >
```

Fig. 2. HOCL program: calculate the maximum integer

An HOCL program is composed of two parts: Rule Definition (Line 1-5) and Multi-set Organization (Line 6). In HOCL, a multi-set is defined by a pair of operators "<" and ">", you can define all kinds of elements inside a multi-set such as data, rules as well as other multi-sets. From the chemical perspective, a multi-set is also called a solution that can be seen as a membrane so that all the molecules inside it cannot react with the ones outside it.

All the rules appearing in the multi-set have to be defined before. A rule defines how to rewrite a multi-set: it replaces a part of elements by new ones. As a result, rules are defined in the form of replace P by M if V. From the perspective of computer science, a rule plays the same role as a function. The let keyword gives the name of the function, replace lists all the input parameters while by keyword indicates the output. More information about how to write HOCL program can be found in [15].

The execution of an HOCL program has the following characteristics: firstly, it is non-deterministic. A rule reacts with the molecules that are chosen non-deterministically. Furthermore, it is parallel in nature. Thus, different reactions can be carried out simultaneously. Thirdly, it is automated and self-coordinated. The results of a certain reaction can activate other rules to perform further calculation. These characteristics make HOCL suitable for distributed applications programming.

4.2 Abstract Service Selection Using Chemical Rules

The chemical implementation of the workflow instantiation (WI) component uses the following chemical representations for input and output data:
- **Abstract Workflow** (input): it is defined in a "NODES" chemical solution that contains the abstract descriptions of workflow tasks and their dependences:

"NODES":<"Node":<"ID":Si, "TO":<T1,...>,"FROM":<F1,...>>,other nodes>

It contains a number of "Node" sub-solutions. Each "Node" sub-solution defines a certain task by specifying its functional requirement (Si) as well as its predecessor nodes ("FROM" sub-solution) and successor nodes ("TO" sub-solution).

- **Offer** (input): a chemical sub-solution representing the abstract service ei that offers the service interface Sm with a QoS ci:

"Offer":<ei:Sm,"QOS":ci>.

- **Instantiated Workflow** (output): a chemical sub-solution representing a composition of service offers:

"FIRST":<ei:Sm>,"LAST":<ej:Sn>,"SPLIT":<ek:Sp>,"MERGE":<el:Sq>,
<first-branch sub-solution>,<second-branch sub-solution>,"QOS":c,...>

ei:Sm and ej:Sn are abstract services selected for the first and last activity of the workflow subgraph, and ek:Sp and el:Sq are those associated to split and merge nodes. c is the QoS value computed by combining the QoS partial values of the abstract service offers selected by the instantiation process when building the composition. In the chemical notation the branched paths of the split are represented by nested sub-solutions.

In Figure 3, a workflow with associated abstract service offers is drawn on the top side, while the result of a full instantiated workflow is on the bottom of the figure. In Figure 4, the chemical notations of the abstract workflow, the offers and the result of the mapping between offers and tasks with reference to the example of Figure 3 are reported. Please note that the task-to-offer mapping is one-to-one. The WI component selects an offer for each task of the workflow as well as combining offers according to QoS constraints.

The chemical rules that implement the workflow instantiation process act recursively by concatenating simpler components, named Partial Instantiated Workflows (PIWFs), defined as follows: a PIWF is a set of abstract service offers that maps a subgraph of the workflow whose nodes have sources and sinks belonging to the same subgraph, except the first and last nodes which may have respectively sources and sinks outside the subgraph. According to this definition, a single node is a PIWF. Every service offer when put in the notation: "FIRST":<ei:Sm>,"LAST":<ei:Sm>,...> represents a PIWF by definition.

Abstract service compositions are built by means of two chemical rules. The first one, named chainrule concatenates two PIWFs linked by only one edge. The second rule, named splitrule assembles four PIWFs: the first one contains the split node as LAST node, the second one contains the merge node as FIRST node. The other two PIWFs represent the workflow paths from a split node to the associated merge node. A more detailed description of the chemical rules can be found in [16].
4.3 Concrete Service Selection Using Chemical Programming Model

A service implementation is selected by activating the corresponding invoker. An invoker can be seen as chemical-level reflection of a service implementation. It is implemented by an object which provides a group of functionalities through a set of interfaces. From the chemical level, this object can be regarded as a molecule which can participate in the chemical reactions. Chemical rules use its interfaces to operate on the invokers, such as activating or selecting an invoker. Here some of the principal interfaces are reported:

- **isValid/setValid/setInvalid.** To select an invoker, some interfaces are exposed to operate on its status. An invoker has two states: valid and invalid. `isValid` operation returns a boolean value indicating whether this invoker is active. `setValid` activates this invoker while `setInvalid` deactivates it.

- **getQoS/addQoS.** An invocation can be associated with multiple QoS requirements. To simplify the description, it is assumed that only one generic QoS parameter (you can regard it as price or response time) is monitored by the monitoring module for each invocation. The monitored data is stored in the relative invoker and can be accessed by a pair of interfaces. `addQoS` records the monitored data from the monitoring module into its built-in memory and `getQoS` returns the QoS value calculated based on the historical information. The calculation is based on a certain algorithm.

- **invoke.** This operation forwards the invocation request to the corresponding service implementation. It requires two parameters: one is the operation
name and the other is a set of parameters. These parameters are encapsulated into a SOAP message to invoke the corresponding operation.

An abstract service is implemented by a multi-set which contains multiple invoker objects. The functionality provided by different modules is performed by a set of rules. Figure 5 lists some of the major rules.

```plaintext
let evaluation = replace "QoS_Client_Req":qos, invoker by invoker.setInvalid(), "QoS_Client_Req":qos, "COMMAND":"SELECT" if invoker.isValid()==true && qos < invoker.getQoS() in
let select = replace "QoS_Client_Req":qos, invoker, "COMMAND":"SELECT" by invoker.setValid(), "QoS_Client_Req":qos, "COMMAND":"INVOKE" if invoker.isValid()==false && invoker.getQoS()<qos in
let invoke = replace invoker,"INVOKE":operation:<?p>,"COMMAND":"INVOKE" by invoker, "RESULT":<invoker.invoke(operation,p)> if invoker.isValid()==true in ...
```

Fig. 5. The Implementation of Concrete Services Selection

The rule evaluation implements the evaluation module. The expecting QoS delivery is expressed by "QoS_Client_Req":qos tuple. The value of qos is determined by the offer that is formerly published by this abstract service. The evaluation rule reacts with the current invoker (invoker.isValid()==true). If the current invoker cannot deliver the service to meet the expecting QoS requirement (qos<invoker.getQoS()), it will be deactivated (invoker.setInvalid()) and a COMMAND:"SELECT" tuple is generated. This tuple informs the system to carry out the selection process by activating the select rule which triggers a series of reactions in succession.

The rule select and invoke are defined in a similar way. Rule select gets an invoker, calculates its QoS performance and compare it with the expecting QoS requirement. If this invoker can meet the needs, it is set to "VALID" and a "COMMAND":"INVOKE" tuple is thrown, indicating the invocation can be performed. This tuple is caught by the rule invoke which will then forward the invocation request to the corresponding implementation through invoke interface of the active invoker.

5 Conclusion

In this paper, we presented a middleware architecture for dynamic workflow instantiation and execution. The run-time service selecting and binding mechanisms are implemented at two levels: the first occurs at the composition level
and it allows to build a full instantiated workflow in consideration of the global QoS requirement. The second is to select a qualified concrete service for each execution activity to guarantee the global QoS compliance. This middleware has implemented the following desirable properties: 1) all partner services are selected and bound dynamically for each execution; 2) client’s QoS requirements are taken into account in the selection of services, and 3) the execution of a workflow is decentralized.

The implementation of this middleware is based on chemical programming model. Each middleware component is implemented by a multi-set as an autonomic system adaptive to the fast changing execution environment. The execution can be viewed as perpetual motion that never need to stop. New elements and rules can be added on-demand to meet the new requirements. All components are implemented in a decentralized way and run independently and concurrently to each other. The interaction among them is performed by writing elements to remote multi-sets.

In the future, our work will concentrate on defining a SLA level in this chemical based middleware dealing with SLA negotiation and management. This level will provide the middleware with additional run-time service adaptation ability. We believe that chemical programming model is suitable for implementing service adaptation because of its dynamic, autonomic, independent execution process and distributed implementation.

References


Abstract. The notion of chemical computing has evolved for more than two decades. From the seminal idea several models, calculi and languages have been developed and there are various proposals for applying chemical models in distributed problem solving where some sort of autonomy, self-evolving nature and adaptation is sought. While there are some experimental chemical implementations, most of these proposals remained at the paper-and-pencil stage. This paper presents a general purpose interpreter for the Higher Order Chemical Language. The design follows that of logic/functional languages and bridges the gap between the highly abstract chemical model and the physical machine by an abstract interpreter engine. As a novel approach the engine is based on a modified hierarchical production system and turns away from imperative languages.

1 Introduction

The advent of large scale distributed systems (such as grids, service oriented architectures) introduced a group of problems that are hard to solve by humans or by any machinery in an exact way due to the very large number of entities, their heterogeneous nature, partial lack of information of their state, unpredictable, error prone behavior and many other factors. Approximately the same time appeared the notion of autonomic computing [17] where entities are supposed to monitor and control themselves according to some strategies: self-configuration, self-optimization, self-healing and self-protection. Since then a large number of reflective, self-* properties of computing entities have been proposed and realized. This new notion of computing naturally attracted non-conventional approaches; in fact the seminal paper [17] also took inspiration from the nervous system [13]. There is a large group of models that mimic various biological, chemical, physical, ethological processes and phenomenons or simply take them as metaphors.

In the chemical programming paradigm, instead of computing steps (instructions) and their strict order, a program is conceived as a chemical solution where data and procedures are molecules floating around and computation is a series of reactions between these molecules. Note, that in this case chemistry is just an inspiration or an abstract metaphor as opposed to chemical models (artificial chemistries) where computation closely simulates some chemical processes [12]. This vision of chemical computing is formalized in the $\gamma$-calculus [3] as (without the chemical guise) a declarative functional computational model where terms are commutative and associative.
The chemical model and the γ-family (the calculus and the related languages) has already been investigated in various distributed scenarios, like self-organizing systems [8] where a self-healing, self-optimizing and self-protecting mail system is studied. Grids are obviously a good target for applying the chemical model in some well-known problems like coordinating a ray-tracing example on desktop grids [7], enacting workflows on-the-fly with strong emphasis on dynamicity both in the environment and in the workflow structure [11] and modeling self-developing secure virtual organisations [2]. Recently service oriented techniques and clouds also attracted great attention and proposals like chemical based service orchestration [6], dynamic service composition [5], dynamic service composition with partial instantiations and re-using instantiations [18] and others. Note, that the chemical model in all these cases is not applied for problem solving (in terms of solving any computational tasks) but coordinates the execution so that it may exhibit some of the features of the chemical metaphor like timely response to events, adaptation, self-evolution, intrinsic concurrency, independency, maximum parallelism and many others.

Albeit application of the chemical metaphor in grids and service oriented systems is well studied and various concepts are elaborated, appropriate interpreter and development tools for executing programs expressed in the chemical metaphor are largely missing. Most of these models require framework that is (i) able to execute the code expressed in a chemical language and (ii) provides interfaces to the embedding system so that some processes can be controlled by the chemical program meanwhile monitored data can be gathered. The work introduced in this paper is focused on (i) and aimed at creating an interpreter that supports the entire Higher Order Chemical Language (HOCL), a language that is based on and extends the γ-calculus. While the chemical model is quite different from any other widespread computing models and languages, careful study revealed similarities in other paradigms and the combination of techniques related to declarative languages and those of production systems allowed a realization of the interpreter in a short development cycle. At deciding the implementation means attention was paid to (ii) so that the interpreter can be interfaced with various tools and environments in the future. The work is focusing on the design and realization of the interpreter. Establishing autonomic or adaptive behaviour in the chemical framework is on one hand presented in papers [7][11][5][8][18], etc., on the other hand related to the application of the interpreter and not presented here.

2 The chemical computational model

Most algorithms are expressed sequentially even if they describe inherently parallel activities. Gamma (General Abstract Model for Multiset Manipulation) [4] aimed at relaxing the artificial sequentializing of algorithms. It is a multiset rewriting system where the program is represented by a set of declarative rules that are atomic, fire independently and potentially simultaneously, according to local and actual conditions. There is no concept of any centralized control, ordering, serialization rather, the computation is carried out in a non-deterministic, self-evolving way. It has been shown in [4] that some fundamental problems of computer science (sorting, prime testing, string
processing, graph algorithms, etc.) can be expressed in Gamma in a concise and elegant way.

The \( \gamma \)-calculus is a formal definition of the chemical paradigm. The fundamental data structure is the multiset \( M \). \( \gamma \)-terms (molecules) are: variables \( x \), \( \gamma \)-abstractions \( \langle x \rangle .M \), multisets \( (M_1, M_2) \) and solutions \( \langle M \rangle \). Juxtaposition of \( \gamma \)-terms is commutative \( (M_1, M_2) \equiv (M_2, M_1) \) and associative \( (M_1, (M_2, M_3)) \equiv ((M_1, M_2), M_3) \). Commutativity and associativity are the properties that realize the 'Brownian-motion', i.e., the free distribution and unspecified reaction order among molecules. The \( \gamma \)-abstractions are the reactive molecules that can take other molecules or solutions and replace them. Due to the commutative and associative rules, the order of parameters is indifferent; molecules, solutions participating in the reaction are extracted by pattern matching – any of the matching ones may react. The semantics of a \( \gamma \)-reduction is \( \langle \gamma(x).M \rangle, \langle N \rangle \to \gamma[M[x := N]] \) i.e., the two reacting terms on the left hand side are replaced by the body of the \( \gamma \)-abstraction where each free occurrence of variable \( x \) is replaced by parameter \( N \) if \( N \) is inert. Reactions may depend on certain conditions expressed as \( C \) in \( \gamma(x)[C].M \) that can be reduced only if \( C \) evaluates to true before the reaction. Reactions can capture multiple molecules in a single atomic step. The universal symbol \( \omega \) matches any pattern. Reactions are governed by: (i) law of locality, i.e. if a reaction can occur, it will occur in the same way irrespectively to the environment; and (ii) membrane law, i.e. reactions can occur in nested solutions or in other words, solutions may contain sub-solutions separated by a membrane. The \( \gamma \)-calculus is a higher order model, where abstractions – just like any other molecules – can be passed as parameters or yielded as a result of a reduction [8][3].

The Higher Order Chemical Language (HOCL) [3] is a language based on the Gamma principles more precisely, the \( \gamma \)-calculus extended with expressions, types, pairs, empty solutions and names. HOCL uses the self-explanatory replace... by... if... construct to express rules. replace \( P \) by \( M \) if \( C \) formally corresponds to \( \gamma(P)[C].M \) with a major difference: while \( \gamma \)-abstractions are destroyed by the reactions, HOCL rules are n-shot and remain in the solution nevertheless, single-shot \( \gamma \)-style rules can also be added. replace... by... if... is followed by in \( \langle ... \rangle \) that specifies the solution the active molecule floats in. Notable features (extensions) of HOCL are: types,\( = \) that can be added to patterns for matching; pairs in form of \( A_1 : A_2 \) where \( A_1 \) and \( A_2 \) are atoms; and naming that allows to identify and hence, match rules, e.g. let \( inc = replace x \text{ by } x + 1 \text{ in } \langle 1, 2, 3, inc \rangle \) specifies an active molecule called \( inc \) which captures an integer and replaces it with its successor, floating in a solution together with integers 1, 2, 3. Some possible reduction steps can be (note, the model is non-deterministic, there are different possible execution paths):

\[
\langle 1, 2, 3, inc \rangle \to \langle 2, 2, 3, inc \rangle \to \langle 3, 2, 3, inc \rangle \to \langle 3, 2, 4, inc \rangle \to \langle 3, 3, 4, inc \rangle
\]

3 The concept of the chemical interpreter

In case of declarative languages, the semantics of the execution model and that of the underlying physical architecture is quite different therefore, they are usually executed via an abstract, hypothetic engine placed inbetween. The program is first transformed
(compiled) into the language of the abstract engine that successively interprets the input and executes it. From the programmer’s point of view, the abstract engine is a machine that is able to execute the high-level language natively, it hides all the details of the real physical machine whereas, the abstract engine and its language is closer to the physical machine and can be executed in a simpler way (the semantic gap is narrower.) The most known such engine is the Warren’s Abstract Machine (WAM) for executing Prolog [1] or SECD and Lispkit [15] for executing functional languages but there are many such examples like some implementations of (early) Pascal [19] or less known and more specific languages like Palingol [10].

The design of our chemical engine is also based on this principle. Thus, in our approach HOCL is first transformed into the code of the abstract engine and then this intermediate code is interpreted. It is easy to see that HOCL execution resembles that of (i) functional languages with the exception of commutative and associative properties and (ii) production systems with the exception of hierarchical knowledge base and concurrent execution; yet not equivalent to any of these. To shorten the development cycle we carefully examined the similarities and differences in the computational models and opted to realize the HOCL abstract engine based on the notion of a production system. A production system consists of facts (knowledge) and rules (behaviour) applied to facts. If the facts fulfill conditions assigned to a certain rule, the rule is activated. From many activated rules one is selected by conflict resolution and fired. Firing a rule means executing its action part that updates the facts and leads to firing further rules. This so-called production cycle is repeated over again.

Some of the key requirements of an efficient and simple realization of interpreting HOCL. (i) Efficient pattern matching. Production systems often apply the RETE-algorithm [14] in such a way, a highly efficient pattern matching, the most important cornerstone of the realization is available ready-made. This is the main inspiration of realizing the HOCL interpreter on the foundation of a production system. (ii) Nested solutions (hierarchical knowledge base). Most production systems assume a global knowledge base and do not allow the structured or hierarchical facts. This aspect needs a careful elaboration in the HOCL abstract engine as it is different in production systems. (iii) Concurrency. The concept of locality (molecules react with their “neighbor” molecules) is simulated by a random choice of potential molecules. Yet, the dynamics of the chemical system is quite different from that of a production system and the random conflict resolution needs further refinement. (iv) Level of parallelism. The γ-model is inherently concurrent and this behavior should be modeled with multiple concurrent execution threads yet, their level (granularity) can be different. To keep the granularity at a reasonable level yet, to enable concurrent behavior, we assigned an execution thread to each solution thus, solutions can evolve independently whereas concurrency within a solution (race condition among molecules) is represented by random choice of reacting molecules.

The conceptual representation of various elements of a HOCL program will be introduced by Dijkstra’s Dutch flag [4], as an example. The aim of the Dutch flag problem is to order three colors, white, red and blue in a randomized array so that they are arranged according to the stripes of the Dutch national flag: red, white and blue.
We introduce a simplified, easy-to-read pseudo code for representing and explaining the code of the production system. While they show all the necessary information many irrelevant details are eliminated. A production system represents its knowledge in facts like (1) or (1 2 3). Some facts can have named slots like ((x 1) (y 2) (z 3)). A rule has a left hand side (LHS) pattern that must be matched to enable the rule followed by ⇒ and a right hand side (RHS) action that is triggered if the rule fires.

Molecules. As one may expect, a passive molecule is simply transformed into a fact like $1 \rightarrow (\text{molecule (value 1)})$ or $\text{red} \rightarrow (\text{molecule (color red)})$. A straightforward (and naive) approach would be to represent active molecules as production rules. This way however, makes it very hard to realize the higher order property of the HOCL model where active molecules can be captured, transformed, canceled or added just like any other molecule. Therefore, active molecules are represented by a rule and a fact. Thus, molecule $\text{red}$ is transformed into a fact (rule red) and a rule with pattern shown as (some parts to be refined later):

```
(defrule red
  (rule red)
  ;match <i, red> and <j, white> if i>j
  ⇒
  ;swap <i, red> and <j, white>
```

This rule can fire if fact (rule red) is present in the same solution. All modifications to the active molecules (added, withdrawn, transferred) are performed on this fact that enables the rule. For instance, moving the active red molecule from one solution to another is simply moving the (rule red) fact.

Solutions are two faced entities: they are data if inert and are separate running processes (and thus, unable to be matched) if active. Solutions can hold passive molecules, active molecules, other solutions or pairs and can be nested in arbitrary depth. Unfortunately, production systems usually do not allow nesting the facts hence, there is no straightforward representation. We opted for a Prolog-like representation of compound terms [1] where not actual terms but references to terms that are stored. Therefore, molecules are augmented with identifiers so that references can be put to them. For instance, $(1, \text{blue})$ is represented as two facts (molecule (value 1) (in id$_A$)) and (molecule (color blue) (in id$_B$)) and then the solution itself is a fact (solution id$_A$) (just the idea is shown here, there is more information related to solutions and molecules). This representation seemingly calls for a complicated recursive pattern-matching but it can be solved very efficiently in a flat manner as (following the above example):
\begin{verbatim}
(defrule red
  (rule red)
  (solution x)
  (molecule (value i)(in x))
  (molecule (color red)(in x))
  (solution y)
  (molecule (value j)(in y))
  (molecule (color white)(in y))
  (test i > j)
⇒ ;swap <i, red> and <j, white>
\end{verbatim}

where the matching variables represent the constraint so that molecules belonging to the given solution are selected. Similarly, multiply nested solutions are represented in the same way. Pairs are a special case of solutions: they have exactly two molecules inside and their order is relevant. With minor differences, all the principles introduced for solutions are used for pairs, too.

\textit{Transfer between solutions.} Molecules can be moved between solutions for instance, in \textbf{replace} \(i, \text{red}\), \(j, \text{white}\) \textbf{by} \(i, \text{white}\), \(j, \text{red}\) the two color molecules are exchanged between the two solutions. This is a very simple example but there are cases where multiple molecules are moved, or every molecule (\(\omega\)) moved except some. Furthermore, deleting a molecule can be traced back to the same situation where it is taken from a solution but put nowhere. In order to handle all these cases efficiently and uniformly, we categorized the following cases as types

- \textbf{replace} \(a : (\omega_a), b : (\omega_b)\) \textbf{by} \(a : (), b : (\omega_a, \omega_b)\) – moving all molecules, e.g. from solution tagged \(a\) to solution \(b\)
- \textbf{replace} \(a : (a, b, c, \omega_a), b : (\omega_b)\) \textbf{by} \(a : (\omega_a), b : (a, b, c, \omega_b)\) – moving certain molecules, e.g. \(a, b, c\) form solution \(a\) to solution \(b\)
- \textbf{replace} \(a : (a, b, c, \omega_a), b : (\omega_b)\) \textbf{by} \(a : (a, b, c), b : (\omega_a, \omega_b)\) – moving all but certain molecules, e.g. all molecules from solution \(a\) to \(b\) except \(a, b, c\)

They can be further classified if the source and target solutions are top-level or nested ones or nil. Altogether 15 types of operations belong to this category. In fact, in reactions most of the actions are putting molecules around therefore, this operation must be very simple in the language (and efficient in the implementation). The intermediate language therefore is extended with (\textit{relocate toMove, notToMove, from, to}), a special custom function. Thus, we can finalize the example as

\begin{verbatim}
(defrule red
  (rule red)
  (solution x)
  (molecule (value i)(in x))
  (molecule (color red)(in x))
  (solution y)
  (molecule (value j)(in y))
  (molecule (color white)(in y))
  (test i > j)
⇒
\end{verbatim}
Hence, the active molecule red has been rewritten into rule red of the intermediate language. It is important to mention that – just like the HOCL reaction – firing a rule is an atomic step. That is, in the above example molecules are transferred in a single step and there are no intermediate inconsistent states.

4 Implementation

The principles of an HOCL interpreter based on a production system drafted above have been implemented in jess [16], a Java based production system. Here some additional, implementation related details are explained only.

The intermediate language. HOCL programs are transformed (compiled) into an intermediate language that is based on the jess script language with (i) some restrictions and (ii) an added function. Restriction means a fixed template of molecules and a strict pattern in the head of rules. These principles were shown in Section 3 but in reality molecules contain more information (technical details) than presented before; there is an inherent need to keep them consistent. Therefore, there is a molecule template that defines all the necessary slots and all other passive molecules are derived from that. Restrictions are also present in the head of rules: capturing a molecule has a certain pattern sequence that must be strictly followed. The added function is the relocate introduced earlier. It is important to notice that this is the only one function that is not part of the jess script language and a large area of possible cases are realized by this single instruction. Due to the minimal changes introduced, the intermediate language is very close to the jess script. One familiar with jess or other production systems can easily read, understand and modify the intermediate language. Minimal changes also ensure that the intermediate language is executed as efficiently as the native jess script.

The interpreter. We kept the same principle: introduce as little changes as possible thus, the HOCL interpreter is just a slightly modified jess engine. Furthermore, in case of the interpreter all these changes are transparent to the user. Albeit invisible, some important modifications and extensions must have been added to the basic execution mechanism of jess mainly due to the required support of hierarchical knowledge base. These include activities related to spawning a new RETE-engine (initiate a new solution) or opposely, stop a RETE-engine. In such cases transferring data to a new process and vice versa by maintaining consistency, correctness and avoid synchronization problem is a complex task. To achieve these goals efficiently, Java procedures operate on the internal data structures of jess. Similarly, the realization of relocate is encoded in the interpreter as a custom Java function. Furthermore, the random conflict resolution must have been modified, see the explanation in Section 5. Measurements showed that these additional functionalities in the jess engine do not add significant overhead or cause performance degradation.
User interface, program control and external interfaces. There is a simple user interface developed that facilitates the execution, tracing and debugging of HOCL programs (Figure 1). The main fields show the current reactions, the possible reactions in each solution and the solution structure. The latter is augmented with some graphical aids to see where reactions are possible and what are the inert solutions. Solutions and molecules are clickable: new molecules can be added to solutions at run-time whereas breakpoints can be added or withdrawn on active molecules. Tracing and debugging is supported by various run modes: step-by-step, continuous run with variable speed and breakpoint. For specific applications custom-made user interfaces can be made for instance, an experimental tic-tac-toe game table was implemented (see Section 5). This latter also demonstrates how easily the interpreter can be interfaced with other programs that is a fundamental requirement for coordinating tasks the chemical paradigm is aimed at. In this case the game board is a separate process and steps made by the user are external events imported into the chemical engine whereas steps made by the computer are events that are exported and displayed graphically. In the same way, other sources of events and control can be realized in different scenarios.

Fig. 1. Graphical interface for the HOCL interpreter
5 Experiences learned

The interpreter was tested by a large set of toy examples to verify the correctness of elementary constructs in the language. Also, it was tested by some nontrivial problems listed in [4][8]. Here we present two experiences we learned beyond the simple correctness tests.

An implementation of the foxes and rabbits problem (Lotka-Volterra equations [9]) revealed that dynamicity in the chemical model is a crucial issue. This type of applications should oscillate (the number of foxes and rabbits change periodically) but our initial attempts diverged. The problem was caused by the random conflict resolution of the production system that did not really simulate the random mixture of molecules and must have been replaced by a custom made one. While this sensitivity seemingly affects a very little portion of computational problems, the chemical approach is associated with realizing self-* autonomic systems where evolution and dynamicity of certain populations is of fundamental importance and such aspects must be carefully researched and elaborated.

A player vs. machine tic-tac-toe game revealed the importance of the appropriate transformation of HOCL into the intermediate language. A very simple implementation of this game was encoded in HOCL in a concise way and was executed by the interpreter. Yet, as the size of the field grew, performance problems started to appear and around the table size of 30*30 the game became unplayable due to large response times. The root of the problem was in expressing the HOCL program in the intermediate language. While HOCL allows a very expressive and elegant problem statement, pattern matching works more efficiently on numerous but simple rules. Therefore, a complex HOCL statement must be transformed into the intermediate language so that it is broken into simpler, more specific rules that facilitate pattern matching; tic-tac-toe was successfully hand coded so that it became scalable. While the transformation of HOCL into the intermediate language (compiler construction) can be described easily, taking into consideration such performance issues requires more research work.

6 Conclusions

In this paper we presented the design and implementation principles of an HOCL interpreter for executing programs written in a higher order chemical language. The chemical computing model is an upcoming candidate for realizing autonomic properties in various distributed settings (such as grid and service based environments, see [5] [11] [18]).

The proposed execution of HOCL programs is an interpreter realized as an abstract engine. The engine is based on a production system that lends its state-of-the-art pattern matching mechanism but modified to support the hierarchical notion of knowledge base of the chemical semantics and fulfill other technical challenges. The interpreter supports the entire HOCL language and has a graphical user interface and a basic support for tracing and debugging. The realization of the interpreter also makes possible to interface it to other systems for observation and control.

Test experiments proved the correctness of the interpreter. They also revealed the importance of efficient representation of the HOCL program at the intermediate level.
and that of the dynamic behavior. Both are strongly related to self-evolving properties of autonomic systems and therefore, will play crucial role in real-life applications. These aspects are targets of further research.

Acknowledgements. The research leading to these results has received funding from the European Community’s Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube).

References

Distribution and Self-Adaptation of a Framework for Dynamic Adaptation of Services

Françoise André, Erwan Daubert, Guillaume Gauvrit
IRISA / INRIA
Campus de Beaulieu, 35042 Rennes cedex, France

Abstract—The dynamism and scale of the infrastructure of the Internet of Services bring new needs to build autonomous services. These services have to be able to self-adapt to the variation of the environment. Moreover, these adaptations may span across multiple services and thus have to be coordinated, without breaking their autonomy. To this end we describe in this paper the approach we have chosen for SAFDIS, a framework to make coordinated adaptations of services. In this presentation, a particular emphasis is made on the distribution of the framework and how it helps to coordinate distributed adaptation. Benefits from the self-adaptation of the framework itself are also presented.

Keywords—Dynamic Adaptation; Distributed Services; Distributed Adaptation; Self-Adaptation of Adaptation Framework.

I. INTRODUCTION

The underlying computing infrastructure for the Internet of Services is characterized by its very large scale, heterogeneity and dynamic nature. The system scale is to be measured in terms of number of users, services, computers and geographical wingspan. The heterogeneity comes from its spreading on multiple sites in multiple administrative domains providing very different computers, devices and network connections. Its dynamic nature results from a number of factors such as Internet node volatility (due to computer or network failures, voluntarily connections and disconnections), services evolution (services appearing, disappearing, being modified) and varying demands depending on human being activities.

In a world of services in which more and more personal, business, scientific and industrial activities rely on them, it is essential to guarantee the high availability of services despite failures or changes in the underlying continuously evolving execution environment. Moreover, providing quality of service (QoS) is important considering the number of services related to legal and commercial aspects.

To take into account this dynamism our objective is to design and implement systems that are context aware and able to adapt applications and services at run-time.

The task of making software adaptable is very cumbersome and encompasses different levels:

• At user or business level, processes may need to be reorganized when some services cannot meet their Service Level Agreement (SLA).
• At service composition level, applications may have to change dynamically their configuration in order to take into account new needs from the business level or new constraints from the services and the infrastructure level. At this level, most of the applications are distributed and there is a strong need for coordinated adaptation.
• At infrastructure level, state of resources (networks, processors, memory, etc.) have to be taken into account by service execution engines in order to make a clever use of these resources, such as taking into account available resources and energy consumption. At this level there is a strong requirement for cooperation with the underlying operating system.

Moreover, the adaptations at these different levels need to be coordinated.

So, our main challenge is to build a generic framework for self-adaptation of services and service based applications. The basic steps of an adaptation framework are Monitoring, Analysis, Planning and Execution, following the MAPE model proposed in [1]. We intend to improve this basic framework by refining each step of the MAPE model, in particular by providing elements that cope with the distribution of the application and the underlying infrastructure. The adaptation system can itself be distributed for the purpose of scalability or to better match the heterogeneity of the environment. Moreover, it can be adaptable, allowing to take into account unforeseen situations.

Our system called SAFDIS for Self-Adaptation For Distributed Services fully exploits the advantages of the framework concept [2]. It gives a frame, paradigms and rules to develop and implement adaptation mechanisms, as well as the liberty and the flexibility for the developer to specialize its system according to its specific needs. Using this framework, the task of developing concrete adaptation systems for some applications, services or infrastructures will be facilitated as many of the different elements that may be composed in adaptation systems are exposed, their
interfaces clearly defined, the relationships between them coherently specified. Our SAFDIS framework is build as a set of services, providing functionalities useful to build an adaptation system. Not all functionalities are necessarily needed for each instantiation of an adaptation system. For instance we provide a negotiator service to negotiate the adaptation decisions when SAFDIS is distributed on several nodes; this service is not useful when SAFDIS is build as a unique centralized adaptation system.

The following sections present the advantages resulting from the design of our framework. The next section gives an overview of the SAFDIS framework. Section III presents how the distribution is handled in our framework and its advantages. Then, Section IV presents some advantages of having an adaptation framework that is self-adaptable. Finally, Section V presents some related-works and Section VI concludes this paper.

II. SAFDIS: Self-Adaptation For Distributed Services

Our framework for self-adaptation of distributed services SAFDIS [3] is divided into the four main phases of the MAPE model. Monitoring is the observation function to detect changes that imply adaptation. When a change is detected, the monitoring phase triggers the analysis to analyze it and find an adaptation strategy if it is required. Then this strategy is given to the planning phase to compute a schedule of actions that will implement the strategy. The last step is the execution of the schedule to reconfigure the system (application, services and the environment).

Our framework is able to work at different levels ranging from a single service, a composition of services for one application, to several applications. Each application can be executed on a set of heterogeneous platforms themselves on a distributed and heterogeneous infrastructure (OS and hardware). Therefore in order to adapt a set of applications, it may be necessary to interact with these platforms and some specific (maybe all) execution nodes which represent only a part of the infrastructure. With SAFDIS, it is possible to monitor the different levels according to the implementation of the available probes and adapt them depending on the adaptation actions (Figure 1).

To cope with the distributed environment, our framework can itself be distributed using multiple autonomous and cooperating instances. An instance has to be deployed on each of the service oriented platforms hosting a least an adaptive service using SAFDIS. Our framework is also fully decentralized, meaning that there are no instances with privileges or special purposes. This design avoids single points of failure and makes the framework scalable.

In the following subsections, we present each phase of the MAPE model and some of their characteristics in the context of our framework and our current implementation.

A. Monitoring

The monitoring phase is used to provide an informative and dynamic view of the adaptive entity and its environment to the other phases of SAFDIS. Thus, it is the starting point of every adaptation undertaken. It builds one local view per instance of SAFDIS picking relevant information from the service-oriented platform, the adaptive services, the operating system and the hardware. SAFDIS can probe both passively or actively the system to generate events and update the view. The pieces of information that have to be gathered are specified by the other phases of the framework.

B. Analysis

The analysis phase of a MAPE adaptation system has two goals. The first goal is to identify situations needing an adaptation. It listens to updates of the view of the system pictured by the Monitoring phase. Then it analyzes the changes in the system and decides if an adaptation is needed consecutively to this change. The second goal of the analysis phase is to build an adaptation strategy when a need arises. A strategy defines which elements (parameters, functions...) need to be adapted and how.

Within our SAFDIS implementation, the analysis phase takes decisions with multiple temporal scopes. This gives the ability to either react fast or to take proactive decisions for the long term. This implies the ability to analyze the context with a variable depth of reasoning. Our implementation of the SAFDIS analysis phase also distributes and decentralizes its analysis process to spread the computational load and make the analysis process scalable.

C. Planning

The planning phase seeks the set of actions (the plan) needed to adapt the system according to the strategy chosen.
by the analysis phase. It also schedules the selected actions to ensure a coherent and efficient execution of the adaptation.

Until now the planning phase has received little attention in the context of adaptation and in many cases the planning algorithms used produce simple total orderings of actions. In these cases, the result is not very efficient since the execution may take more time than necessary as the actions are sequentially executed. Moreover in distributed environments, where actions can be asynchronous, some synchronization actions explicitly have to be added to ensure the predefined sequential order.

The planning topic is a well known subject in AI research works and many algorithms already exist in that field to produce efficient schedules. With our SAFDIS framework, the planning phase is able to reuse these algorithms. The resulting plan of actions can have actions that can be executed in parallel.

D. Execution

Once the planning phase has computed the action plan corresponding to the strategy, the execution phase is called to perform the adaptation actions on the service, the application or the environment. These actions are application, platform, OS or hardware specific. That’s why, with SAFDIS, we have introduced two kind of actions. The first kind called concrete actions corresponds to the action implementations which are specific to the adapted element. The second kind called abstract actions constitutes an abstraction of the concrete actions. This allows the planning phase to work with abstract actions without taking into account their specific implementations and doing so to build generic action plans.

III. DISTRIBUTION

The SAFDIS framework is meant to be distributed in the same way the applications it adapts are distributed. When deployed, it is composed of multiple autonomous instances, each one in charge of the adaptation of the services deployed on its platform. However, these autonomous instances cooperate in order to coordinate the adaptations involving distributed elements.

Moreover it is also fully decentralized: there are no instances with privileges or special purposes, therefore there is no single point of failure. When an instance fails, for example from a hardware failure or power issue, the other instances can continue to operate normally, even though the adaptations related to the failed instance will fail.

The absence of an instance with a central role avoids the bottleneck problems that could arise from this role. Also, there is no need for a server dedicated to the management of the adaptation of the services.

For example, let’s consider the adaptive services \( S_a \), \( S_b \) and \( S_c \) respectively executed in the service-oriented platforms \( P_a \), \( P_b \) and \( P_c \) on the execution nodes \( N_a \), \( N_b \) and \( N_c \). The service \( S_a \) uses the services of \( S_b \) and \( S_c \). The three services are using SAFDIS in order to be adaptive, thus there is an instance of SAFDIS on each service-oriented platform: \( I_a \), \( I_b \) and \( I_c \). If \( I_a \) and \( I_b \) are involved in an adaptation on \( S_a \) and \( S_b \) whereas at the same time the node \( N_c \) sees its CPU and memory load decrease, \( I_c \) can without having to ask the other SAFDIS instances make the adaptation decision consisting in allocating more memory to \( S_c \) thus improving the quality of this service. Both adaptations, the one concerning \( S_a \) and \( S_b \) and the one concerning \( S_c \), can be executed in parallel.

When deployed, SAFDIS is a set of distributed instances. Each instance is a set of services which fulfill the four main functions of SAFDIS: monitoring, analysis, planning and execution. The cooperation between the instances is done at the level of the services. For example, analysis services cooperate among themselves but none of them interact with the monitoring and planning services that are outside of its SAFDIS instance. This respects the separation of the adaptation process into four phases.

As there is one instance of SAFDIS on each service oriented platform, each monitoring service is in charge of monitoring its execution node, the platform itself, the services deployed on its platform and the SAFDIS instance it is part of. The other services of SAFDIS always send their requests of information to the local monitoring service. This monitoring service then retrieves the information from another monitoring service if it doesn’t have it. The connection between the various monitoring services are made on demand, using a service registry.

Instead of trying to picture a global view of every elements contributing to the application, which would consume communication resources and not be scalable, SAFDIS pictures multiple local views. This allows to spread the computation load of the analysis on the execution nodes related to the adaptations. But this means that the instances of the analysis component have to take decisions based on partial knowledge of the system. This knowledge alone is not always enough to decide of adaptation strategies. Thus, the analysis instances use negotiation mechanisms in order for them to cooperate in the decision process.

The analysis services cooperate by negotiating strategies. A strategy is initiated by an analysis instance and then negotiated with the other analysis instances that are (or may be) impacted in the adaptation. Those instances can enhance the proposed strategy. They may in turn involve other analysis instances into the negotiation process.

In the previous example involving three adaptive services and SAFDIS instances, if \( I_a \) chooses a strategy and negotiates it with \( I_b \) and \( I_c \), the last two instances analyze in parallel the portion of the strategy requiring their involvement.
Ia takes the final decision to apply the negotiated strategy or to abandon it.

Each analysis service uses three sub-services: a decision maker service in charge of the reasoning and decision making process and a pair of services to handle the negotiation: the negotiation manager and the negotiator services. Each negotiator handles one to one negotiations while the negotiation manager divides the negotiations involving more than two peers into multiple negotiations involving two peers and coordinates them. This design was chosen to enforce a separation of concerns and to ease potential upgrades when SAFDIS is deployed. In our implementation of the framework, the negotiation protocol used by the negotiation manager and negotiator services is the Iterated Contract Net Protocol [4]. However, this is transparent for the decision maker service, so other negotiation protocols could be used.

Once a strategy is negotiated, it is sent to the planning service that is in charge to make a plan of actions to implement it. As said in II, due to the planning algorithms used in SAFDIS, some actions can be scheduled to be executed parallel.

So, in the last phase of the adaptation process which is the effective execution of the adaptation, the distributed aspect of the adaptation process is again emphasized as the actions that can be executed in parallel are executed in parallel, which in a distributed context improves the execution time of the adaptation.

Overall, the distribution of the analysis process and the distribution and the parallelism of the execution phase allow our framework to spread the computation load of the adaptation process and to gain time in this process.

### IV. SELF-ADAPTATION OF THE ADAPTATION SYSTEM

As our SAFDIS framework is developed as a service-oriented application it can itself be adapted as any other application, using its own mechanisms. In this section we present this self-adaptation property and detail its use for the planning phase.

The current implementation of each MAPE phase of SAFDIS can be dynamically replaced by a new implementation. At deployment a first implementation for each phase is chosen by the expert. If the expert think that there is no chance that the context will necessitate his choice to be called into question, SAFDIS will remain the same a long time. But the expert can predict that his initial choice may not be the best one in every circumstances or if some changes appear in the future. In that case he can add policies to the SAFDIS framework he initially used to adapt the application, in order to adapt the framework itself.

Thanks to the use of a service-based adaptation framework design, the need to stop the application and its execution environment when changing the adaptation system is avoided. The adaptation system itself needs not to be completely stopped as a phase may be changed without affecting the others. The expert only has to have foreseen other implementations for the phase subject to potential changes and the policies to decide the change. Of course the new implementation should respect the services specifications, especially be conform with the interfaces defined in our framework to ensure the communication between the MAPE phases. At run-time, the new implementation will be looked for in the services repository, started and then the previous one will be stopped. Interconnections between phases are automatically done through the interfaces, without the help of the expert.

Portions of a phase are also self-adaptable without having to replace the complete phase. This is the case for instance of the negotiation part of the analysis phase.

To illustrate this self-adaptation property, we detail below the way it has been conceived and developed in our current prototype for the Planning phase.

As adaptation is performed at run-time, the time needed to actually perform the adaptation have to be minimized. Therefore, planning is an important phase of the MAPE model. It chooses the actions necessary to properly apply the adaptation strategy, and schedules the actions to ensure the consistency of the adaptation execution.

A simple planning algorithm as used by most adaptation systems uses a static total ordering between all possible actions and leads to a sequential schedule.

For example, if we consider the three possible actions stop service, update service and start service and an order that imposes that whatever the number of services to change all the stops must be done before the updates and all updates before the starts, this adaptation method will maximize the time during which all the services are unavailable, and also consume more time than needed in case some actions may have been processed in parallel.

Moreover, if the adaptation takes place on a distributed and asynchronous environment, explicit synchronization operations should be added to enforce the respect of the schedule between the different parts of the actions that have to be executed on different platforms.

Research works on planning methods such as Artificial Intelligence planning, Motion planning or Control theory, have produced algorithms [5], [6] that overcome these limitations, but without applying them in the context of dynamic adaptation. In SAFDIS, we propose an architecture for the planning phase (called F4Plan for "Framework For Planning adaptation" [7]) that offers the possibility to use, according to the needs, one of these algorithms. Moreover this architecture includes a set of language translators that allow to translate the possible output languages from the analyze phase (languages used to describe the current configuration...
of the application and the target configuration) into the different input languages used by the planning algorithms.

The self adaptation of the planning phase consists in choosing the most suitable planning algorithm according to policies defined by the expert of adaptation. These policies are based on some non-functional constraints defined by the system such as the system overload or the duration which may be acceptable in order to apply the strategy but they also take into account the strategies sent by the analysis phase. For example, if the strategy comes from a reasoning engine that is used to do local adaptations to solve local problems, such as the one we use to make reactive decisions based on event-conditions-actions rules, it is not necessary to use a planning algorithm that searches for a parallel schedule. Indeed, there will probably be relatively few actions to schedule and all of them should be executed on the local node. In that case the simplest planning algorithm is convenient, being able to plan the strategy as quickly as possible, thus minimizing the time spent in the planning phase.

At the opposite, if the strategy comes from a reasoning engine based on utility functions such as the we use to make proactive decisions to do wide adaptations impacting the distributed system, it is interesting to use a planning algorithm able to plan a strategy as efficiently as possible. This planning algorithm should take into account several constraints for example the potential asynchronism between actions and the amount of resources that will be used during the execution phase.

So, the modularity and the service based design of our SAFDIS framework allows a great flexibility in the conception of an adaptive system. We do not neglect of course the task of the adaptation expert who has to conceive the adaptation policies.

V. RELATED WORKS

Today research works on autonomic computing aim mainly to build autonomic components but very few works consider building autonomic services or autonomic service-based applications. Among these works most of them as [8], [9] integrate the adaptation process into the components or services. Each element constitutes an autonomous element of the system and it doesn’t interact with other elements to coordinate more complex adaptations. So, these solutions are not appropriate to manage wider adaptation spanning over multiple services constituting one or more applications. Meanwhile in [8], the authors add some predefined high-level adaptation components to be able to adapt a set of elements constituting an application. But this possibility is restricted to some specific cases for example to resource management or application deployment.

Other works as in [10], [11] separate the behavior of the components or services from the adaptation process. In [10] the generic framework called Dynaco needs to be specialized and is specific for each application, so several instances of the Dynaco framework are needed to adapt multiple applications.

Among these solutions, very few manage distributed systems and are themselves distributed. Based on Dynaco, [12] proposes some coordination patterns to cope with the distribution and decentralization of the adaptation system. However, to our knowledge these solutions are not able to manage heterogeneous applications.

VI. CONCLUSION

Nowadays, software developments should consider the issue of their adaptation when confronted with the dynamism of execution environments. However current solutions for adaptation are most often ad hoc and in consequence are not satisfying as long term solutions.

With our framework, which targets service-based applications, we propose to externalize the adaptation process into a distinct and distributed application. This new application is able to interact with various heterogeneous applications, services and execution platforms to adapt them. Moreover, as a distinct application it is able to adapt itself. In this paper, we have described some characteristics and advantages of our SAFDIS framework to ease the design of adaptation systems for service-based distributed applications. Some relevant parts of our implementation have also been presented.

Our framework provides a set of interfaces useful to build an adaptation system including some optional functionalities, such as a negotiator service which is used to negotiate the adaptation decisions when SAFDIS is distributed on several nodes. It is the role of an expert designer who knows his application and the execution environment to specialize our framework and to choose whether to use those optional functionalities. Moreover, our implementation is built as a Service-Based Application in order to take advantage of the service-oriented approach. For example, the dynamic binding between services eases the replacement of services when updating some part of the adaptation system. We also integrate some self-adaptation capabilities in our adaptation system and use them to select the planning algorithm.

The SAFDIS framework has been experimented to adapt test applications such as video streaming and multi-support video conferences applications. The planning phase has been used for the adaptation of an home-automation application [7], showing significant improvement compared to the initial version of the adaptation system. We are currently working on the design of the adaptation system for a large and very dynamic firemen assistance application.
In order to improve our implementation, we plan to study the use of already defined planning algorithms which are able to distribute the planning process ([13], [14], [15]) and to integrate them. This will help distribute the computation load in the same way it helps the analysis process. We also plan to work on conflicts that may appear in simultaneous adaptation processes. This kind of conflicts may appear because since a distributed and decentralized adaptation system is used, many adaptation processes may be launched and these processes may have to adapt the same element. In that case one of those adaptations may fail or may be inefficient. A third point we plan to study is the use of a knowledge base to share data between adaptation phases and to build a history about the system. This history may be used to improve the quality of the analysis phase by providing feedback on previous adaptations and to ease the resolution of conflicts by providing some information about the state of the running adaptations.

ACKNOWLEDGMENT

The research leading to these results has received funding from the European Community’s Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube).

REFERENCES


Combining SLA Prediction and Cross Layer Adaptation for Preventing SLA Violations

Eric Schmieders¹, Andras Micsik², Marc Oriol³, Khaled Mahbub⁴ and Raman Kazhamiakin⁵

¹ Paluno, University of Duisburg-Essen, Essen, Germany
eric.schmieders@paluno.uni-due.de
² MTA SZTAKI DSD, Budapest, Hungary
micsik@sztaki.hu
³ Universitat Politècnica de Catalunya, Barcelona, Spain
moriol@lsi.upc.edu
⁴ SOI (CITY University), London, United Kingdom
k.mahbub@soi.city.ac.uk
⁵ FBK-Irst, Trento, Italy
raman@fbk.eu

Abstract. Service-based Applications (SBA) are deployed in highly dynamic and distributed settings, where various parts of the constituent components - services and their infrastructure - are controlled by different third parties. In such a loosely coupled environment, adaptation capabilities are needed to manage deviations and unforeseen situations which might lead to negative consequences (e.g. contractual penalties). Current approaches either focus on cross-layer-adaptation or the prevention of SLA violations. In contrast to this, the approach presented in this paper combines both. The paper presents an architecture as a generic framework for the management of arising problems during service execution. Multiple adaptation mechanisms are available to react on adaptation needs, acting on different layers of the SBA (including e.g. the composition layer and the infrastructure layer). The final goal of the cross-layer adaptation capability is to avoid the violation of agreed Service Level (in SLAs) and thus ensure the benefits of SBAs for both customers and providers.

1 Introduction

Service-based Applications (SBAs) provide their functionality in a rather dynamic way using a number of possibly independent services in a loosely coupled way based on the paradigm of Service-oriented Architecture (SOA). The main difference between a traditional modularly built application and an SBA is that the latter cannot control fully its components, but relies on external services provided by different parties. An SBA typically operates on different levels in order to implement its functionality. On the highest level, the performance and quality of offered services as a whole is the main issue, which breaks down into the issue of composing services and executing service compositions. On a lower level, each
service of a composition has to be discovered, selected and executed, which are the basic and classical service provisioning functions. Some services may run at a third party (the case of outsourcing), while some services are executed in a local infrastructure or in a Cloud environment. In latter cases, the platform and proper environment for the execution have to be prepared and maintained for service executions.

As studies reveal (e.g. [12]), distributed applications are extremely fragile, due to the execution of uncontrollable external services. Unexpected changes of third party services or unpredicted network latencies, for example, can cause failures avoiding the timely execution of the SBA. In consequence, failures can cause violations of the agreed Service Level Agreements (SLAs), what might lead to further negative consequences, e.g. contractual penalties. In order to prevent SLA violations, adaptation capabilities are needed to react to failures and unforeseen situations during the execution before the SLAs are violated.

Current approaches either focus on performing cross-layer-adaptation in a strictly reactive way, hence not avoiding SLA violations and the implicit contractual penalties. Or they aim to prevent SLA violations, by focusing on the Service Composition and Coordination Layer (also known as SCC, cf. definitions in [9]). Those approaches don’t exploit every opportunity to prevent an SLA violation and thus face situations where a prevention is not possible, e.g. when internal services are to be invoked which cannot be simply replaced. To address this gap, the paper describes a framework which prevents SLA violations of SBAs, by applying cross-layer adaptation via the cooperation of monitoring and adaptation facilities.

Real-life use-cases demonstrating the need for our approach can be found for example among emergency services where certain reactions have to be made in given time limits, or among financial services where transactions going late may cause money loss (e.g. stock exchange).

The rest of the paper contains an overview of related work on cross-layer adaptation (Section 2), the detailed description of our approach (Section 3), and the conclusion of our work (Section 4).

2 Related Work

In [3] Gjerven et al. propose a framework to support cross-layer self-adaption in SBAs. They present a technologically independent middleware named QUA to support coordinated cross-layer adaptations by integrating interface and application layer adaptation mechanisms. Similarly, Popescu et al. also present a cross-layer adaptation framework in [10] with internal and external services which uses adaptation templates to define the behaviour of adaptation processes. Focusing on particular layers, Vidackovic et al. [11] present a cross-layer monitoring and adaptation framework, where they aim at the relationships of the Business Model with respect to the lower layers, and propose a cross-layer adaptation strategy based on a top-down approach. Although these approaches
handle cross-layering adaptations, they do not avoid the violation of the SLA during the execution of the SBA.

Regarding approaches limited to the service composition layer, V. Cardellini and S. Iannucci [1] present a framework named MOSES with two concrete adaptation strategies, namely service selection and coordination pattern selection. When a service fails, the adaptation is performed only for future invocations of the SBA. For that reason, it does not avoid the violation of SLAs during the SBA’s execution. This problem is addressed by Leitner et al. in [4]. Leitner et al. present a framework, named PREvent, in order to predict end-to-end performance violations of SBAs during their execution. In contrast to our work the presented framework needs hundreds of SBA executions to provide precise violation predictions. This is due to the exploitation of machine learning techniques, which in general requires a huge amount of training data for adequate results.

In summary, the above mentioned solutions fall into two disjunct categories: one category facilitates cross layer adaptation, while the other avoids SLA violations of executed SBA. Our contribution combines both aspects: instrumenting cross layer adaptation in order to avoid SLA violations.

3 Approach

The key idea behind the integrated solution represented in this approach is based on the use of assumptions on the SBA’s context. The context is not under control of the SBA provider, as it includes e.g. third party services. The important aspect of the assumptions in our approach is, that the assumptions are used to relate the continuously monitored data to the SBA’s requirements. In particular, we exploit monitors to check whether the assumptions are still satisfied. In case an assumption is violated, we check immediately whether the end-to-end requirement is violated as well. If the check reveals a violation of at least one requirement, the service composition must be adapted in order to compensate the delay occurred in the preceding execution of the SBA instance. To achieve the adaptation an appropriate strategy is generated by a multi-agent community. Finally, the adaptation strategy must be executed.

The architecture of the solution is represented in Fig. 1. The workflow is executed by a Process Engine that communicates with the services through an Enterprise Service Bus (ESB). The ESB routes the service invocations and serves as an aggregation point for different types of events from different sources (e.g. from service calls). Furthermore, the ESB enables dynamic rerouting of requests to different services. The information about available services and their current properties is stored in the service repository. In the following the components and the relation to the different layers are introduced.

Service Monitoring The key monitoring component is SALMon (cf. [8]). SALMon is used to monitor assumptions regarding the different properties of individual services, specially their non-functional properties (e.g. availability, response time). SALMon is able to (1) monitor the QoS of services in a SBA (Service Composition and Coordination Layer) and (2) check if the retrieved
QoS matches with expected values. Furthermore, SALMon is able to monitor process KPIs like the process’ end-to-end response time, hence also addressing the Business Process Management (BPM) layer. In order to be easily integrable with other technologies, SALMon has been implemented as an SBA by itself.

**Adaptation-based Detection** If a violation of the stated conditions occurs, SALMon notifies the Specification and Assumption Based Detection (SPADE) of adaptation needs, whose basics are introduced in [6]. The notification contains the violated assumptions and the violating value.

The SPADE component is used to evaluate the impact of the violated assumption on the corresponding application requirement. SPADE performs a runtime verification of the process model against the requirements. SPADE checks, whether the workflow specification \( S \), the monitored data \( M \) and the assumptions in \( A' \) satisfy the given requirements \( R \), that is: \( S, M, A' \models R \). The set \( A' \) comprises assumptions related to services which are not invoked at the point in time when the check is performed.

If \( R \) is satisfied, then the workflow execution is continued. If \( R \) is not satisfied, the SBA or the service infrastructure (SI) must be adapted in order to compensate the delay. In the latter case the adaptation strategy engine is invoked.

**Adaptation Strategy Engine** The adaptation strategy engine is manifested as a multi-agent platform where agents implement intelligent behaviors and negotiations with each other in order to collect available information and make decisions on adaptation strategies. Each service composition instance is represented by a Process Agent responsible for the proper and timely execution of this instance. Internal and external services are represented by Service...
Agents, comprising information of the service properties (like response time) and supported adaptation capabilities.

A Process Agent is instantiated together with the instantiation of each process. SALMon triggers the responsible Process Agent whenever a deviation from the agreed SLA is predicted. In this case, the Process Agent issues requests for an adaptation solution to Service Agents, which, in turn, respond with their offered adaptation solution. The Process Agent chooses one from the offered adaptation strategies. By running the SPADE check the Process Agent assures the adherence to the SBA requirements for the chosen adaptation strategy. This strategy can comprise several invocations of one or more Adaptation Capabilities affecting the SCC and SI layer. Once the Process Agent has chosen an adaptation strategy the Process Agent forwards this strategy to the Adaptation Enactment Engine for execution. This engine invokes the Adaptation Capabilities according to the strategy.

Adaptation Enactment Engine and Adaptation Capabilities

The adaptation strategy is executed by the Adaptation Enactment Engine. The adaptation capabilities invoked during the execution are described in the following.

Service replacement

When performing the service replacement, the service repository is involved (cf. [5]). In particular, the request for a service replacement is transformed into the query of the runtime service discovery tool. The identification of alternative services is based on various characteristics of the published services such as structural, behavioral and quality characteristics that services should have in order to be acceptable replacements for a constituent service. The actual service replacement is performed via the ESB, which reroutes the service invocations to the replacing service. The service replacement is located at the SCC layer.

SLA re-negotiation

The SLA negotiation broker performs the SLA negotiation for each candidate service identified by the service discovery tool, located at the SI-layer (cf. [5]). The desired quality level is negotiated with the selected candidate service. The QoS characteristics of each candidate service are negotiated in order to achieve the best possible SLA for the service that is within the boundary constraints (e.g. costs and response time) of the service provider and the consumer.

SI Adaptation

In case of internal services we propose to influence the infrastructure of the internal services. For example, in order to catch up lost time caused by preceding service failures, the execution speed of an internal service could be increased. This is possible, as the process owner has access to the infrastructure of internal services, and can make executions run faster. For this purpose several approaches are available (eg. [7]). The SI adaptation is located at the SI-layer.

4 Conclusions and Future work

In the paper, a solution was presented to avoid SLA violations in the complex settings of Service-based applications, thus addressing a gap in the current lit-
erature which either prevents SLA violations or applies cross-layer-adaptation. The novelty of the approach is the exploitation of all SBA layers (BPM, SCC and SI) for the prevention of SLA violations. The identification of adaptation needs is based on SLA prediction, which uses assumptions on the characteristics of the running execution context. Multiple adaptation mechanisms are available to react on the adaptation need, acting on different layers of the SBA. The adaptation strategy chooses the right adaptation mechanism, coordinated by a multi-agent community. In the future, we plan to further generalize the mechanism for handling the adaptation management cycle of detection, selection and enactment, and to incorporate more components, event types and adaptation capabilities.

Acknowledgments: We cordially thank Andreas Metzger for helpful comments on the paper draft. The research leading to these results has received funding from the European Community’s 7th Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube).

References

ECMAF: An Event-Based Cross-Layer Service Monitoring and Adaptation Framework

Chrysostomos Zeginis, Konstantina Konsolaki, Kyriakos Kritikos, and Dimitris Plexousakis

Department of Computer Science, University of Crete, Greece and Information Systems Laboratory, ICS-FORTH, Greece {zegchris, konsolak, kritikos, dp}@ics.forth.gr

Abstract. Although several techniques have been proposed towards monitoring and adaptation of Service-Based Applications (SBAs), few of them deal with cross-layer issues. This paper proposes a framework, able to monitor and adapt SBAs across all functional layers. This is achieved by using techniques, such as event monitoring and logging, event-pattern detection, and mapping between event patterns and appropriate adaptation strategies. In addition, a taxonomy of adaptation-related events and a meta-model describing the dependencies among the SBA layers are introduced in order to “capture” the cross-layer dimension of the framework. Finally, a specific case study is used to illustrate its functionality.

Keywords: event, monitoring, adaptation, cross-layer, service, pattern, Event-Calculus, non-functional

1 Introduction

Web services are an emerging technology attracting a lot of attention from both academia and industry in recent years. Thus, more and more businesses adopt them to facilitate and automate their business processes. However, once services and business processes become operational, several emerging issues must be considered throughout the life-cycle of a Service-Based Application (SBA), such as the ones concerning service monitoring and adaptation. These two processes are tightly connected and result in improving and customizing non-performing services, so as to adapt to the context changes and satisfy new requirements.

As far as monitoring is concerned, SBAs need to be managed and monitored, so that stakeholders have a clear view of how services perform within their operational environment, take management decisions, and perform control actions to modify and adjust their behavior. In [5], Web service monitoring is defined as the process of collecting and reporting relevant information about WS service execution and evolution. Many monitoring approaches have been proposed, most of them focusing on measuring QoS metrics, gathered on the basis of SLAs, given that an SLA dictates what are the service obligations in terms of performance.
The measured metrics are then compared with the corresponding SLA bounds to detect possible violations.

Furthermore, the dynamic and ever-changing nature of the business and physical environment requires Web services to be highly reactive and adaptive to the changes and variations they are subjected to. They should be equipped with mechanisms to ensure that they can adapt to meet changing requirements. Such changes are identified, detected, and foreseen in the running SBA during the monitoring process. In [10] Web service adaptation is defined as a process of modifying SBAs so as to satisfy new requirements dictated by environmental changes on the basis of adaptation strategies designed by the system integrator.

However, it is critical that monitoring and adaptation occur across all the SBA functional layers, as they are adopted by many projects, such as the S-cube (http://www.s-cube-network.eu) and SLA@SOI (http://sla-at-soi.eu); namely the Business Process Management layer (BPM), the Service Composition and Coordination layer (SCC) and the Service Infrastructure layer (SI). These layers are closely related and many dependencies exist among them. A thorough review of current monitoring and adaptation techniques [25] reveals that these techniques are very fragmented. Although current approaches cover a wide spectrum of monitoring and adaptation [19, 2, 3], few of them deal with cross-layer issues. The latter ones either do not provide a concrete solution to the problem [12], or do not take into consideration all SBA layers [9, 22], or do not elaborate on monitoring issues [26].

In this paper, a cross-layer monitoring and adaptation framework is outlined that is based on monitored events. In addition, a layer-based taxonomy of adaptation-related events and a meta-model describing the dependencies among components of a cross-layer system are introduced to pinpoint the need for such a type of framework. The main strengths of this approach are the ability to handle both functional and non-functional service properties and the use of a reasoning tool deriving appropriate adaptation strategies by exploiting a set of rules. Furthermore, it has also proactive adaptation capabilities using pattern detection techniques and can be easily distributed into many computer nodes. In Section 2 a comparison table clearly states the advantages of this work.

The rest of the paper is structured as follows. Section 2 summarizes the related work. Section 3 introduces the proposed taxonomy of adaptation-related events. Section 4 presents the dependency model. Section 5 presents the proposed framework, while Section 6 exemplifies how this framework supports the case study. Finally, some concluding remarks and future work directions are presented in Section 7.

2 Related Work

The need for monitoring different functional and non-functional requirements, as well as taking adaptation actions is widely recognized by industry and academia, as a means of improving SBAs. There are several works that propose monitoring and adaptation architectures but most of them only focus on the SCC layer.
Baresi et al. [4] present an approach for self-healing of BPEL processes. This approach is based on Dynamo [3] monitoring framework along with an AOP extension of ActiveBPEL and a monitoring and recovery subsystem using Drools ECA rules. A composition designer provides assertions for invoking, receiving or picking activities in the business process. These assertions can be specified using two domain specific languages (WSCoL and WSReL). The problem of selecting alternative services and dealing with possible interface mismatches when forwarding a request to an alternative endpoint recovery is not explicitly addressed. Additionally, the recovery rules cannot be changed dynamically, as they need to be compiled off line.

The VieDAME environment [19] extends the ActiveBPEL engine to enable BPEL process monitoring and partner service substitution based on various strategies. The services are selected according to defined selectors. VieDAME requires service registration in a repository and marking services to be monitored and eventually substituted as replaceable. It uses an engine adapter to extend the engine’s functionality, but does not explicitly address fault handling.

Barbon et al. [2] present an event-based monitoring approach, developed within the Astro project, which also extends the ActiveBPEL engine and defines RTML, an executable monitoring language to specify SBA properties. Events are combined by exploiting past-time temporal logics and statistical functions. Monitors run in parallel with the BPEL process as independent software modules verifying the guarantee terms by intercepting the input or output messages received or sent by the process. This work does not allow for dynamic (re-)configuration of the monitoring system in terms of rules and meta-level parameters.

In [24, 17] the authors present an approach towards extending WS-Agreement [1]. This approach supports monitoring of functional and non-functional properties. EC-Assertion is introduced to specify service guarantees in terms of different types of events, which is defined by a separate XML schema. It is based on Event Calculus (EC) [23]. By proceeding in parallel with the business process execution, it leads to less impact on performance but also to a smaller degree of responsiveness in discovering erroneous situations.

Farrel et al. [8] present an SLA-based monitoring approach exploiting EC as the underlying formalism. This approach addresses the utility computing domain, where the cornerstone aspect is to provide resources with certain quality characteristics. Contracts are defined based on contract patterns, which are then axiomatized using EC. Then, the effects of critical events on the contract state/evolution are defined. The respective framework is based on a particular architecture and comprises an analyzer managing the contract analysis and reporting, and a visualizer representing the SLA monitoring results.

A platform for developing, deploying, and executing SBAs is proposed in [6], incorporating tools and facilities for checking, monitoring, and enforcing service requirements expressed in WS-Policy notations. The Colombo platform comes with a module that manages policy assertions. Apart from evaluating the assertions attached to particular service-related entities at both design and
The framework provides the means for policy enforcement, e.g., it may approve a message’s delivery or reject it, or defer further processing.

Despite the previous layer-specific approaches, some approaches towards cross-layer service monitoring and adaptation have been proposed. Gjørven et al. [9] propose a coarse-grained approach, which exploits mechanisms across two layers (Service Interface and Application corresponding to ours SCC and SI layers) in a coordinated fashion. Kazhamiakin et al. [12] define appropriate mechanisms and techniques to address the adaptation requirements and constitute an integrated cross-layer framework. Both approaches tackle the problem in an inflexible manner, as the adaptation logic is predefined and static. Popescu et al. [22] provide support for dynamic cross-layer adaptation using adaptation templates, composed either directly, through invocations of WSDL operations or indirectly, through events. This approach, though, does not consider the Infrastructure layer. Finally, Zengin et al. [26] introduce an adaptation manager (CLAM) that can deal with cross- and multi-layer adaptation problems. This approach ranks a set of adaptation paths, after constructing and adapting a tree starting from an initial adaptation trigger at any of the three SBA layers.

A summary of all the aforementioned approaches is presented in Table 1. The approaches are compared according to their (cross-layer) monitoring and adaptation capabilities, dynamicity, intrusiveness and timeliness. These properties acquire a different meaning towards monitoring and adaptation. The dynamicity of a monitoring framework concerns the ability of the framework to change monitoring properties during the execution of the process whereas the dynamicity of an adaptation framework allows additions and deletions of adaptation rules. Moreover, approaches which perform monitoring by weaving code that implements the required checks inside the code of the system that is being monitored are concerned as intrusive approaches. Regarding adaptation approaches, a framework is intrusive whether the applied adaptation actions change the process. We assume that intrusiveness is not desirable for monitoring unlike adaptation. The timeliness of monitoring system presents the ability of the framework to signal violations of the monitoring properties the time they occur and not after the termination of the instance. On the other hand, timeliness of an adaptation framework is about the proactive or the reactive execution of the adaptation actions. Furthermore the kind and the scope of the monitored information is provided. The former one refer to the functional and non-functional properties of a SBA while the latter one to instance or class application of the approach. Finally, the last two properties refer to the availability of a dependency meta-model describing the dependencies among the SBA layers and a taxonomy of monitored events.

As shown in Table 1 few of current monitoring and adaptation approaches deal with cross-layer issues [9, 22, 26]. These works focus mainly on adaptation process and lack some important properties that ECMAF efficiently address, such as proactive adaptation, functional and non-functional aspects consideration and wide scope, enriched with a dependency meta-model and an event taxonomy.
3 Taxonomy of Adaptation-Related Monitored Events

Many of the proposed monitoring approaches can detect different event types. These events deliver information about the SBA evolution and context change. They are used to indicate whether the SBA execution evolves normally and whether there are some deviations or even violations of the desired or expected functionality. Most events are recurring during service executions and usually with the same order. Thus, it is desirable to introduce a taxonomy of monitored events to enable the mapping between these events and the suitable adaptation strategies as well as the event derivation applied in the proposed framework.

The taxonomies of common monitored events proposed are either generic or domain-specific (e.g. real-time SBAs). [22] introduces an event taxonomy for three possible application layers: organization, behavior and service, to semi-automate the discovery and selection of adaptation templates needed to fulfill complex adaptation requirements. [14] categorizes monitored events into Interface-level mismatches, i.e., services with similar functionality but through different WSDL interfaces, and Protocol-level mismatches, i.e., mismatches concerning the order or number of supplied and required messages.

The proposed high-level event taxonomy is based on two different criteria: a) the affected SBA layer and b) the service aspect concerned (functional, non-functional). The affected layer concerns the three functional layers analyzed in Section 1, while the service aspect concerns the service functional and non-functional characteristics [20]. The former ones detail the operational aspects that define the overall service behavior, such as the way and time it is invoked, while the latter concern quality attributes (e.g., response time and throughput). Its main advantage upon the other taxonomies is that it fits perfectly to the
adopted SBA layers as well as the consideration of both functional and non-functional service aspects.

Fig. 1 illustrates the proposed taxonomy of adaptation-related events for the three functional SBA layers. Indicatively, at the BPM layer there are classified mismatches regarding the business process, such as KPI violations or monitored events stemming from modifications at this layer (e.g., business goal or process model modifications). At the SCC layer, monitored events focus on mismatches about service execution and QoS violations, such as I/O failures and SLA violations. Finally, at the SI layer, events mainly concern device failures affecting the overall SBA, such as limited resources or network failures.

4 Dependency Meta-Model

When faults occur, it is imperative to be able to detect the root of the problem by following a process, usually called root-cause analysis. Such a process benefits from the use of a specific model called dependency model. This model should describe both the functional and non-functional dependencies between the system components across all possible layers to enable the detection of the component causing the fault through a top-down traversal of the respective dependencies, starting from the component where the problem is detected. Dependency models should also allow a bottom-up traversal of the respective dependencies. Such a traversal is essential so as to enable the derivation of events at higher-layers with respect to the events occurring at lower levels. Moreover, dependency models should be able to describe both static as well as dynamic component dependencies. The former are usually known and established at design time, while the
latter are detected and created at runtime. Dependencies may also change during the lifetime of a SBA or system. For example, the memory requirements for installing a service may be different from those related to the service execution.

Dependency models must also conform to a particular structure and must be described in a formal way so as to enable reasoning on them. To this end, a novel dependency meta-model is proposed in this paper, which has been specified through the OWL ontology and is depicted in Fig. 2. This meta-model has been carefully designed to capture all the relevant aspects of component dependencies.

The central concept in this meta-model is \textit{Dependency}, which is characterized by the following four main properties \cite{13}: a) \textit{strength}: how strong (optional or mandatory) is the dependency between two or more components, b) \textit{formalization}: what is the dependency’s degree of formalization which directly relates to the automation degree with respect to the dependency’s capturing, c) \textit{criticality}: how critical is to capture this dependency, d) and \textit{period}: what is the time period for which this dependency holds. Dependencies can be either \textit{Functional} or \textit{NonFunctional}. Functional dependencies exist between functional components, while non-functional dependencies typically exist between non-functional components. Both the \textit{FunctionalComponent} and \textit{NonFunctionalComponent} concept are sub-concepts of \textit{Component}. Functional components are characterized by two main properties: a) \textit{type}: indicating if the component is a hardware, software, or logical entity (e.g. service, activity), and b) \textit{activity}: if the component can be directly queried or instrumented or requires the existence of an intermediary for obtaining the component’s information. On the other hand, a non-functional component can be either a \textit{QoSAttribute} or \textit{QoSMetric}.

Non-functional dependencies can be \textit{Qualitative} or \textit{Quantitative}. Both dependency types can be expressed with the OWL-Q \cite{16} semantic, non-functional based service description language, which has been slightly extended to enable the description of process and infrastructural quality attributes and metrics. Fig. 2 shows with black color which novel non-functional concepts have been introduced and with grey color which were the original OWL-Q concepts. Qualitative dependencies between two non-functional components describe particular, general non-functional relationships which can be either only qualitatively assessed, or also quantitatively assessed through e.g. instrumentation but their quantitative dependency model holds only for particular systems and services. Such a type of dependencies is characterized by two main properties: a) \textit{valueDirection}: indicating that the value of the first non-functional component changes in the same or opposite way with respect to the value of the second one; b) \textit{valueImpact}: describes the impact that the first non-functional component’s value has on the second non-functional component’s value.

The OWL-Q extensions introduced allow the description of quantitative dependencies across the same or different layers. In particular, three different metric types have been introduced: \textit{ProcessMetrics}, \textit{ServiceMetrics}, and \textit{InfrMetrics}, where each metric type not only corresponds to one of the respective layers considered but also measures a particular \textit{QoSAttribute} and applies to particular functional components at this layer. Similarly, a \textit{QoSAttribute} can be either a
ProcessAttribute, a ServiceAttribute, or an InfrAttribute. Each metric can then be measured through the application of a Function on other metrics at the same or lower layers. For example, a ProcessMetric can be measured through process, service, and infrastructural metrics and concerns a particular process component, such as the process itself or one of its activities. It must be noted that OWL-Q is already capable of describing the way both single and composite metrics can be computed from measurement directives and other metrics, respectively.

Fig. 2. The dependency meta-model

Therefore, OWL-Q allows for the description of both quantitative and qualitative cross-layer metric models. Quantitative models can be exploited not only for monitoring but also for conformance checking as they do not only allow the calculation of the values of the metrics at the higher levels from the values of metrics at the same or lower levels, but also the comparison of the computed values against the stated requirements. On the other hand, qualitative models allow inspecting the correctness of the monitored values, as e.g. particular qualitative dependencies may not be respected by the monitored values of specific quality components, produced by particular error-prone sources of information.

5 Monitoring and Adaptation Framework

Fig. 3 presents the architecture of the proposed cross-layer monitoring and adaptation framework. This framework comprises a Monitoring Engine able to collect the monitored events during the service execution, an Adaptation Engine able
to perform adaptation actions, and an Execution Engine. The first two engines communicate with each other via events through a publish/subscribe mechanism.

**Monitoring Engine.** The Monitoring Engine comprises a Monitor Manager and a number of individual Monitoring Components. Each of the latter components is assigned to detect events at a specific SBA layer and immediately deliver them to the Monitor Manager. The Monitor Manager, in turn, continuously delivers information about the service execution produced by the Execution Engine while collecting events from the Monitoring Components. The Monitor Manager communicates with the Translator via a publish/subscribe mechanism. A needed monitored event is delivered to the Translator as soon as it is detected. It is imperative to send the events in the order that they are received so as to have an as reliable as possible pattern matching mechanism. As there are many Monitoring Components delivering events, specific techniques are required to ensure this, such as event timing [18] and clock synchronization [15].

---

**Adaptation Engine.** The Adaptation Engine comprises a number of components supported by suitable repositories:

- The Translator, also incorporating a subscribe mechanism, receives the events sent by the Monitor Manager and translates them into a suitable format required by the Event Pattern Detector and Reasoner components.
- It is very common that failures at the SI layer lead to other failures and violations both at the same and higher layers, forming a chain of monitored events. So, it is desirable to detect those event patterns causing service fail-
ures by exploiting specific mechanisms (e.g. pattern matching ones [11]). These patterns are detected by the Event Pattern Detector. Our approach relies on using such a technique to detect dynamically patterns of monitored events to prevent other chains of events from occurring. As such, it enables the proactive service adaptation by mapping the detected patterns to suitable adaptation strategies. For example, having received two monitored events the Event Pattern Detector detects a specific pattern composed of one more event. This pattern is mapped to an adaptation strategy, and then, this mapping is translated into a rule that is passed to the Reasoner. The adaptation strategy which is finally derived from the Reasoner, into an appropriate format, prevents the third event from occurring. A Database supplies the needed information for the mapping and the translation.

- The Reasoner receives events from the Translator and rules from the Event Pattern Detector, and derives an adaptation strategy to be performed by the Adaptation Manager. The rules provide the required data to perform the derivation, supported by a Knowledge Base (KB), containing the appropriate information to take the right decision.

- The Adaptation Manager executes the adaptation strategy exported by the Reasoner with the aid of two components. The Infrastructure Manager is able to treat malfunctions regarding the SI layer, which is the main source of many service failures, and the Model Repository supplies the appropriate information, such as service descriptions and requirements, layer dependencies, and metric and SLA models, so as to fulfil the supported adaptation strategies. The Execution Engine is called to support the adaptation process, especially for strategies regarding the BPM and SCC layers.

The main benefits of the monitoring and adaptation framework are as follows:

- **Distributed workload.** As there are layer-specific Monitoring Components that pass monitored events to the Monitor Manager, monitoring is distributed among the available monitoring mechanisms. In addition, there can be many computer nodes with a separate Monitoring and Infrastructure Manager component, that can handle a portion of the whole monitoring and adaptation workload, as depicted in Fig. 4.

- **Extensibility.** As services evolve, new monitoring and adaptation techniques are required in order to cope with continuous context changes and other unpredictable malfunctions. This framework can integrate such techniques with the existing ones, while preserving its functionality and integrity.

- **Cross-layer capability.** The framework is able to support all three SBA functional layers. It incorporates mechanisms to detect events across all layers and derive additional events using pattern matching techniques.

- **Pro-active adaptation.** The use of pattern matching techniques, as well as the mapping between patterns and adaptation strategies allows for proactively adapting the SBA.

Some preliminary implementation solutions have already been investigated. As far as the monitoring is concerned we plan to use the Astro project [2],
which has already been discussed in Section 2. In addition, we have decided to use the ESPER language (http://esper.codehaus.org) for the specification of the events and the pattern detection process. ESPER provide efficient event processing, comprehensive pattern detection and publish/subscribe capabilities. Finally, the reasoning process can be efficiently accomplished by the powerful EC-Jess Reasoner [21], which translates Event Calculus axioms into Jess rules and then encodes the domain description as a Jess program. Its main benefits are that it handles Event Calculus rules and produces models as output. The joining of these mechanisms as well as other gaps filling within this framework is in our future plans.

6 Traffic Management Case Study

This case study describes a traffic management system designed to manage normal traffic situations as well as emergency cases [7]. Such emergency case handling includes several different actions, such as directing the rescue forces to the accident location and managing traffic deviations. Fig. 6 and Fig. 5 respectively illustrate these two cases. Each figure depicts the three functional layers. In both cases, workflow tasks are executed either manually or by mapping them on (Web) services. Each service is then mapped to the appropriate infrastructure.

The actors involved are traffic managers, i.e., individuals accountable for entities controlling the traffic management system, generic rescue forces (e.g., police and ambulances), and citizens, such as motorists and pedestrians.

Fig. 6 illustrates normal traffic conditions, where the system tries to optimize some parameters such as total noise, overall throughput, and air pollution. In particular, the system shall consider different needs, such as the ones of pedestrians and motorists, and other factors like heavy traffic, public events, school and working hours, holidays or public regulations which may alter traffic demand and needs during conditions that do not involve emergencies. The system interrupts the Normal Traffic Situation process, when an accident happens, and jumps to the Critical Traffic Situation subprocess.
Fig. 5 depicts a critical traffic situation, in which a serious car accident occurs at a central road. In particular, the involved citizens inform the traffic manager that must control the overall traffic situation (control traffic devices, inform citizens) and assess the incident so as to inform the appropriate rescue forces about the accident and direct them to the specific location. Moreover, the traffic manager monitors the environment variables, such as air pollution and noise. Different adaptation actions must be taken by the traffic manager as well as by the rescue forces, such as:

- Traffic management device reconfiguration (e.g., traffic lights) by the traffic manager, in order to reduce stop-and-go traffic. This should also help to keep air pollution low, even if it is not critical during emergency situations.
- Accident reporting to citizens via their devices (e.g., GPS, mobile phones) by the traffic manager to avoid traffic congestion at the accident location.
- Traffic closing/limiting to or from the involved location by the rescue forces.
- Traffic deviation by the rescue forces through alternative places not intended for heavy traffic.

After a complete emergency handling, there is a gradual return back to the normal situation. The rescue forces inform the traffic manager, who updates the system and informs the citizens through their devices.

As already discussed, there are various dependencies among the three SBA layers. The occurrence of a failure at one layer may result in a failure at other layers. This work aims at locating the failure event and taking adaptation actions in order to prevent its spread at the others layers, as soon as possible.

Fig. 6 presents an illustrative example. Suppose that a KPI exists dictating that the maximum duration of the process should be less than 10 seconds. Further, suppose that an SLA exists for the assessment service \( AS \) dictating that its maximum execution time must be less than 6 seconds. Thus, as it can be seen, a violation of the respective SLA constraint may cause a violation to the KPI,
by considering that the previous process activities do not run longer than 3 seconds. It must also be indicated that AS’s execution time is inverse proportional to the main memory size and the CPU percentage allocated for its execution. Moreover, there is a low limit for the main memory allocated, after which the SLA violation will be unavoidable as the service behavior will be unpredictable and even if it does not fail it will certainly take a long time to execute. In fact, after 2 seconds from the AS’s execution, the main memory allocated to it has indeed surpassed the low level of 50 MB.

A Monitoring Component, running at the server where AS is executed, detects that the available main memory is not sufficient (SI layer) for AS. At the same time, another Monitoring Component detects that there is an I/O failure at the SCC layer as AS has produced a wrong output. Both events are first sent to the Translator, which transforms them to the appropriate format and sends them to the Reasoner. Based on the two events received, a specific rule is fired which derives that the best strategy is to execute another instance of the AS service at a more powerful server and with a better memory and CPU allocation. The suitability of the strategy lies on the fact that by executing a “better” service instance and with better allocation for the hardware resources, the probability that the SLA is not violated becomes very high (as we do not know if another failure may occur in the near future regarding the new instance) and in this way also the KPI violation may be avoided. Such a rule has been derived by the Event Pattern Detector based on the previous history log and has been already inserted into the Reasoner. The derived strategy is sent to the Adaptation Manager which executes it with the assistance of the Infrastructure Manager and the Execution Engine.

As can be understood, ECMAF handles perfectly such a cross-layer scenario. The adaptation actions performed are the appropriate ones, based on the event history and the current context. Moreover, the dependencies among the layers are clearly discerned.
7 Conclusions and Future Work

To sum up, this paper describes a framework that is able to detect monitored events across the three functional layers of a SBA and derive suitable adaptation strategies through a reasoning mechanism. The communication between the monitoring and adaptation engines is achieved by a publish/subscribe mechanism. In addition to the framework’s description, a taxonomy of adaptation-related events and a dependency meta-model between the system components across all functional SBA layers are provided, in order to pinpoint the need for such a cross-layer approach. The main benefits of this approach are that it can handle both functional and non-functional service aspects, as well as it can proactively adapt the SBA across all the functional layers.

The following future directions are planned. First, developing such a distributed cross-layer monitoring and adaptation framework, using existing technologies and mechanisms and extending them if necessary. Second, extending the current taxonomy of adaptation-related events, exploiting additional aspects. Third, extending the proposed dependency meta-model. Finally, experimentally evaluating and optimizing the framework.

8 Acknowledgements

The research leading to these results has received funding from the European Community’s Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube).

References

A QoS Assurance Framework for Distributed Infrastructures

André Lage Freitas, Nikos Parlavantzas, Jean-Louis Pazat
Université Européenne de Bretagne
INSA, INRIA, IRISA, UMR 6074
F-35708 Rennes, France

ABSTRACT

Enforcing SLAs (Service Level Agreements) for services deployed on large-scale distributed infrastructures, such as grids and clouds, is complex owing to fluctuating customer demand and unpredictable resource availability. Current systems either address specific application domains or fail to provide a complete QoS assurance solution. This work proposes a generic framework to assist service providers in enforcing quality properties in distributed environments. The framework provides a rich set of QoS management functions, including negotiation, translation, and resource provisioning. Importantly, the framework supports dynamic adaptation; that is, it automatically modifies service behavior and resource usage in order to maintain agreed service levels while satisfying service provider-specific constraints. We have implemented an initial prototype in a grid environment and demonstrated its effectiveness in minimizing SLA violations.

Keywords

Service-Oriented Architecture, QoS assurance, Self-Adaptation

1. INTRODUCTION

Service-Oriented Architectures (SOAs) promote building software systems by integrating loosely-coupled services discovered over the network [18]. The interaction between service consumers and providers is governed by service level agreements (SLAs), which constrain not only functional aspects, but also non-functional requirements such as QoS properties of the provided service. Maintaining the promised QoS properties is a major concern for service providers in order to avoid losses and penalties.

Most research on QoS assurance in SOAs targets composite services, where managing QoS typically involves changing the service composition (e.g., replacing services by more suitable ones) [21]. Such work does not address how individual, atomic services guarantee QoS properties, which unavoidably requires controlling the underlying infrastructure. On the other hand, work that targets atomic services in the SOA context, such as [4, 19, 11], fails to address all the phases of the SLA life-cycle. This work considers QoS assurance for atomic services, taking into account the complete SLA life-cycle. In particular, the paper concentrates on atomic services that build on large-scale distributed infrastructures, such as clusters, grids or clouds. Examples include services that expose scientific HPC (high-performance computing) applications deployed on grids and services that expose multi-tier applications deployed on clusters.

Guaranteeing QoS compliance for such services is complex and involves multiple, interrelated management activities, such as negotiating QoS properties with customers, translating SLA terms to resource requirements, allocating resources and deploying service implementations on the resources. The problem is further complicated by fluctuating service workloads and unpredictable faults, very common in large-scale environments. Accommodating this dynamism requires continuous monitoring of the operating conditions and performing corrective actions, such as re-allocating resources, migrating computational elements or re-negotiating agreements. The objective of these actions is to avoid violations of SLA terms while satisfying service provider-specific constraints, such as maximizing resource utilization. In order to address these challenges, reusable mechanisms and tools for providing and enforcing quality properties are essential. Such tools must be configurable to service-provider needs while remaining applicable to a large variety of distributed infrastructures.

This work proposes a generic framework to assist service providers in enforcing quality properties in distributed environments. The framework integrates a rich set of QoS management mechanisms, including negotiation, translation and resource provisioning. Importantly, the framework supports dynamic adaptation; that is, support for monitoring and automatically modifying service behavior and resource usage. Dynamic adaptation is not well-supported by existing approaches [19, 4, 12, 19, 25, 11] and is a major focus of this paper. To increase its applicability, the framework builds on the standard SAGA API that provides a uniform and consistent interface to the most commonly used distributed functionality. A prototype of the framework has been developed; this prototype builds on the XtreamOS [7] grid and leverages the Dynaco [6] adaptation model which is based on the MAPE Autonomic Computing control loop [13].

The remainder of this paper is organized as follows. Section 2 discusses the gap in the QoS life-cycle. Following
that, the Section 3 explains the foundations of our proposal. In Section 4, we introduce the QU4DS framework, use cases and details about its current implementation. The Flac2Ogg service provider is exposed as a case study which is followed by an evaluation in Section 5. Related work is summarized in a table and further discussed in Section 6. Finally, Section 7 concludes the work and discusses future research directions.

2. THE GAP IN THE QoS LIFE-CYCLE

The Service-Oriented Architecture [18] relies on the service abstraction to support building loosely-coupled, distributed applications. Relationships between service customers and providers are defined through electronic contracts, called Service-Level Agreements (SLAs). SLAs define the expected quality of service as well as rules that apply if these expectations are violated [5]. Ensuring the quality aspects of SLAs and thus avoiding SLA violations is important to service providers in order to avoid penalties, reduce costs, and improve their reputation.

QoS assurance is addressed by the SLA life-cycle which comprises three phases [20]:

1. Description. Defines the qualities, how they will be measured, and the penalties that should be employed in case of violations.

2. Negotiation. Covers the interaction between the parties in order to agree on the contract terms.

3. Assurance. Deals with enforcing the contract, that is, guaranteeing that the agreed terms will not be violated.

Describing, negotiating and assuring QoS have generally been examined from two distinct points of view. On the one hand, QoS description and negotiation have been widely investigated by means of *which are the service qualities that service providers should expose in order to enrich their business model*. The WSLA [16] and WS-Agreements [2] specifications are widely used to describe how to describe and negotiate QoS. On the other hand, QoS ensuring mechanisms at runtime have often been investigated on clusters, grids and clouds environments by means of *which are the qualities that the resource infrastructures are able to provide*. Basically, such approaches guarantee that resource provision will be delivered according to predefined QoS as, for instance, exposed by [10]. Interestingly, both points of view have investigated QoS by only considering their own needs, visions and scopes and not thinking of the problem of describing, negotiating and assuring QoS as the same problem. While the QoS description and negotiation phases delegate QoS assurance to infrastructure layers, the latter limit themselves to understanding resource-level QoS, which is not straightforwardly translated to higher-level QoS. Even though both points of view have adopted common standards such as Web Services (WS) [24], they still limit themselves to their specific problems. Therefore, there is a gap between describing and negotiating QoS and the actual underlying mechanisms which ensure them.

3. BASIC QoS ASSURANCE CAPABILITIES

We deal with the QoS life-cycle gap (see Section 2) based on the following three core capabilities: a uniform infrastructure interface, QoS translation, and self-adaptation. The following paragraphs describe these three capabilities, which are then put together in Section 4.

3.1 A Uniform Infrastructure Usage

Developing QoS assurance mechanisms relies heavily on the interfaces and constraints exposed by the underlying distributed infrastructure. Cloud infrastructures offer a simple interface [3, 14], composed basically of the create(), terminate() and status() operations for managing virtual machines. On the contrary, infrastructures such as grids and clusters offer complex interfaces based, for instance, on batch jobs and MPI (Message Passing Interface). Despite the benefits of relying on simple interfaces, the simpler the infrastructure usage is, the more limited the capability of providing QoS assurance mechanisms is. Moreover, simpler interfaces make it harder to take full advantage of existing infrastructures. There is clearly a trade-off between the simplicity and abstraction provided by infrastructure interfaces and their level of support for developing QoS assurance mechanisms.

An interesting solution to the aforementioned trade-off is the Simple Grid API (SAGA) [9, 8], an open standard maintained by the Open Grid Forum. SAGA proposes a high-level API that simplifies grid usage and promotes grid interoperability. The API is designed to be extensible, thus increasing its applicability to different contexts. To take advantage of existing distributed infrastructures while easing their usage, we propose a uniform and simple view of the distributed infrastructure based on extending a SAGA subset. Specifically, our interface is based on the SAGA job abstraction and associated life-cycle\(^1\) and includes the most important job-related operations enhanced with price accounting. The details of each operation are as follows:

- **create(jobDescription)** Creates a job based on a description, whose only mandatory attribute is the binary file. The description might optionally include job resource requirements.
- **run()** Launches the job. The resource on which the job will run depends on the infrastructure scheduling policies and on the resource requirements, if the latter are provided.
- **cancel()** Cancels the job execution.
- **checkpoint()** Saves the current job execution state without suspending it.
- **suspend()**Suspends the job execution.
- **resume()** Continues the execution of a suspended job.
- **migrate()** Migrates the execution of a previously checkpointed and resumed job to another resource. The resource to which the job will be migrated will be chosen by the infrastructure according to its scheduling policy.

\(^1\)The SAGA job life-cycle has the following states: NEW, RUNNING, SUSPENDED, DONE, CANCELED, FAILED [8].
registerCallback(metric, callback) Subscribes to events about the current value of a job metric.

Finally, the use of the underlying infrastructure should be accounted for and linked to prices based on an appropriate pricing model, such as subscriptions, pay-per-use, or auctions. Linking prices to infrastructure-provided services is useful for using clouds as the underlying infrastructure, but this aspect is outside the scope of this paper.

3.2 QoS Translation
Executing services on top of the earlier-defined infrastructure requires mapping service requirements (e.g., response time constraints) to abstractions understood by the infrastructure (e.g., number of resources). This is not trivial, even if considering high-level infrastructure interfaces [11]. Indeed, the QoS requirements must be interpreted, understood, and possibly translated to lower-level resource requirements which are oriented to infrastructure usage. QoS translation can rely on analytical models, application profiling techniques or service implementation details. Moreover, it should be possible to interpret resource configurations in order to determine which QoS the infrastructure is able to provide. The efficiency and accuracy of the QoS translation depends on the available knowledge of service behavior and implementation. For instance, consider a throughput QoS, defined as the capability of treating 50 requests per second. This requirement could be simply translated to a ‘high-throughput’ resource requirement or to the more accurate requirement of ‘64 resources, 3GHz CPU, 16GB memory, interconnected by a high-speed network’, depending on the knowledge of service implementation details.

3.3 Self-Adaptation
Although translating QoS to resource configurations is necessary for conceiving QoS assurance mechanisms, it is insufficient for providing QoS guarantees. The mechanisms should take into account the dynamism inherent in the underlying infrastructure, the unpredictability of service demand, as well as possible SLA re-negotiations. To deal with this dynamism and unpredictability, the mechanisms should support self-adaptation [17, 1], that is, they should support adapting the service implementation dynamically and autonomously in response to changes in underlying resources and measured QoS.

To facilitate implementing self-adaptation, this work uses the Dynaco adaptation framework [6], which separates the adaptation behavior from functional interests and decomposes it into distinct adaptation concerns. Dynaco is based on the MAPE (Monitor, Analysis, Planning, Execution) autonomic model [13], as explained in the following.

Monitor Sensors gather information about specific metrics (e.g., completion time) and send it to a monitoring system, which informs subscribed entities about events.

Analysis Based on received events, the analysis decides whether any action should be taken. If so, it creates a strategy containing the adaptation goal to be performed. In Dynaco, the analysis phase is implemented by a generic decision-making engine which is driven by domain-specific policies. Policies can be defined as ECA (event-condition-action) rules of the form ‘if service task to it’ or ‘if service task is late, replicate it and get the result from the first finished replica’. Importantly, these policies can be dynamically replaced, thus allowing Dynaco to handle unforeseen scenarios, not only predictable changes.

Planning The planning phase relies on a generic planning engine, which is driven by guides. Based on those guides, the engine generates a plan with the necessary commands to implement the chosen strategy. For example, a plan for the service migration strategy would be ‘suspend(){}, checkpoint(){}, migrate(){}, resume(){}’.

Executor The Executor is in charge of executing the plan, affecting the actual changes on application execution. If there is a problem during plan execution, the monitor informs the analysis entity, which decides how to react.

4. QU4DS: QUALITY ASSURANCE FOR DISTRIBUTED SERVICES
Our goal is to address the gap in the QoS life-cycle by providing mechanisms that ensure the agreed contract terms. In order to tackle this problem, we put together the aforementioned uniform way of using distributed infrastructures, the QoS translation and the support for self-adaptation (cf. Sections 3.1, 3.2, 3.3) into a flexible framework, called QU4DS (Quality Assurance for Distributed Services). Thus, we provide a solution which is able to: (i) negotiate QoS objectives with the customer; (ii) translate QoS parameters and resource configurations in a bi-directional way; (iii) automatically deploy the service on appropriate resources; and (iv) ensure the agreed QoS by reacting to underlying infrastructures changes while keeping compliant to the QoS objectives. The following sections detail the framework architecture as well as its uses cases and implementation.

4.1 Architecture
The QU4DS architecture is depicted by Figure 1. QU4DS aims to assist service providers in providing QoS assurance for services deployed on distributed infrastructures. In a nutshell, the service provider and customer agree on an SLA that includes QoS objectives through a negotiation process. This negotiation involves QoS translation and resource checking to verify that it is possible to satisfy the required QoS. After an agreement is established, a service instance is finally deployed. During service execution, the QoS Assurance Controller verifies periodically if the execution keeps compliant to the QoS. If the QoS Assurance Controller becomes aware of any event that may impact QoS provision (e.g., resource failure, establishments of a new agreement), it reasons about a strategy for reacting to such an event. The strategy is translated to a plan and finally executed on the infrastructure that will employ the changes. The framework builds on a monitoring and actuation API that abstracts over heterogeneous infrastructures. QU4DS relies on a uniform infrastructure API based on SAGA as described in Section 3.1. The structure of the framework and its elements as shown in Figure 1 are explained next.

QU4DS It serves as the interface with service customers while coordinating the SLA management activities.
SLA Negotiator  Responsible for describing and negotiating SLAs with customers. It takes into account QoS objectives and their translation to resource configurations.

QoS Translator  It converts QoS objectives to resource configurations and vice-versa. It may rely on analytical performance models or on profiling data from previous service runs.

Service Instantiator  It discovers and allocates appropriate resources for hosting the service, and instantiates and configures the necessary service elements.

QoS Assurance Controller  Analyses events, plans and executes appropriate actions based on both service-specific QoS objective and service-provider objectives.

4.2 Use Cases

Two use cases are exposed on the sequence diagram illustrated by Figure 2 and the QU4DS architecture (cf. the numbered flows on Figure 1). The first use case represents SLA negotiation and contract establishment. In this use case, the service costumer proposes an initial contract to the SLA Negotiator by describing its desired QoS (1,2). The QoS Translator translates such QoS (3) to a resource configuration, whose availability and price are checked using the infrastructure API (4). As the required resources were not available, the SLA Negotiator contacts the QoS Translator (5) that interprets the current resource configuration and its price to the possible QoS that is able to be provided. The service costumer accepts the contract and establishes an SLA with the provider (6). Following that, the SLA Negotiator asks the Service Instantiator (7) to instantiate the service on the required resources taking into account the SLA (8,9) and configures the QoS Assurance Controller (10) to monitor the service execution.

The second use case represents QUADS-supported adaptation to avoid SLA violations. The service customer requires a service through the interface (11,12). While the service is being executed on a number of resources, an event is sent to the QoS Assurance Controller (13) warning about a resource availability limitation (e.g., resource overload, non-responding resource). Aiming at preventing an SLA violation, the QoS Assurance Controller searches for an alternative resource and verifies with the help of the QoS Translator that this resource can maintain the required QoS (14). The QoS Assurance Controller then creates a strategy that migrates the service tasks running on the previous resource to the alternative one (15,16,17). Finally, the service provider responds the request with no SLA violations (18,19).

4.3 Implementation

The current QU4DS implementation is written in Java and builds on SAGA, in particular, on its extension for the XtreemOS grid called XOSAGA. QU4DS relies on the Apache CXF framework for Web Service (WS) support and
targets services implemented according to a Master/Worker pattern. The Service Instance treats service requests by splitting the work in \( n \) tasks and asking the QoS Assurance Controller to execute them. The QoS Assurance Controller wraps these tasks as grid jobs, manages their parallel executions as \( n \) workers on XtreemOS and informs the Service Instance when they have finished. The Service Instance then merges the task outputs and responds to the service request.

4.3.1 QoS Translation and SLA Negotiation

The QoS Translator is based on log analysis profiling. The QoS parameter considered is the time to answer a request, which depends on the number of workers. Thus, QoS translation and interpretation involve determining how many workers \((n)\) are needed to satisfy a given response time \((rt)\), and, inversely, which response time \( n \) workers are able to provide. QU4DS is designed to support the WS-Agreements [2] specification which addresses both negotiation and provision interface modules by means of a language and protocol. However, the current QU4DS version only supports fix contracts represented by a WSDL binding. Furthermore, all QU4DS configuration parameters including the QoS Translator and QoS Assurance Controller parameters are set in a general configuration file called \texttt{quads.properties}.

4.3.2 The QoS Assurance Controller

With respect to the QoS Assurance Controller, QU4DS implements a simplified version of Dynaco. The details of the QoS Assurance Controller elements are explained next:

Monitor

Implements a publish-subscribe communication pattern. Any QU4DS entity can subscribe to events about a metric or a subject (i.e., a set of pre-defined metrics). Metrics can be related to jobs, resources or QoS. For example, it is possible to subscribe to events about job elapsed execution time, consumed CPU and memory as well as QoS request elapsed time. Monitoring relies on grid and QoS sensors. Grid sensors are designed to be XOSAGA customized metrics callbacks. However, due to current XOSAGA limitations, sensors are also based on UNIX scripts that monitor job metrics (CPU and memory usage, execution elapsed time, number of threads). QoS sensors are currently implemented as Java classes that feed the monitoring system with the elapsed request response time.

Analysis

It is implemented as an ECA (event-condition-action) decision-making engine whose policies are specified as parameters loaded from the QU4DS configuration file. QU4DS currently supports an adaptation strategy called Single Replacement for Late Jobs (SRLJ) as summarized in Table 1. It checks if a job execution elapsed time is greater than a threshold \((jo\text{-}bETime > j)\), and if so, it cancels the job and launches a single replacement for it. The SRLJ strategy assumes that jobs can only be replaced once (i.e., replacement jobs cannot be replaced by others), and it only decides to adapt if there is enough time for that \((\text{requestETime} < rt)\).

Planning

The plan is generated based on the SRLJ strategy actions. In the planning phase, each action is translated to \texttt{GridInterface} methods that use an XOSAGA backend as guides.

<table>
<thead>
<tr>
<th>Policies</th>
<th>Conditions</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( j ) : jobETime &gt; j</td>
<td>AND</td>
<td>1) create a job to replace the late job</td>
</tr>
<tr>
<td>( rt ) : requestETime &lt; rt</td>
<td></td>
<td>2) cancel the late job</td>
</tr>
<tr>
<td>( \text{jobETime} &gt; j ) AND ( \text{requestETime} &lt; rt )</td>
<td></td>
<td>3) submit the job replacement</td>
</tr>
</tbody>
</table>

Table 1: The SRLJ strategy implemented by QU4DS: it launches a replacement for a late job if there is enough time to adapt.

Executor

Ultimately, the XOSAGA backend executes the plan by calling the specified methods with the right parameters. Specifically, XOSAGA communicates with the XtreemOS broker which will finally perform the actual actions.

5. CASE STUDY AND EVALUATION

5.1 The flac2ogg Service Provider

As case study, we implemented the \texttt{flac2ogg} service provider which encodes FLAC (Free Lossless Audio Codec) [22] audio files to the lossy Ogg [23] audio format as depicted by Figure 3. Customer requests contain the audio file(s) to be encoded and then the service applies the \texttt{oggenc} encoder and returns the resulting Ogg file to the customer. This service implementation relies on a Master/Worker model. The master is responsible for receiving and treating service requests. When a request arrives, the master splits the contained FLAC file into segments, which are encoded in parallel by workers. QU4DS launches the workers, wrapped as jobs, on distinct resources and manages their execution by trying to minimize violations of the response time objective. Finally, the master merges the Ogg worker files and returns the audio encoded to the customer.

![Figure 3: The flac2ogg service provider encodes FLAC audio files to Ogg based on the Master/Worker pattern.](image)

The \texttt{flac2ogg} case study focuses on the request response time \((rt)\) as the QoS parameter to be assured. Therefore, the QoS Translator maps required QoS values to actual service instance configurations. The configuration parameter is the degree of parallelization, i.e., number of resources. The less the required response time is, the higher the required parallelization degree is. The QoS translator currently uses a simple mapping from QoS values to number of resources based on experimental data from previous executions.
5.2 Preliminary Evaluation

To evaluate the effectiveness of QU4DS in QoS assurance, we used the flac2ogg case study. The flac2ogg service provider was executed within a virtual machine configured with 2.4 GHz CPU clock and 1.5GB memory; the machine was acting as both an XtreemOS core and resource roles. The evaluation was based on 30 sequential customer requests to encode a 22MB flac song. The agreed SLA specified 500 seconds as a fixed response time. Its translation resulted in splitting the file in 12 parts which were encoded in parallel.

Two experiments were performed in an environment characterized by faults as exposed by Figure 4. The first experiment analyzed the system behavior in the presence of infrastructure faults without active QoS ensuring mechanisms. The faults were represented as non-responding jobs, a common situation in grid infrastructures. In particular, seven jobs were randomly stopped\(^2\) which made XtreemOS and our monitoring mechanisms assume that they were still running. We expected to have seven SLA violations, but there were twelve instead, owing to an overhead side-effect. Specifically, the stopped jobs were not fully killed by XtreemOS at the end of their requests as programmed in QU4DS due to an XtreemOS issue. This caused an extra overhead in both XtreemOS and our monitoring scripts which periodically gather information about job executions. As a consequence, there were more SLA violations than expected, mainly concentrated at the end of the request executions (Requests 20 to 30).

The second experiment analyzed the system behavior in the presence of faults but configuring QU4DS to self-adapt using the Single Replacement for Late Jobs adaptation strategy. As one can see in Figure 4 and in Table 2, QU4DS was able to decrease to half the SLA violations in comparison to the earlier experiment. When employing the SRLJ QoS assurance mechanism, there were only six SLA violations thanks to three QU4DS adaptation actions which were successfully employed to prevent further violations. While the Requests 4, 10 and 24 were successfully adapted, QU4DS could not avoid the violation of Request 19 which had a replacement job that got late and could not be replaced again. Finally, QU4DS also failed to react to Requests 8, 16, 27, 29 and 30. The cause is the amount of XtreemOS I/O operations combined with the overhead side-effect explained earlier. During these requests, an event was sent to QU4DS to inform it of the misbehaving.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Violated requests</th>
<th># adaptations</th>
<th># successful adaptations</th>
</tr>
</thead>
<tbody>
<tr>
<td>NA-F</td>
<td>12 (40%)</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>A-F</td>
<td>6 (20%)</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: QU4DS prevented further SLA violations by successfully reacting three times to environment changes.

6. RELATED WORK

We have summarized some related work in Table 3 by analyzing them according to the following criteria:

\[^2\]By using the UNIX command \texttt{kil1 \(-S70P\) PID} to suspend the execution of the \texttt{oggenc} process.
They propose an architecture that considers different actors and service provider information when translating QoS to resource requirements. However, actual QoS assurance mechanisms are not provided; they assume that the infrastructure supports them by means of self-adaptation techniques. Moreover, the authors refer to autonomy as a means of translating from SLA to resource requirements, not of ensuring QoS.

The GridWay [12] project proposes a framework that self-adapts job execution on Globus. They provide a set of tools for monitoring the job and for analyzing if their performance degrades. If this happens, they re-schedule the job on more suitable resource. Unlike QU4DS, this work supports neither negotiation nor QoS translation, thus exposing low-level resource details to applications. Secondly, the way of how they employ job self-adaptation differs from QU4DS architecture proposal since QU4DS relies on existing monitoring mechanisms not being intrusive to the grid infrastructure.

### 7. CONCLUSION AND FUTURE WORK

This paper has presented a framework, QU4DS, that facilitates QoS management for services built on distributed infrastructures, such as grids and clouds. The framework has three main features. First, the framework provides flexible support for dynamic adaptation, necessary for maintaining agreed SLAs in the face of fluctuating environmental conditions. Second, the framework supports an integrated way the complete SLA life-cycle, from contract negotiation to service termination. Finally, the framework has a modular, extensible structure, cleanly separating the different QoS management functions and service implementations. Importantly, service implementations and underlying resources are managed through a general, consistent and uniform infrastructure API, which minimizes the framework’s dependence on specific platforms and increases its applicability. The paper has also presented initial experimental results that provide evidence that QU4DS can effectively reduce SLA violations in dynamic environments.

There are two main directions for future work. First, we intend to expand the set of supported QoS properties, QoS objectives, and adaptation strategies in order to evaluate more thoroughly the extensibility and usability of the framework. The next version will include support for for WS-based SLA negotiation, allowing the framework to be validated in a real-world WS environment. Second, we intend to investigate the required mechanisms to allow service providers to integrate resources on-demand, thus allowing them to take advantage of the elasticity of cloud infrastructures and pay-per-use pricing models. In this context, work such as [15] may be used to smoothly couple SAGA and clouds.

### 8. ACKNOWLEDGMENT

The research leading to these results has received funding from the European Community’s Seventh Framework Programme [FP7/2007-2013] under grant agreement 215483 (S-CUBE).

### 9. REFERENCES


A Self-Adaptable Approach for Easing the Development of Grid-Oriented Services

André Lage Freitas and Jean-Louis Pazat
INRIA / IRISA – MYRIADS Team
Campus de Beaulieu
35042 Rennes, France
Emails: {André.Lage, Jean-Louis.Pazat}@irisa.fr

Abstract—The Service-Oriented Architecture (SOA) leverages the service abstraction to enable the development of modular, loose-coupled and distributed applications. In order to use such an architecture, service-based applications directly rely on services or compose them for conceiving new functionalities. In spite of these capabilities, they do not support the development of services which need high-performance computing. Grid computing offers an infrastructure for high-performance computing which is based on the sharing of distributed, low-cost and heterogeneous resources in large-scale. Thus, grids can be used to satisfy these high-performance service requirements. This work aims at easing the development of grid-oriented services. The iPOJO service-component model is used to propose an architecture that automatically manages job submission for services. This architecture is based on Dynaco (Adaptation for Components) and the XtreemOS grid operating system.

I. INTRODUCTION

The Service-Oriented Computing (SOC) [17] uses the service abstraction to address the development of modular and loose-coupling distributed applications. These service-based applications directly use atomic services or compose them in order to obtain other functionalities. The relationship among services are defined and established through agreements which are formalized as contracts. In order to enable the development of such applications, the Service-Oriented Architecture (SOA) proposes a support which addresses service composition as well as basic and high-level service management features. However, the SOA was not conceived to take into account non-trivial computational capabilities for services that require high-performance computing.

Grid computing [7] leverages low-cost and heterogeneous resources in order to provide a distributed infrastructure for high-performance computing in large-scale. Grids precisely define resource access through the management of Virtual Organizations (VO) in a dynamic fashion. Thereby, grids offer an infrastructure suitable for supporting services that have high-performance computing needs. In order to use the grid, such services rely on the grid job abstraction by submitting and managing them through the grid interface. However, in spite of interesting efforts as the Simple Grid API (SAGA) [8] that addresses to ease the use of grids, grid usage still remains complex. This drawback becomes an inconvenience for the development of grid-oriented services.

This work proposes a self-adaptable support for easing the conception of grid-oriented services. The self-adaptable aspect is addressed by Dynaco (Dynamic Adaptation for Components) [3] in a design based on components. We rely on iPOJO [6] due to its dynamic and flexible capabilities for developing and maintaining services on OSGi [14] platforms. Finally, a self-adaptable iPOJO handler is proposed which addresses the management of jobs on the XtreemOS grid operating system.

This work is organized as follows. Section II presents the Service-Oriented Computing and iPOJO. Grid Computing is discussed in Section III. In Section IV, we introduce dynamic adaptation and the Dynaco adaptation model. Our design is proposed in Section IV and it is followed in Section V by an architecture that describes the enabling technologies for such a proposal. We discuss about related works in Section VII. Ultimately, we conclude our discussion in Section VIII and we present our future plans concerning our investigation in this field.

II. SERVICE-ORIENTED COMPUTING

A. Overview

The Service-Oriented Computing (SOC) [17] paradigm proposes a modular and loose-couple design for developing distributed applications. Such a design is based on the service abstraction which uses contracts to define how these service will interact with each other. The SOC is represented by the Service-Oriented Architecture (SOA). The SOA describes a three-layer architecture widely known as the SOA pyramid which enables the conception of service-based applications. In a brief, the lowest SOA layer addresses basic functionalities as service discovery and binding. The middle-layer deals with service composition in which services are combined in order to conceive composite services. Last, the upper layer handles high-level service management features that are mainly related to service life cycle.

There exists different understandings of the SOA which are usually defined by open technical specifications. Firstly, we can evidence Web Services (WS) [19] due to its wide adoption. Web Services main address the integration of distinct systems by using Web standards as UDDI, XML, HTTP and SOAP. Secondly, the Service-Component Architecture (SCA) [13] specification addresses a uniform and structured way of developing services by means of defining a component model. Such a model enables component instances to implement services
through specific bindings. Finally, different from the formers, OSGi [14] addresses the management of service dynamism by proposing a flexible SOA platform framework. It is further explained as follows.

B. OSGi

The OSGi Alliance was created by a group of industries in 1999 targeting to define open specifications for services that operate on local network devices. Currently, it still aims at providing open service specifications but addressing Internet services that include desktops, homes, cars and ubiquitous devices. The OSGi framework relies on the Java technology and targets to ease the development of services that operate on “small-memory devices that can be deployed on large scale” [14]. However, there are current efforts exposed in [16] address this issue by decentralizing the platform and providing interoperability with other service technologies.

The OSGi specification is based on a modular design that takes advantage of deployment units called bundles which implement OSGi services. OSGi bundles communicate to each other in a dynamic fashion by exporting and importing Java packages. A bundle can contain one or more services that either provide or require other services. In OSGi, service descriptors are defined by Java interfaces. The OSGi framework offers then a simple and efficient way of managing bundles and their services which facilitates the application management and maintenance. However, though the OSGi framework provides such advantages, it is still the service developer charge to deal with this dynamism.

C. iPOJO

There has been proposed by [4], [20] the use of the component-based design [18] to develop services due to their complementary characteristics. On one hand, services are loose-coupling, dynamic and business-oriented. On the other hand, components offer an efficient development model that separates concerns and promotes re-usability. These properties simplify the service development. Moreover, with regard to OSGi platform, component-oriented service models such as Declarative Services [15] and iPOJO [6] do not only facilitate the service development, but they also support the management of the OSGi dynamism which is often addressed by service developers.

Among approaches that offer a component model for OSGi, iPOJO stands out due to two main aspects. First, it provides structured component composition through composites whose instances can be visualized as an architectural view of the service. Second, iPOJO’s dynamism management separates the functional and non-functional service codes which are respectively represented by POJOs¹ (content) and handlers (controllers). Thus, service developers may focus on the service main development while iPOJO handlers deal with service non-functional requirements. Such non-functional requirements may concern details about servicing and requiring services or triggering actions according to life cycle callbacks, for instance. Moreover, iPOJO provides extended handlers that offer non-trivial features as support for design patterns, event-based communication and time-based service requiring. Finally, further handlers can be implemented thanks to extensible features in iPOJO’s design and implementation.

III. GRID COMPUTING

A. Overview

Grid computing [7] addresses the use of distributed and low-cost resources for performing high-performance computing. Even though such resources may be heterogeneous and geographically dispersed, grids target to share them in a transparent fashion. In order to ensure such a transparency, grids rely on open standards that unify the way of how grid resources are used. Furthermore, concerning the resource access policies, grids leverage the concept of Virtual Organization (VO) to define and ensure user privileges in each resource. Thereby, distinct organizations or different sectors of an organization can configure the grid usage to suit their needs.

Grid applications take advantage of the job abstraction and may consist of one or more grid jobs. Jobs are commonly formed by a parallel program whose tasks may be related to each other or not. Moreover, job descriptions contain both the program binary and the resource requirements to execute it. These requirements may describe the number and type of resources and detailed information about them which may include operating system, libraries, input data, for instance. With respect to the job execution, grids propose distinct interfaces such as APIs, terminals and web portals through which jobs can be submitted and managed. Job management commands are dealt with by the grid broker which is in charge of allocating the resources according to the job requirements. Then the broker sends the job-resources mismatch to the grid scheduler in order to execute the job. Once the job finishes, the results are stored in the user’s space in the grid file system.

In spite of the grid benefits earlier exposed, grid usage still remains complex. Moreover, each grid platform provides its own interface with customized commands. In order to ease the grid application development and to promote the interoperability among different grids, the Grid Simple API (SAGA) [8] proposes a standardized grid API. SAGA allows grid applications to be developed taking into account a unique and simple interface while it enables the execution of such an application in distinct grids. Moreover, SAGA achieves this goal by leveraging grid adaptors which can be developed under grid interfaces written in Java or C++ languages currently.

B. XtreemOS

The XtreemOS project proposes the idea of a grid operating system in contrast to conventional grid approaches based on grid middlewares. XtreemOS uses the Linux kernel to build specific modules which transparently enables Mandriva GNU/Linux PCs and mobile devices to become grid resources. Moreover, it also supports single-system image clusters which grid users can use as it would be a single and powerful

¹Plain Old Java Object.
machine. With respect to resource management, XtreemOS takes advantage of such kernel implementation to decrease the system overhead when performing tasks that require detailed information of the system. Finally, XtreemOS interface (XOSAGA) is based on SAGA by providing higher-level abstractions for developing grid applications; besides XtreemOS also provides a POSIX-compliant terminal interface.

The design of the XtreemOS grid operating system is divided in foundation and grid layers. The former addresses the low-level kernel implementations that allow to use different hardwares as grid resources. Thus, the communication infrastructure is built under those kernel modules and allows both foundation and grid layers to interact with each other. The latter layer is responsible for providing grid functionalities. It consist of three entities that provide Application Execution Management (AEM), Virtual Organization Management (VOM) and the XtreemOS File System (XFS). Those functionalities are transparently offered as in conventional operating systems thanks to the foundation layer which offers such a transparency.

IV. Dynamic Adaptation

Systems that rely on dynamic environments should be able to adapt themselves in order to deal with unpredicted changes. Dynamic environments commonly present changes which are not able to predict at design time. These changes may lead systems to face undesirable situations which degrade and compromise the application working. Moreover, it is harder to both predict and deal with such changes in distributed scenarios. Therefore, the adaptation of applications at runtime becomes then a fundamental feature in order to ensure their proper behavior.

In [3], [10], [1], [12] there are further discussions about adaptation techniques. While [1], [12] address general issues and enabling technologies for software adaptation, [3], [10] propose an adaptation design that divides distinct adaptation concerns: Monitor, Analyze, Plan and Execute\(^2\). On one hand, in [10], the authors propose a general architecture for Autonomous Computing. On the other hand, Dynaco (Dynamic Adaptation for Components) [3] proposes an adaptation model validated by a framework for developing self-adaptable application on grids. Furthermore, Dynaco promotes the re-usability of adaptation generic mechanisms which can be customized according to the adaptation domain.

The Dynaco adaptation model proposes to separate the adaptation implementation from the application functional code. It leverages the component-based design by addressing the adaptation implementation as component controllers. Thus such a controller is based on an adaptation functionality decomposition which separates the four aforementioned adaptation aspects. These adaptation aspects are represented by the following entities whose relationships and further details are described as follows.

1) Monitoring: The monitor is in charge of gathering information that are related to adaptation interests. It relies on pull and push flows in order to either keep other entities aware about a change (i.e., event-based communication) or to let them ask about specific metric measures.

2) Decision: When a change occurs, the decider decides whether it is enough relevant for performing an adaptation or not. If so, it concerns about what should be done by means of a strategy which contains the goal that should be achieved in order to change the application behavior to achieve a proper state. For instance, it could be “load balancing among resources R1, R2 and R3” because R1 presents performance degradation due to three concurrent processes p1, p2 and p3.

3) Planning: Once the decider defines which strategy would be employed, it is sent to the planner that translates such a high-level goal to a set of instructions as a plan. According to the earlier example, the plan would be an instruction as “migrate process p2 to R2 and p3 to R3”.

4) Execution: Thereby, the plan is sent to the executor which performs the plan that will finally make the component adapt to a more suitable configuration. For achieving this, it must intercept the application flow execution and then execute the received instructions.

V. Towards a Self-Adaptable Support for Grid-Oriented Services

In order to use the grid, service developers must interact with the grid interface which is commonly employed through the grid API. Thus grid jobs can be submitted and managed by invoking respective methods. However, the task of managing jobs is complex, time consuming and error prone. We understand that service developers should be free from this task and finally focus on main aspects of the service development. Consequently, it claims a support which is able to transparently manage job submissions for services aiming at easing the development of grid-oriented services. Furthermore, such a support should consider QoS related to the job execution in order to properly satisfy service specific needs.

We can outline two technologies that enable the conception of automatic job management for grid-oriented services. The first one the component-based design which have been used to facilitate the development of services [6], [15], [20]. They allow to separate functional from non-functional services requirements which facilitates both service development and maintenance. Moreover, components do not only ease the development of services, but they also are a suitable technology to enable the job management for services. By using components, it is possible to separate the job management task from other interests related to the service development.

The second technology that enables to automatically manage jobs is self-adaptation. Even though grids offer a robust platform for executing complex applications, grids are dynamic distributed systems which face unforeseen changes. Some of these changes may affect the job execution in such a way that would require restarting or even resubmit the job. In order to deal with such a scenario, self-adaptive
techniques ought to be exploited when providing a support for job management. Finally, we combine components and self-adaptation as the foundation of a job self-management support for services. This support uses the Dynaco (Dynamic Adaptation for Components) adaptation model and it is further explained as follows.

### A. Proposal Overview

Our goal is to ease the development of services that use grids by providing an integrated approach which automatically manages job submissions. Figure 1 summarizes and positions our proposal. Firstly, we tackle the composition layer of the SOA pyramid, more precisely services that need to execute grid jobs. Secondly, we rely on services which are based on components due to great advantages that such a design offers as earlier explained. Then we separately deal with the job management and the component functional concerns by addressing the former as a component controller and the latter in the component content. As follows, we use the Dynaco adaptation model to tackle the self-adaptive behavior of the job management support. Finally, the grid is used as the underlying infrastructure for executing the job. Furthermore, we assume that services have agreed on QoS values which is then used to guide the adaptation strategy. However, though agreement negotiation is taken into account, it is not in the scope of our proposal. In [9], it is further discussed how such a negotiation can be performed as well as the translation of SLA QoS to resource-level QoS related to the job execution.

### VI. A Self-Adaptable iPOJO Handler for XtreemOS

#### A. Architecture

In this section it is presented an architecture that describes how the previous design proposed in V can be employed. First, as our goal is to ease the development of grid-oriented services, we propose to use technologies that facilitate such a development. They are explained as follows:

1) **SOA Platform:** We leverage the OSGi platform which proposes a dynamic and modular framework for developing and maintaining services.

2) **Component-Oriented Service Model:** iPOJO rightly supports the management of OSGi services. It provides a component model that dynamically manages service bindings while allowing to define details about both servicing and requiring services.

3) **Adaptation Model:** Dynaco proposes an adaptation model which separately addresses distinct adaptation aspects. It offers a clear design for developing self-adaptive techniques.

4) **Grid Infrastructure:** In order to have a deeper control of grid resources in a transparent way, we use the XtreemOS as grid infrastructure. It offers the XOSAGA higher-level interface and transparently deal with resource management.

Based on those technologies, we propose the design of an iPOJO handler that relies on the Dynaco adaptation model and performs job self-management on the XtreemOS grid.
operating system as depicted in Figure 2. The monitoring system uses the iPOJO Event Admin and has information about the jobs, their requirements and QoS through the component metadata file. Besides the monitoring system keeps the decider, planner and executor informed about these facts as well as about the grid environment. Finally, jobs are submitted and managed on XtreemOS through XOSAGA commands sent by the executor.

B. An iPOJO Handler for XtreemOS

We take advantage of the extensible iPOJO component model that allows to implement further iPOJO handlers. iPOJO handlers are implemented by extending the PrimitiveHandler abstract class as illustrated by Figure 3. The PrimitiveHandler class inherits theFiledInterceptor and the MethodInterceptor classes that allow iPOJO handlers to employ introspective techniques. iPOJO exploits meta-information of OSGi bundles in order to trigger actions according to method invocations and filed accesses. This is a powerful mechanism to implement dynamic adaptation techniques due to its capacity of performing actions based on information which is only available at runtime. Last, once an iPOJO handler is implemented, it is only necessary to define its XML scheme to make it available for iPOJO components.

In Figure 3, the XtreemOSSelfAdaptableHandler class represents how the XtreemOS iPOJO handler can be implemented. The Dynaco adaptation model is addressed by the PrimitiveSelfAdaptableHandler which separately deals with monitoring, decision, planning and execution concerns. This class should be used to implement generic adaptation mechanisms as decision-making engines. Thereby, each iPOJO self-adaptable handler ought to inherit this class by customizing it with domain-specific adaptation knowledge. With respect to the XtreemOS handler, this customization is represented by the XtreemOSSelfAdaptableHandler class. Finally, iPOJO flexible design allows iPOJO handlers to be used by each other, thus taking further advantages of usability as we propose to use the iPOJO Event Admin handler when implementing the monitoring system.

C. Usage

The use of the XtreemOS iPOJO Handler is based on the Job Submission Description Language (JSDL) [2] and requires proper XtreemOS certificates. The JSDL relies on the XML standard to define the job and describe its resource requirements. Listing 1 exposes an example of a JSDL job. TheExecutable attribute at line 10 points to the program that comprises the job. It is followed by its arguments at line 11, the output and error files at lines 12 and 13. At line 18, we can realize that it was chosen 1 as the number of resources to run the job. Furthermore, with regard to security issues on XtreemOS, it uses X.509 certificates whose public keys must be informed to let the grid identify the user and grant right permission access on grid resources.

```xml
  <JobDescription>
    <Executable>
      <Program>myProgram</Program>
      <Arguments>arg1 arg2 arg3</Arguments>
      <OutputFile>output.txt</OutputFile>
      <ErrorFile>error.txt</ErrorFile>
    </Executable>
    <NumberOfResources>1</NumberOfResources>
    <X.509Certificate>publicKey</X.509Certificate>
  </JobDescription>
</JobDefinition>
```
As iPOJO component descriptions are addressed in the component metadata.xml file, we propose to add there the job (i.e., a path to the JSDL file) as well as the XtreemOS user certificate and the job execution QoS. In Listing 2, there is an example of such a metadata file. In order to use the XtreemOS iPOJO Handler, the component requires it at line 2. At line 6, it is defined which class implements the component. Line 8 and 9 define the use of the XtreemOS iPOJO Handler and the job to be submitted respectively. As follows, some QoS are required such as a fault-tolerant execution that relies on three job execution replicas (line 11) whose data should also be twice replicated and stored in a persistent way (lines 12 and 13).

VIII. Conclusion and Future Work

This work presents an approach that aims at easing the development of grid-oriented services. We propose a self-adaptable architecture that enables the automatic management of job submissions and considers QoS related to the job execution. Such an architecture leverages the iPOJO component model to enable the conception of OSGi services that use the grid. An iPOJO handler is proposed by leveraging Dynaco (Dynamic Adaptation for Components) to automatically address the self-adaptable job management on the XtreemOS grid infrastructure.

Furthermore, the scope of our approach is confined to services which must be aware of grids. By that very fact and intending to ease the development of such services, we put together dynamic and flexible service development and job self-management. On one hand, it presents an interesting approach for grid-oriented services. On the other hand, our proposal enforces service developers to deal with grid jobs by exposing the grid infrastructure to other service developers that do not need to execute grid jobs but do need a distributed, large-scale and heterogeneous infrastructure for executing services. In future works, we will investigate how generic services can be executed using grids. That leads us to still exploit job self-management but in such a way that services could be executed with no knowledge of grids.

VII. Related Work

Some approaches have targeted the development of grid-oriented services [11], [9]. In [11], the authors propose an approach for automatically managing grid applications. They rely on SOA standards as Web Services to define the communication among grid functionalities. In [9], the authors propose an architecture that translates high-level service requirements to resource-level requirements. They provide an automatic way of negotiating both requirements by establishing contracts which define specific QoS. However, both approaches do not target to ease the development of grid-oriented services as we do. In other words, [11] focus on providing automatic mechanisms for managing grid applications and [9] aim at a autonomous architecture for translating and negotiating QoS. They have such specific goals rather than providing a solution that brings together both service development easing and job self-management.

Ultimately, the Web Service Resource Framework [5] addresses a grid interface driven to Web Services. It adds the idea of stateful Web Services that describes grid resource states in order to manage their life cycle. Despite the fact that it enables grids to conceive service-based applications, the job management is still the service developer charge. In contrast, we propose to automatically manage the job submission by using the component-based design and an adaptation model that eases the conception and maintenance of dynamic and flexible services.

ACKNOWLEDGMENT

The research leading to these results has received funding from the European Community’s Seventh Framework Programme [FP7/2007-2013] under grant agreement 215483 (S-CUBE).
REFERENCES


CLAM: Cross-layer Adaptation Manager for Service-Based Applications

Asli Zengin
Fondazione Bruno Kessler
Trento, Italy
zengin@fbk.eu

Annapaola Marconi
Fondazione Bruno Kessler
Trento, Italy
marconi@fbk.eu

Marco Pistore
Fondazione Bruno Kessler
Trento, Italy
pistore@fbk.eu

ABSTRACT
The heterogeneous and dynamic execution context of service-based applications (SBA) makes the problem of adaptation critical. However, in most cases the adaptation is not trivial due to the following facts. First, SBA has a complex layered system where the application is implemented through a composition of services, which in turn are provided by platforms and run on top of infrastructures. Second, as a result of this multi-level application system there exist several adaptation approaches isolated from each other, which focus on a specific concern of one level ignoring the overall impact of the adaptation on the whole service-based system. To tackle this problem we propose a cross-layer adaptation manager (CLAM) whose contribution is two-fold: (i) It provides a platform that integrates and coordinates existing analysis and adaptation tools, which target specific system concerns, to assess the impact of an adaptation at the different levels. (ii) Covering the whole system for the SBA, it provides an analysis algorithm that incrementally constructs consistent adaptation strategies starting from an initial adaptation trigger originated at any level. The paper introduces the proposed approach and presents its first implementation with concrete analysis and adaptation tools.

Categories and Subject Descriptors
C.2.4 [Computer Systems Organization]: Distributed Systems—Distributed applications; C.4 [Computer Systems Organization]: PERFORMANCE OF SYSTEMS—Reliability, availability, and serviceability; H.3.4 [Information Systems]: Systems and Software—Performance evaluation (efficiency and effectiveness); K.6.4 [Computing Milieux]: System Management—Quality assurance

General Terms
Design, Management, Performance

Keywords
adaptation management, service-based applications, cross-layer adaptation
The manager is based on a comprehensive high-level model of the application and of the layers behind it. Each model element is associated with (i) a set of analyzers, specific to the different concerns, to understand the problem, (ii) a set of solvers, to identify possible solutions, and (iii) a set of enactors, to apply them on the element. The coordinated operation of analyzers, solvers, and enactors is governed by domain-specific rules that identify the dependencies, and consequences, between the elements of the model and run the different tools. For each adaptation need, CLAM produces a tree of the possible alternative adaptations, identifies the most convenient one, and applies it. The paper exemplifies all main concepts through a simple application for the smart management of taxi reservations.

The rest of the paper is organized as follows. Section 2 motivates and explains the need for holistic, cross-layer adaptations. Section 3 introduces the system representation for the proposed solution, Section 4 presents the overall approach and the system architecture behind it, and Section 5 presents a first implementation of the approach. Section 6 surveys some related approaches, while Section 7 concludes the paper.

2. PROBLEM STATEMENT

In this section we present a scenario to illustrate the cross-layer adaptation problem addressed in this paper (Figure 1). It will be used also to explain our approach in the next section.

We define a service-based system (SBS) as a system that is composed of an application layer where the service-based application takes place along with its non functional process properties such as key performance indicators (KPI), a service layer, which corresponds to the partner services of the process forming a platform for it, and finally the underlying infrastructure layer for the composite service (application) and the partner services. Our scenario is an SBS with the following layer characteristics:

- **Application layer:** “Call & Pay Taxi” composite service (CPTS), implemented as a BPEL process.
- **Service layer:** The partner services of our application, namely a short messaging service (SMS), a location service (LS) and a payment service (PS) provided by the telecom company, and the taxi service (TS) provided by the taxi company.

- **Infrastructure layer:** The underlying platforms on top of which CPTS, SMS, LS, PS and TS run.

In CPTS, the client requests a taxi by sending a text message (SMS) to the application. Then, his/her location is identified and the taxi company is contacted to organize the real taxi service. After transporting the user to destination, the process terminates with a successful payment.

On this scenario we would like to show an adaptation case to motivate our problem. The CPTS provider decides to reduce the overall cost and consults the business analyst. Given that technologically it is already possible to replace services in the process, business analyst decides to switch to a cheaper telecom provider. This means replacing SMS, LS and PS in the BPEL process. However, there is a problem with the new LS service’s output message format. It provides client’s location in geographical coordinates instead of full address while in our application design we use full address as the input message to the taxi service. To address this data mismatch problem, the service composition is adapted by adding a mediator service in the workflow. Basically it converts geographical coordinates into the full addresses. Yet it triggers a new problem: the new service introduced for data mediation increases the overall cost of the process in an unforeseen way. The new workflow is functionally correct but the actual goal that led to the substitution of the telecom provider, that is cost reduction, is not met. In fact, the updated version of CPTS is more expensive than the original one.

**Problem.** With the existing approaches when an adaptation is performed, it targets a particular problem occurring at a specific concern of a specific SBS layer. Thus, they tend to propose local solutions to local problems in a way that is isolated from the overall application context. To avoid this, we must understand the impact of a change across different layers, which have their own characteristics and constraints. In our example while we are trying to improve the cost, one of the application key performance indicators (KPI), we do not know the consequences of replacing the services of the new telecom provider for the whole SBS. Consequently, we need an approach to meet the following requirements [20]:
R1. Identify the problems that might occur in different concerns of the SBS layers due to an adaptation.

R2. If there exist problems, tackle them by proposing new adaptations in a consistency with the overall system.

We believe that addressing this problem is not trivial considering the complex, layered structure of the service-based systems as well as the numerous existing analysis and adaptation approaches proposed and practiced independently from each other. In next sections of this paper, we present CLAM that handles the complexity of the problem and meets the requirements enumerated above.

3. SBS REPRESENTATION FOR CLAM

For our CLAM approach, we need a proper representation of the service-based system that we introduced in the previous section.

In service-based systems, each layer has its own elements and each element has its own concerns. E.g. at the application layer while KPI and process are layer elements, application execution time is a concern related to the KPI element, and data flow is a concern related to the process element.

SBS’s layered structure already contains the dependencies implicitly among different layer elements while those dependencies are not trivial considering adaptions coming from different layers. Hence, in our technique we benefit from the layer dependencies and model the SBS in a cross-layer manner so that the dependencies get explicit to be used easily by our tool.

We follow a graph-like approach for the cross-layer modeling of the SBS, which we call “Cross-layer System Model” (Figure 2).

In this model the graph nodes represent the system elements from different layers and the graph edges represent the cross-relations between these elements. We have “process”, “process activity” and “KPI” elements for the application layer; “service provider”, “service”, “service operation” and “service QoS” elements for the service layer; and finally “infrastructure provider”, “infrastructure” and “infrastructure QoS” elements for the infrastructure layer.

The graph edges in the model display the dependency relation between different system elements. We distinguish two types of dependency relations: has and consumes. While has can only be a relation between two elements from the same layer, consumes defines the inter-layer relations whether an element from a layer relies on another element from a different layer. E.g. in infrastructure layer, the provider has the infrastructure and the infrastructure has the QoS attributes. On the other hand, since services are running on top of infrastructures, naturally service element from the service layer has a consumes relation with infrastructure element from the infrastructure layer. Similarly, for instance service availability depends on the infrastructure availability as well. Then, naturally there is a consumes relation between service QoS and infrastructure QoS.

To each system element we associate the set of relevant application concerns. For instance, attached to the process element we have the data flow concern (e.g. matching of exchanged message formats, satisfaction of data flow requirements) and the control flow concern (e.g. satisfaction of service composition requirements). CLAM allows to attach to each concern a set of adaptation capabilities that can be used to deal with its specific adaptation problems.

There are several state-of-the-art approaches for various types of SBS analysis and adaptation, and they already run in the system environment in an uncoordinated way. In our approach we would like to reuse those mechanisms to analyze the impact of an adaptation trigger and to extend it if required. Thus, we would like to include them in our “Cross-layer System Model”. Each existing approach which can be used by our approach is called an adaptation capability.

We can have three different types of capabilities: (i) analyzers, (ii) solvers and (iii) enactors. While an analyzer checks the compatibility of a new adaptation for a system concern, a solver gets an incompatibility problem triggered by an analyzer and tries to propose an adaptation plan to handle the problem. After CLAM completes its overall analysis through a continuous interaction with analyzers and solvers, validated final adaptations are applied in the system through enactors. E.g. for the concern “process execution time” at KPI system element, we can attach a time analyzer. On the other hand, a process migrator is an enactor that can migrate the running process instances to a new process model that can be attached to the process system element.

Apart from “Cross-layer System Model”, another important concept of the proposed approach is the “Instant System Configuration” Mi; an instantiation of the “Cross-layer System Model” with concrete system elements. E.g., for our reference scenario, CPTS, we can instantiate a system configuration with the current BPEL file in use for the execution, partner services SMS, LS, PS that are provided by the current telecom provider Vodafone, and the TS that is provided by RadioTaxi Trento. Similarly we can instantiate each graph node with concrete system elements and overall graph instantiation will be an Mi.

We keep an Mi as a combination of an Instant Process Model for the application layer, an Instant Service Model for the service layer and an Instant Infrastructure Model for the infrastructure layer. Each of those models is an XML file keeping the concrete instances of the relevant system elements.

Before moving on to the next section we would like to mention that although CLAM needs as input the cross-layer representation of the application domain where main elements of the system are introduced as nodes of the graph and the edges correspond to the relationships among them, the CLAM algorithm to make the analysis is not domain specific. While in this paper we have the taxi reservation scenario, which is introduced as an SBS; CLAM is able to work with other application domains too, as long as the application system can be modeled by its elements and their dependencies to each other.

4. CLAM ARCHITECTURE

The system architecture of our approach, given in Figure 3, has two main parts: the core CLAM platform, implemented in Java, where the entire analysis is orchestrated, and the external capabilities that we plug-in to the platform.

CLAM core consists of the following main parts: Cross-layer rule engine where CLAM rules are implemented for the overall supervision of the execution, a number of checkers each of whom is responsible to act as an interface between CLAM and a capability that is plugged in the platform, model updater that takes care of updating instant system configuration whenever a new adaptation proposal enters CLAM, tree constructor where the results of the cross-layer adaptation analysis are continuously updated and kept in a tree-form structure, and lastly the strategy ranker where the final tree is traversed to output validated and ranked adaptation strategies, which are safe to be realized.

Cross-layer Rule Engine, implemented by using JBOSS Drools,1 is the fundamental part of the entire system. It consistently communicates with all the other parts of the CLAM, and through the execution of its rules it initiates, guides and completes an adaptation analysis. Its main functionality is to update the cross-layer analysis tree based on the exchanged information with the analyzers and solvers. Communication with those capabilities are through

1http://www.jboss.org/drools
Analyzers and Checkers are responsible for discovering the necessary input to be provided to an analyzer or a solver. Basically each capability produces a report as an output and the rule engine decides on the next step depending on the report type it gets from the component.

When the rule engine is activated due to an initial trigger or due to a solver report, the rule engine passes the adaptation information to the model updater so that the instant system configuration $M_i$ can be updated. As well as sending the new system configuration $M_i$ to the rule engine, the model updater also keeps track of all the versions of the instant process, service and infrastructure models. In this way, when a final adaptation strategy is decided by CLAM, the corresponding system configuration can be retrieved and realized in the SBS.

Model Updater. Whenever a new adaptation proposal enters the system due to an initial trigger or due to a solver report, the rule engine passes the adaptation information to the model updater so that the instant system configuration $M_i$ can be updated. As well as sending the new system configuration $M_i$ to the rule engine, the model updater also keeps track of all the versions of the instant process, service and infrastructure models. In this way, when a final adaptation strategy is decided by CLAM, the corresponding system configuration can be retrieved and realized in the SBS.

Tree Constructor. At each time a new analysis is initiated due to an initial trigger CLAM creates a new tree structure in order to keep track of the result of each capability invocation. We call it CLAM Tree, $T$, and each time CLAM receives back a report from a capability it adds an edge and a node to the $T$ where a tree edge $E_{tree}$ keeps the status of that report; a tree node $N_{tree}$ keeps an updated instance of the system configuration $M_i$ and an updated instance of the Queue of the capabilities to be invoked. CLAM continues to construct the tree until the overall analysis is completed.

Strategy Ranker. When the construction of a cross-layer adaptation tree is completed, next step is to extract the adaptation paths from the tree and to rank them based on our predefined criteria. Each path from the root node to a green leaf is an alternative adaptation proposal. CLAM consumes the analysis at the process element is more emergent for us compared to a time or cost analysis at KPI element because, before analysing its performance, we want to make sure that an updated process is correct in terms of compositional requirements.

The Cross-layer Adaptation Manager (CLAM) is an extended version of the Cross-layer System Model that includes a prioritized queue of capabilities to be invoked. When a new adaptation proposal enters the system due to an initial trigger or due to a solver report, the rule engine passes the adaptation information to the model updater so that the instant system configuration $M_i$ can be updated. As well as sending the new system configuration $M_i$ to the rule engine, the model updater also keeps track of all the versions of the instant process, service and infrastructure models. In this way, when a final adaptation strategy is decided by CLAM, the corresponding system configuration can be retrieved and realized in the SBS.

5. IMPLEMENTATION

We have a prototype implementation for CLAM. In this implementation we have five real tools (three analyzers and two solvers) from the state-of-the-art, plugged in the CLAM platform and CLAM...
coordinates them to produce a solution for the case study that we introduced in section 2

Below are the details of the tools that we used for our case study. However, our approach is not dependent on this case study, the implementation of CLAM is already extensible to accommodate new analyzers or solvers in the system.

QoS4BPEL is used both for time and cost analyses in our platform [8]. It gets as input the BPEL file of the application and the execution times and costs of each process activity in the BPEL, then it produces an aggregate value for them. Then those values can easily be compared with KPI target values to detect if a KPI is violated or not regarding a new process model.

Mismatch Patterns and Adaptation Aspects is used as both datanet analyzer and data mismatch solver [11]. In datanet analysis, whenever a new service is introduced to the composition, the compatibility of the new service interface is checked against the data flow requirements of the process. Mismatch Patterns and Adaptation Aspects tool gets as input the BPEL file and the WSDL file of the new service and then performs the analysis based on some predefined mismatch patterns for the application. If a mismatch is identified, the tool is also capable of producing an adaptor, i.e., a data mediator, which can be introduced to the composition as a service.

Structural Adjustment of BPEL is used as process optimizer to reduce the execution time of a process [18]. While in the case study we used it for time optimization, the tool is capable of optimizing the memory usage of the process as well. It gets as input the BPEL file and produces an optimized BPEL file if any task parallelization is possible for the process.

The resultant tree that CLAM produces for our case study is given in Figure 4. When we receive the new adaptation proposal “replace telecom provider X by Y” from the business analyst (human), first CLAM checks which graph nodes are to be updated in the instant models. Then CLAM invokes the model updater asking to replace in the instant service models the old services with those of the new telecom provider (i.e. SMS, LS and PS). From the updated system configuration CLAM identifies that service provider, service, service operation and service QoS system elements are to be updated. This overall action implies an update in the instant system configuration (from $M_0$ to $M_1$). Then CLAM reasons about the graph nodes that might be affected due to these changes in the configuration and finds out that process activity and KPI system elements might be affected since they consume the changed elements in the system. Next step is the decision of which analyzers to invoke in order to validate the telecom provider replacement. CLAM applies a principal rule for this purpose: It puts in the queue all the analyzers that are attached to the changed and affected graph nodes, in our example, DataNet analyzer that is connected to the process activity element, and time and cost analyzers that are connected to the KPI element in the cross-layer system model. After the creation of the queue, analyzers are invoked one by one and each time a report is received, tree is updated with the new tree edge and the tree node. If during the whole process, an analyzer reports back an adaptation need, then in this case, CLAM searches for a solver to be put in the queue, which can address the problem and produce an adaptation proposal. E.g. in the tree in Figure 4, DataNet analyzer reports back a datamismatch problem for the new LS, then CLAM discovers and invokes the MismatchSolver, and this capability produces three alternative mediator services (A, B, C) that can be added in the process to solve the data incompatibility problem.

After building the tree, we can select an adaptation strategy among alternative paths. Suppose that in the tree that we built for our scenario we select the rightmost path because the final application configuration in this path has the best QoS values. In this case the corresponding new application model would be as in Figure 5 where the updated system elements are highlighted in yellow, i.e., the new process activity that is responsible for data mediation and its new underlying mediator service DMS, the replaced TelCo services LS’, PS’ and SMS’, our modified composite service CPTS’ and finally the parallelized activities in the process.

The construction of the tree in Figure 4 took around 3 seconds on the 2 GHz Intel Pentium M Processor Windows XP laptop with 1 GB of RAM.

6. RELATED WORK

Existing work for SBA adaptation mostly focuses on specific aspects of the application where adaptation problems are solved merely in this narrow scope without taking into account the con-
sequences for the whole SBA stack. BPM adaptation [14, 3], dynamic service binding [12, 7], self adaptation and self healing systems [13, 5], QoS-awareness [15, 6], mediator design for service interactions [11] and finally context-awareness [1] constitute these prevalent aspects. In principle these approaches might serve our platform as solver/analyzer capabilities and in this way they can be aligned with other SBA concerns and be more effective.

Yet, there are few approaches in literature that propose the use of some cross-relations for adaptation. In [9] coordinated adaptation is introduced for multiple applications interacting with each other in the same environment. These applications are not composed, but rather single entities which are affected by the same contextual attributes such as sharing common resources. The authors claim that a coordinated platform is important before adapting these applications to the context changes since they have interactions in the same environment. This problem is addressed in a narrow scope, only in terms of management of shared resources. Another problem they mention is conflict resolution for two adaptation mechanisms. However, they expect the user to perceive the conflict, and to modify the mechanisms accordingly.

The approach in [10] analyzes the dependencies of KPIs on process quality factors from different functional levels of an SBA such as QoS parameters, and then an adaptation strategy is decided to improve all the negatively affected quality metrics in the SBA. This work proposes a set of adaptation solutions for KPI violations through the consideration of the non-functional dependencies of SBA layers. However, the work is at preliminary stage and it is not clear based on what kind of rules they decide on which adaptation actions to perform in order to tackle a KPI violation.

Another approach is [17], which presents a framework for cross-layer adaptation of service oriented applications that comprise of organization, coordination and service layers. The application stack is similar to the SBA stack, which we introduce in this paper. They propose a technique where cross-layer adaptation designer prepares the taxonomies of adaptation mismatches, and later designs the adaptation templates, also known as patterns, that define generic solutions to tackle mismatches. Their technique directly models cross-layer adaptation templates, i.e., their dependencies are known from design time. Eventually they are like fixed patterns for cross-layer adaptation cases. Instead in our approach we model the application in a cross-layer fashion and then discover the cross-layer adaptation paths on the fly through the coordination of available tools and mechanisms.

Like our CLAM approach, the works [2, 19, 4] have a cross-layer representation of the application model. While [19, 4] target limited number of adaptation cases such as service replacement, [2] makes use of the cross-layer model for monitoring and analysis rather than adaptation.

7. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a holistic SBA management framework, CLAM, that can deal with cross- and multi-layer adaptation problems. This is achieved in two ways: on the one hand CLAM identifies the application capabilities affected by the adaptation actions, and on the other hand it identifies an adaptation strategy that solves the adaptation problem by properly coordinating a set of concern-specific adaptation capabilities. The proposed solution relies on a comprehensive high level system model that allows to easily extend the framework by plugging-in new adaptation capabilities. We implemented the proposed framework for the service-based application, taxi reservation.

In the future we plan to consolidate our implementation and evaluate it on application scenarios entailing higher level of complexity and requiring to deal with different system layers and concerns (e.g. Cloud-based applications).

8. ACKNOWLEDGMENT

The research leading to these results has received funding from the European Community’s Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube).

9. REFERENCES


Stepwise and Asynchronous Runtime Optimization of Web Service Compositions

Philipp Leitner, Waldemar Hummer, Benjamin Satzger, and Schahram Dustdar

Distributed Systems Group, Vienna University of Technology
Argentinierstrasse 8/184-1, A-1040, Vienna, Austria
lastname@infosys.tuwien.ac.at

Abstract. Existing research work considers runtime adaptation of service compositions as a viable tool to prevent violations of service level agreements. In previous work we have formalized the optimization problem of identifying the most suitable adaptations to prevent a predicted set of violations, and presented suitable algorithms to solve this problem. Here, we introduce the idea of stepwise optimization as a solution to the problem of how to deal with situations when the optimization result is not available in time, i.e., when decisions need to be taken before the optimization problem can be fully solved.

1 Introduction

In information systems based on the concept of Service-Oriented Architecture (SOA), business processes are implemented as higher-level compositions of Web services (service compositions [1]). Providers of service compositions often guarantee certain quality characteristics using service level agreements (SLAs). Basically, SLAs are collections of target qualities (service level objectives, SLOs) and monetary penalties that go into effect if the promised target quality cannot be achieved. Hence, providers of service compositions have strong incentives to prevent cases of SLA violation. One promising approach to achieve this is predicting violations at runtime, before they have actually occurred, and using adaptation to prevent these violations [2, 3]. Evidently, an important part of runtime adaptation is deciding which adaptations to apply. We argue that this decision should be based on both, the costs of violation (the penalties associated with SLOs), and the costs of adaptation. We have presented a formalization of this decision process as an optimization problem as part of our work on the PREvent (Event-Based Prediction and Prevention of SLA Violations) project [4]. However, a limitation of the PREvent approach so far is that it is inherently assumed that the optimization problem can be solved in time, before the first adaptation has to be applied. Even using fast meta-heuristics this is not guaranteed, especially so for shorter service compositions.

* The research leading to these results has received funding from the European Community’s Seventh Framework Programme [FP7/2007-2013] under grant agreement 215483 (S-Cube) and grant agreement 257483 (Indenica).
In this paper, we improve on this by proposing an asynchronous and stepwise optimization model, in which we do not generate a decision for all adaptations at once. Instead, we run the optimization in parallel to the execution of the composition. At so-called decision points we make a decision for only those adaptations that absolutely need to be decided at that moment, update the optimization problem according to the made decisions, and continue optimizing for all remaining possibilities for adaptation. Note that the contribution presented here is not specific to the approach presented in [4]. Much more, the same ideas are applicable for other runtime adaptation approaches facing similar problems as well, e.g., [3].

2 Runtime Optimization of Service Compositions

The decision which combination of runtime adaptations is best suited to prevent one or more predicted SLA violations in an instance of a service composition can be formulated as an optimization problem. For this paper the concrete structure of this optimization problem is not essential. However, for ease of understanding we briefly summarize the formalization used in PREvent, which is the basis of the remaining discussions. This model has originally been presented in [4].

We assume the following inputs to the optimization. Let $I$ be the set of all possible instances of the service composition, and let $i \in I$ be one instance that we need to optimize. Furthermore, let the relevant SLA be given as a set of SLOs $S = \{s_1, s_2, \ldots, s_k\}$. Every SLO $s$ has an associated penalty function, which governs the payment that the composite service provider has to pay based on a measured SLO value $m_s$. Penalty functions are defined as $p_s : \mathbb{R} \rightarrow \mathbb{R}, s \in S$. We refer to the collection of all penalty functions as $P = \{p_{s_1}, p_{s_2}, \ldots, p_{s_k}\}$. Moreover, let $A = \{a_1, a_2, \ldots, a_l\}$ be the set of all possible adaptations, and $A^* \in \mathcal{P}(A)$ one concrete subset of adaptations. Applying adaptations transforms composition instances, which we capture with the $\circ$ operator ($\circ : I \times \mathcal{P}(A) \rightarrow I$). For simplicity, we assume that all adaptations have constant costs, defined as $c : A \rightarrow \mathbb{R}$. However, adaptations are not necessarily independent, i.e., there can be constraints on which adaptations can be selected at the same time. We use a simple penalty term to express that two adaptations are mutually exclusive ($v(A^*) = \infty$ if $A^*$ contains at least one constraint violation, $v(A^*) = 0$ otherwise).

With these definitions, we can can describe the total costs ($TC$) of a composite service provider for one instance of the business process as $TC(A^*) = v(A^*) + \sum_{s_x \in S} p_{s_x}(i \circ A^*) + \sum_{a_x \in A^*} c(a_x) \rightarrow min!$ In this definition, the first term is the potential penalty for constraint violations in $A^*$. The second term represents the costs accrued via penalty payments for SLA violations. Finally, the third term represents the costs of adaptation. Naturally, the goal of the provider is to select $A^*$ so that $TC$ is minimal for a given $i$.

Evidently, the exact penalties that have to be paid ($p_{s_x}(i \circ A^*)$) are unknown at runtime for any combination of adaptations (even if we do not apply any adaptations, we still do not know for sure which SLOs are going to be violated). However, we assume that it is possible to predict the penalty before and after
adaptation with a reasonably small estimation error, using a set of estimation functions \( e_s : I \rightarrow \mathbb{R}, s \in S \). We can then replace the penalty payments with their estimation, leading to the following estimation: \( TC(A^*) \approx v(A^*) + \sum_{s_x \in S} e_{s_x}(i \circ A^*) + \sum_{a_x \in A^*} c(a_x) \rightarrow \min \)

In practice, estimation can be implemented for instance use machine learning based regression, as presented in [5, 6]. Using this technique one can estimate monitorable SLO values \( m_s \) in advance, and use these estimated values to calculate what the penalty will be after applying a given set of adaptations. This is the approach that we have chosen to follow in the PREvent framework, but in principle our model is not restricted to these machine learning based estimation functions.

3 Stepwise Service Composition Optimization

The problem presented in Section 2 can be solved at runtime, however, generating a good solution may be time-consuming. One promising approach that we have utilized in PREvent with good results are genetic algorithms [7] (GA). In the remainder of the paper, we assume that the runtime optimization of \( A^* \) is implemented using GA, however, the general principles presented here still apply if other means are used. However, even using GA, optimization is still time-consuming. Therefore, in order not to delay the execution of the composition, it is desirable to execute the optimization asynchronously, i.e., in parallel to the service composition. However, in this case we need to keep timing aspects of the optimization and the composition in mind.

Before explaining optimization timing, we need to concretize what adaptation actually means in the scope of this paper. In general, we assume that adaptation can either be implemented via service rebinding, i.e., exchanging a service in the composition for another, or via structural adaptation of the composition, i.e., freely adding, removing or modifying activities in the composition. Figure 1 exemplifies these types of adaptation. This adaptation model is in line with related work, as other approaches to self-adapting compositions usually assume similar possibilities for adaptation [8, 9]. The excerpt in Figure 1 is a small part of an assembling case study presented in [4]. We will use this example in the remainder of the paper. Note that even though we present our work on a simple sequential process for simplicity, the same ideas can be used for arbitrarily complex composition graphs, as long as they are circle-free.

For any adaptation of the types discussed above we can identify the affected region in the service composition, i.e., the activities in the composition which are affected by the adaptation. For rebinding, this is exactly one activity. For structural adaptation the affected region may be arbitrarily large, but is still always clearly defined. In the remainder of this paper, we use the term “beginning of the affected region” \( t_a^x \) as the time that the first activity affected by adaptation \( x \) starts to execute. In Figure 1, the beginning of the affected region is indicated by \( X \).
3.1 Timely Optimization and Stale Results

Figure 2 showcases the timing of asynchronous optimization. For an instance of the composition, an optimization is triggered at time $t_0$ (e.g., because SLO estimation mechanisms have predicted that this instance is going to violate its SLA). A meta-heuristic optimization algorithm starts searching the solution space. Meanwhile, the service composition continues executing. Firstly, assume that at time $t_1$ the optimization has converged and delivers the result that two adaptations have to be applied. For both actions the affected region has not yet been reached, i.e., the optimization was timely and the result is useful. However, if we assume now that the algorithm takes more time and delivers its result at time $t_2$. Now, the activity “Invoke Planning” (P) has already been executed, and part of the result (the decision to adapt P) came too late.

Intuitively, for every adaptation $x$ with a defined affected region there is also a decision point $t_d^x$, the latest time in the execution of the composition when a decision needs to be made. Assuming that we know $t_d^x$, and the time that the application of the adaptation technically takes ($d_x$), we can define the decision
point of an adaptation as \( t_x^d = t_x^a - d_x \). We refer to an optimization result \( A^* \) produced at time \( t \) as stale if \( \exists a \in A^*: t_a^a < t \). Stale optimization results cannot be applied in full anymore when they are available, and evidently should be avoided.

### 3.2 Stepwise Optimization

Two approaches can be used to handle the problem of stale results. Firstly, one can decide not to deal with the problem at all, ignoring stale adaptations and applying only what is still possible when the result becomes available. This approach is very simple, and even in the worst case this is at least never worse than not doing optimization to begin with, even if the result may be suboptimal in the presence of stale results. Secondly, one can drop the idea of asynchronous optimization and halt the service composition while the optimization is running. This trivially prevents stale results, but severely degrades the performance of the service composition. It is well possible (if the optimization takes more time than what can be gained using adaptation) that using this approach is actually worse than not doing any optimization at all. It is easy to see that both of these ideas are not optimal.

![Stepwise Asynchronous Optimization](image)

**Fig. 3.** Stepwise Asynchronous Optimization

Hence, we now introduce stepwise asynchronous optimization as an alternative principle to prevent stale results. The general approach is sketched in Figure 3. First of all, we order all adaptations according to their decision points. Actions with identical decision points \( (t_x^d = t_y^d) \) are collected in decision sets \((D_i)\). Let \( t_s^d \) be the decision point of a decision set \( D_x \), defined as the decision point of all adaptations contained in the set \( (\forall a \in D_i: t_a^a = t_s^i) \). In the figure, two decision sets with decision points \( t_s^P \) and \( t_s^C \) exist. The first decision set contains only the action “Invoke Planning” (P), the second contains only the action “Check Parts” (C).

As before, an optimization is triggered at \( t_0 \). However, results are now delivered differently as in the naïve approach sketched before. Instead of waiting for...
the optimization to converge, we now trigger the decision procedure sketched in
the algorithm in Figure 4 when $t_p^k$ and $t_c^k$ are passed.

For every decision point associated with a decision set, we briefly halt the com-
position and decide on all adaptations associated with this set. If it is decided
to apply one or more adaptations, we do so. Finally, we resume the composi-
tion. This way, instead of producing the solution for the optimization problem
in one big bang (and risking that the result arrives too late), we use a stepwise,
constructive approach which defers all decisions about adaptations to the latest
possible time, but never later. Hence, results are never stale, and the service com-
position needs to be halted only briefly. Note that it is generally advantageous
to wait as long as possible before making a decision (per definition, that means
waiting until $t_x^d$), under the assumption that the quality of a decision is monoton-
ically increasing with optimization time. This is true for most implementations
of optimization algorithms, including GA (if elitism is used).

The decision algorithm in Figure 4 contains three separate challenges. Firstly,
one needs to be able to apply adaptations at runtime (Line 8). We do not discuss
solutions for this problem here, and refer the reader to existing work instead [4].
Secondly, we need to be able to actually make a decision on a single adapta-
tion based on a still ongoing optimization (Line 6). One simple, yet promising,
strategy is to base the decision on the best currently known intermediary re-
sult, i.e., decide to apply an adaptation if and only if the currently best known
solution applies the adaptation. We refer to this strategy as current-optimum
based decision. Thirdly, after deciding on an adaptation, the target function of
the optimization problem needs to be modified. This is to reflect the fact that
whenever a decision is made (and a given adaptation is applied or rejected) the
underlying problem of the ongoing optimization has in fact changed. If the adap-
tation has been applied, all solutions that do not use this adaptation are invalid.
Similarly, if the adaptation has not been applied, all solutions using the adap-
tation are invalid. This change is easily represented using an additional penalty
term $v_x$ in the target function. For instance, if $a_x$ is applied, a new penalty term
$v_x$ defined as $v_x(A^*) = \infty$ if $a_x \notin A^*$, and $v_x(A^*) = 0$ otherwise, is added to the

---

```plaintext
1 decide(DecisionSet ds, Optimization opt):
2 suspend process() 
3 foreach(Adaptation a in ds):
4     decision = decide on adaptation(a, opt)
5     if (decision == true)
6         apply adaptation(a)
7         adapt target function(a, decision, opt)
8 resume process()
```

Fig. 4. Decision Procedure for Stepwise Optimization
target function. Hence, in Line 9 of the algorithm, we pause the optimization algorithm, add an additional constraint for each adaptation in the decision set and resume optimizing.

4 Related Work

The general idea of optimizing running composition instances is related to the larger research field of QoS-aware service composition. QoS-aware service composition is usually a static process, which aims at finding the best instantiation of an abstract composition before or during deployment. The optimization problem is finding the most suitable combination of concrete services. The seminal work that introduced this idea already dates back to 2004 [10], however, newer research is still able to provide new insights. For instance, [11] defined a domain-specific language from which dynamic QoS-optimized compositions are generated. [12] improved on the methods of optimization that are used in QoS-aware composition, and proposed to use a combination of local selection and global optimization. The approach presented in this paper differs from all these contributions in that we do not consider optimization statically. Indeed, traditional QoS-aware composition does not face the problem of timeliness at all, as optimization is done once and is not repeated for every problematic instance, as it is the case in our approach. The research work most closely related to this paper is work on runtime adaptation of service compositions. First ideas on self-adaptive compositions can be found in [13], even if this work is more closely related to adaptive workflows than service compositions. Recently, work in this area seems to have gravitated towards using the aspect-oriented programming (AOP) paradigm to technically implement adaptation, as exemplified by [9,14]. Other approaches use pure service rebinding [8] or parametrization of compositions [15]. The PREvent framework [2] supports adaptation on many different levels, and forms the basis of the research work presented in this paper. Other research of note in the area of adaptation include [3], which uses machine learning techniques similar to the mechanisms used in PREvent to trigger adaptation. All of these approaches have a similar base premise (optimization of the performance of a service composition through monitoring and runtime adaptation), but none discusses timing aspects explicitly. We argue that our work is complementary to all these approaches, and similar ideas as discussed here may be worthwhile to incorporate in any framework that aims at optimizing running composition instances.

5 Conclusions

In this paper we have introduced the problem of stale results in the optimization of service compositions, and proposed stepwise optimization as a possible solution to tackle this issue. As future work, we plan to investigate how appropriate stepwise optimization is for usage with different optimization algorithms, e.g., Ant Colony Optimization or Simulated Annealing, and compare the stepwise optimization approach with quick construction heuristics.
References

FCM: an Architecture for Integrating IaaS Cloud Systems

Attila Csaba Marosi, Gabor Kecskemeti, Attila Kertesz, Peter Kacsuk
MTA SZTAKI
Computer and Automation Research Institute
of the Hungarian Academy of Sciences
H-1528 Budapest, P.O.Box 63, Hungary
Email: {atisu, kecskemeti, keratt, kacsuk}@sztaki.hu

Abstract—Cloud Computing builds on the latest achievements of diverse research areas, such as Grid Computing, Service-oriented computing, business processes and virtualization. In this paper, we reveal open research issues by envisaging a federated cloud that aggregates capabilities of various IaaS cloud providers. We propose a Federated Cloud Management architecture that acts as an entry point to cloud federations and incorporates the concepts of meta-brokering, cloud brokering and on-demand service deployment. The meta-brokering component provides transparent service execution for the users by allowing the system to interconnect the various cloud broker solutions available in the system. Cloud brokers manage the number and the location of the utilized virtual machines for the received service requests. In order to fast track the virtual machine instantiation, our architecture uses the automatic service deployment component that is capable of optimizing service delivery by encapsulating services as virtual appliances in order to allow their decomposition and replication among the various IaaS cloud infrastructures. Our solution is able to cope with highly dynamic service executions by federating heterogeneous cloud infrastructures in a transparent and autonomous manner.

Keywords—cloud federation; cloud brokering; IaaS; virtual appliance.

I. INTRODUCTION

Highly dynamic service environments [1] require a novel infrastructure that can handle the on demand deployment and decommission of service instances. Cloud Computing [2] offers simple and cost effective outsourcing in dynamic service environments and allows the construction of service-based applications extensible with the latest achievements of diverse research areas, such as Grid Computing. Service-oriented computing, business processes and virtualization. Virtual appliances (VA) encapsulate metadata (e.g., network requirements) with a complete software system (e.g., operating system, software libraries and applications) prepared for execution in virtual machines (VM). Infrastructure as a Service (IaaS) cloud systems provide access to remote computing infrastructures by allowing their users to instantiate virtual appliances on their virtualized resources as virtual machines.

Nowadays, several public and private IaaS systems co-exist and to accomplish dynamic service environments users frequently envisage a federated cloud that aggregates capabilities of various IaaS cloud providers. These IaaS systems are either offered by public service providers (like Amazon [3] or RackSpace [4]) or by smaller scale privately managed infrastructures. We propose an autonomic resource management solution that serves as an entry point to this cloud federation by providing transparent service execution for users. The following challenges are of great importance for such a mediator solution: varying load of user requests, enabling virtualized management of applications, establishing interoperability, minimizing Cloud usage costs and enhancing provider selection.

This paper proposes a layered architecture that incorporates the concepts of meta-brokering, cloud brokers and automated, on-demand service deployment. The meta-brokering component allows the system to interconnect the various cloud brokers available in the system. The cloud broker component is responsible for managing the virtual machine instances of the particular virtual appliances hosted on a specific infrastructure as a service provider. Our architecture organizes the virtual appliance distribution with the automatic service deployment component that can decompose virtual appliances to smaller parts. With the help of the minimal manageable virtual appliances the Virtual Machine Handler rebuilds these decomposed parts in the IaaS system chosen by the meta-broker. As a result, the cloud broker component uses the VM Handler to maintain the number of virtual machines according to the demand.

Related works have identified several shortcomings in the current cloud infrastructures [5]: e.g., federated clouds will face the issue of scalability and self-
management similarly to Grid systems, or users of the cloud systems should be in control of their computing costs. We propose an architecture that aims at both of these problems by allowing users to utilize metabrokering between public and private cloud systems as a result lowering their operation costs. Our architecture also handles the issue of scalability by offering the cloud brokers that manage the virtual machines according to the actual demands of the user applications.

This paper is organized as follows: first, we introduce the related research results in Section II. Then, we discuss an advanced use case in Section III that involves our proposed architecture and discusses its advantages in contrast to previous research results. Next, we detail the operational roles of the brokering components in our architecture in Section III-A and Section III-B. Afterwards, in Section IV, we discuss an optimization approach to rebuild virtual appliances within the virtual machine that is used to execute them. Finally, we conclude our research in Section V.

II. RELATED WORK

Matthias Schmidt et al. [6] investigate different strategies for distributing virtual machine images within a data center: unicast, multicast, binary tree distribution and peer-to-peer distribution based on BitTorrent. They found the multicast method the most efficient, but in order to be able to distribute images over network boundaries ("cross-cloud") they choose BitTorrent. They also propose to use layered virtual machine images for virtual appliances consisting of three layers: user, vendor and base. By using the layers and a copy-on-write method they were able to avoid the retransmission of images already present at the destination and thus decrease instantiation time and network utilization. The authors only investigated distribution methods within the boundaries of a single data center, going beyond that remained future work.

There are several related works focusing on providing dynamic pool of resources. Paul Marshall et al. [7] describe an approach for developing an "elastic site" model where batch schedulers, storage and web services can utilize such resources. They introduce different basic policies for allocating resources, that can be "on-demand" meaning resources are allocated when a service call or task arrives, "steady stream" assumes steady utilization, thus leaves some elastic resources continuously running, regardless of the (temporary) shortage of tasks, or "bursts" for fluctuating load. They concentrate on dynamically increasing and decreasing the number of resources, but rely on third party logic for balancing load among the allocated resources. Constantino Vázquez et al. [8] are building complex grid infrastructures on top of IaaS cloud systems, that allow them to adjust the number of grid resources dynamically. They focus on the capability of using resources from different cloud providers and on the capability of providing resources for different grid middleware, but meta-scheduling between the utilized infrastructures and developing a model, that considers the different cloud provider characteristics is not addressed.

In 2009, Amazon Web Services launched Amazon CloudWatch [9], that is a supplementary service for Amazon EC2 instances that provides monitoring services for running virtual machine instances. It allows to gather information about the different characteristics (traffic shape, load, disk utilization, etc.) of resources, and based on that users and services are able to dynamically start or release instances to match demand as utilization goes over or below predefined thresholds. The main shortcoming is that this solution is tied to a specific IaaS cloud system and introduces a monetary overhead, since the service charges a fixed hourly rate for each monitored instance.

Mohsen Amini et al. [10] are focusing on so called marketing-oriented scheduling policies, that can provision extra resources when the local cluster resources are not sufficient to meet the user requirements. Former scheduling policies used in grids are not working effectively in cloud environments, mainly because Infrastructure as a Service providers are charging users in a pay-as-you-go manner in an hourly basis for computational resources. To find the trade-off between to buy acquired additional resources from IaaS and reuse existing local infrastructure resources he proposes two scheduling policies (cost and time optimization scheduling policies) for mixed (commercial and non-commercial) resource environments. Basically two different approaches were identified on provisioning commercial resources. The first approach is offered by the IaaS providers at resource provisioning level (user/application constraints are neglected: deadline, budget, etc.), the other approach deploys resources focusing at user level (time and/or cost minimization, estimating the workload in advance, etc.).

III. FEDERATED CLOUD MANAGEMENT ARCHITECTURE

Figure 1 shows the Federated Cloud Management (FCM) architecture and its connections to the corresponding components that together represent an interoperable solution for establishing a federated cloud environment. The FCM targets the problem area outlined in the Introduction, and provides solutions for most of the listed open issues. In the following, we exemplify
the interaction of the main components of this solution through a low level use case.

In this scenario we restrict our solution to support standard stateless web services described with WSDL [11]. Using this solution, users are able to execute services deployed on cloud infrastructures transparently, in an automated way. Virtual appliances for all services should be stored in a generic repository called FCM Repository, from that they are automatically replicated to the native repositories of the different Infrastructure as a Service cloud providers.

When a user sends a service call to the system, he/she submits a request to the “Generic Meta-Broker Service” (GMBS) specifying the requested service with a WSDL, the operation to be called, and its possible input parameters. The GMBS checks if the service has an uploaded VA in the generic repository, then it selects a suitable CloudBroker for further submission. The matchmaking is based on static data gathered from the “FCM Repository” (e.g., service operations, WSDL), and on dynamic information of special deployment metrics gathered by the CloudBrokers. Currently we use the average VA deployment time and the average service execution time for each VA. VA deployment time assumes that the native repository already has the requested VA, thus includes only the service provision time on a specific IaaS cloud. The role of GMBS is to manage autonomously the interconnected cloud infrastructures with the help of the CloudBrokers by forming a federation.

Each “CloudBroker” has an own queue for storing the incoming service calls (called \( Q_1 \) and \( Q_2 \) in Figure 1), and manages one virtual machine queue for each VA \( V A_x \rightarrow V M Q_x \). Virtual machine queues represent the resources that currently can serve a virtual appliance specific service call. The main goal of the CloudBroker is to manage the virtual machine queues according to their respective service demand. The default virtual machine scheduling is based on the currently available requests in the queue, their historical execution times, and the number \((n, m, o, p)\) of running VMs. The secondary task of the CloudBroker involves the dynamic creation and destruction of the various \( V M Qs \).

Virtual Machine Handler (“VM Handler”) components are assigned to each virtual machine queue. These components process the virtual machine creation and destruction requests placed in the queue. The requests are translated and forwarded to the corresponding IaaS system \( C l o u d_x \). This component is a cloud infrastructure-specific one, that uses the public interface of the managed infrastructure.

Independently from the virtual machine scheduling process the CloudBroker also handles the queue of the incoming service calls. As a result, these calls are dispatched to the available VMs created in the previously discussed manner.

In order to optimize service executions in highly dynamic service environments, our architecture organizes the virtual appliance distribution as a background process with the automatic service deployment component that can decompose virtual appliances to smaller parts. With the help of the minimal manageable virtual appliances (MMVA – further discussed in Section IV) the Virtual Machine Handler is able to rebuild these decomposed parts in the IaaS system on demand, that results in faster VA deployment and in a reduced storage requirement in the native repositories.

In the following, subsections we detail how resource management is carried out in this architecture. At the top-level, a meta-broker is used to select from the available cloud providers based on performance metrics, while at the bottom-level, IaaS-specific CloudBrokers are used to schedule VA instantiation and deliver the service calls to the clouds.

A. Top-level Brokering in FCM

As we already mentioned in the scenario discussed in the previous section, brokering takes place at two levels in the FCM architecture: the service call is first submitted to the Generic Meta-Broker Service (GMBS – that is a revised and extended version of the Grid Meta-Broker Service described in [12]), where a top-level decision is made to that cloud infrastructure the call should be forwarded. Then the service call is placed in the queue of the selected CloudBroker, where the bottom-level brokering is carried out to select the VM that performs the actual service execution. This bottom-level brokering and the detailed introduction of the architecture of the CloudBroker is discussed later in Section III-B.

Now, let us turn our attention to the role of GMBS. An overview of its architecture is shown in Figure 2. This meta-brokering service has five major components. The Meta-Broker Core is responsible for managing the interaction with the other components and handling user interactions.

The MatchMaker component performs the scheduling of the calls by selecting a suitable broker. This decision making is based on aggregated static and dynamic data stored by the Information Collector (IC) component in a local database. The Information System (IS) Agent is implemented as a listener service of GMBS, and it is responsible for regularly updating static information from the FCM Repository on service availability, and aggregated dynamic information collected from the Cloud-
Brokers including average VA deployment and service execution times. The Invoker component forwards the service call to the selected CloudBroker and receives the service response.

Each CloudBroker is described by an XML-based Broker Property Description Language (BPDL) document containing basic broker properties (e.g., name), and the gathered aggregated dynamic properties. The scheduling-related attributes are typically stored in the PerformanceMetrics field of BPDL. More information on this document format can be read in [12]. Namely, the following data are stored in the BPDL of each CloudBroker:

- Estimated availability time for a specific virtual appliance in a native repository – collected from the FCM Repository;
- average VA deployment time and average service execution time for each VA – queried from the CloudBroker;

The scheduling process first filters the CloudBrokers by checking VA availability in the native cloud repository, then a rank is calculated for each broker based on the collected static and dynamic data. Finally, the CloudBroker with the highest rank is selected for forwarding the service request.

B. CloudBroker

The CloudBroker handles and dispatches service calls to resources and performs resource management within a single IaaS system, it is an extended version of the
The system described in [13].

The architecture of the CloudBroker is shown in Figure 1. Its first task is to dynamically create or destroy virtual machines ($V_M^2$) and VM queues ($V_MQ_2$) for the different used virtual appliances. To do that, first, the VA has to be replicated to the native repository of the IaaS system from the FCM Repository (an alternative method is discussed in Section IV). Alongside the appliance, the FCM Repository also stores additional static requirements about its future instances, like its minimum resource demands (disk, CPU, and memory), that are needed by the CloudBroker. This data is not replicated to the native repository, rather the FCM Repository is queried.

A VM queue stores references to resources capable of handling a specific service call, thus instances of a specific VA. New resource requests are new entries inserted into the queue of the appropriate VA, while resource destruction requests are modification of entries representing an already running resource. The entries are managed by the VM Handler, that is a cloud fabric specific component designed to interact with the public interface of a single IaaS system. It simply translates and forwards requests to the public interface of the IaaS system ($\text{Cloud}_a$). Each VA contains a monitoring component deployed, that allows the CloudBroker to monitor the basic status (CPU, disk and memory usage) of the running resources along the average deployment time for each VA and average service execution times. These data can be queried by the IS Agent of the GMBS.

The service call queue ($Q_1$ and $Q_2$) stores incoming service requests and, for each request, reference to a VA in the FCM Repository. There is a single service call queue in each CloudBroker, while there are many VM queues. If the native repository does not contain the requested VA it is replicated first. Dynamic requirements for the VA may be specified with the service call:

- Additional resources (CPU, memory and disk);
- an UUID, that allows to identify service calls originating from the same entity.

The UUID will allow to meet SLA constraints later, e.g., to enforce a total cost limit on public clouds for service calls of the entity, or to be in compliance with deadlines. If any dynamic requirements are present the CloudBroker treats the VA as a new VA type, thus creating a new VM queue and starts a VM. The service calls may now be dispatched to the appropriate VMs. Most IaaS systems offer predefined classes of resources (CPU, memory and disk capacity) not adjustable by the user, in this case the CloudBroker will select the resource class that has at least the requested resources available. This may lead to allocating excess resources in some cases (e.g., the resource class has twice the memory requested to meet the CPU number requirement).

The CloudBroker also performs the scheduling of service call requests to VA’s and the life-cycle management of resources. Scheduling decision is made based on the monitoring information gathered from the resources. If the service request cannot be scheduled to any resource the CloudBroker may decide to start a new VM capable of serving the request. The decision is based on the following:

- The number of running VM’s available to handle the service call;
- the number of waiting service calls for the VA in the service call queue;
- the average execution time of service calls;
- the average deployment time of VA’s;
- and SLA constraints (e.g., total budget, deadline);

VM decommission is also based on the above, but the CloudBroker takes into account the billing period of the IaaS system; shutdown is performed only shortly before the end of the period with regard to the average decommission time for the system.

IV. VIRTUAL APPLIANCE DELIVERY OPTIMIZATION

IaaS systems require virtual appliances to be stored in their native repositories. Only those virtual appliances, that were previously stored in these repositories, can be used to instantiate virtual machines. Our architecture allows users to upload their virtual appliances to the FCM Repository that behaves as an active repository and handles the distribution of the appliances to the native repositories according to [14]. As an active repository, the FCM repository identifies the common parts of the appliances and decomposes them into smaller packages that allow appliance delivery and rebuilding from multiple repositories.

Central virtual appliance storage would require the VM Handler to first download the entire appliance from the FCM repository to a native one, then instantiate the appliance with the IaaS system. To avoid the first transfer, but keep the convenience for the users of our architecture, we have investigated options to rebuild virtual appliances in already running virtual machines. We have identified two distinct approaches for rebuilding: (i) native appliance reuse, (ii) minimal manageable virtual appliances. The first approach utilizes already available virtual appliances in the native repositories and extends them towards the required virtual appliance. In this article, we do not aim at this approach because it requires the investigation of the publicly available
appliances in order to find the appliance most suitable for extension.

The second approach proposes the minimal manageable virtual appliance that we define as basic appliance with the following three properties:

- Offers content management interfaces to add, configure and remove new appliance parts.
- Offers monitoring interfaces to analyze the current state of its instances (e.g., provide access to their CPU load, free disk space and network usage).
- Optimally sized: only those files present in the appliance that are required to offer their extensibility with the previously mentioned interfaces.

As a result, our architecture only needs to replicate the MMVAs to every native repository. If the FCM repository identifies high demands for specific virtual appliance parts, then the active repository functionality automatically replicates the appliance to those IaaS systems where most requests were originated from.

Our VM Handler is prepared to control virtual appliance rebuilding using minimal manageable virtual appliances. Consequently, the VM Handler applies a new strategy when it receives a virtual appliance instantiation request for a specific appliance that is not available in the native repository. This strategy starts with the instantiation of the MMVA. Next, the Handler waits until the virtual machine of the MMVA has started up. Then, it requests the content management interfaces to add the parts of the specific appliance that were identified as unique during the decomposition of the MMVA and the specific appliance. As a result, the specific appliance is rebuilt and ready to serve the scheduled service requests in the virtual machine instantiated for the MMVA.

V. CONCLUSION AND FUTURE WORKS

In this paper, we proposed a Federated Cloud Management solution that acts as an entry point to cloud federations. Its architecture incorporates the concepts of meta-brokering, cloud brokering and on-demand service deployment – their interaction is exemplified through a low-level use case. The meta-brokering component provides transparent service execution for the users by allowing the system to interconnect the various cloud broker solutions managed by aggregating capabilities of these IaaS cloud providers. We have shown how CloudBrokers manage the number and the location of the utilized virtual machines for the various service requests they receive. In order to fast track the virtual machine instantiation, our architecture uses the automatic service deployment component that is capable of optimizing its delivery by decomposing and replicating it among the various IaaS cloud infrastructures. Regarding future works, we plan to investigate various scenarios that arise during handling federated cloud infrastructures using the FCM architecture (e.g., the interactions and interoperation of public and private IaaS systems).

ACKNOWLEDGMENT

The research leading to these results has received funding from the European Community’s Seventh Framework Programme FP7/2007–2013 under grant agreement 215483 (S-Cube) and 261556 (EDGI).

REFERENCES

Abstract—Cloud Computing offers simple and cost effective outsourcing in dynamic service environments, and allows the construction of service-based applications using virtualization. By aggregating the capabilities of various IaaS cloud providers, federated clouds can be built. Managing such a distributed, heterogeneous environment requires sophisticated interoperation of adaptive coordinating components. In this paper we introduce an integrated federated management and monitoring approach that enables autonomous service provisioning in federated clouds. In this architecture, cloud brokers manage the number and the location of the utilized virtual machines for the received service requests. In order to provide seamless service executions, a state of the art monitoring solution is proposed that supports cloud selection performed by the management layer of the architecture. Our solution is able to cope with highly dynamic service executions by federating heterogeneous cloud infrastructures in a transparent and autonomous manner.

Keywords—Cloud Computing; Service Monitoring; Cloud Brokering; On-demand deployment;

I. INTRODUCTION

Cloud Computing [1] offers simple and cost effective outsourcing in dynamic service environments and allows the construction of service-based applications extensible with the latest achievements of diverse research areas, such as Grid Computing, Service-oriented computing, business processes and virtualization. Cloud-based highly dynamic service environments [3] require a novel infrastructure that incorporates a high-level monitoring approach to support autonomous, on demand deployment and decommission of service instances. Virtual appliances (VA) encapsulate a complete software system (e.g. operating system, software libraries and applications) prepared for execution in virtual machines (VM). Infrastructure as a Service (IaaS) cloud systems provide access to remote computing infrastructures by allowing their users to instantiate virtual appliances on their virtualized resources as virtual machines. Nowadays, several public and private IaaS systems co-exist provided by public service providers (like Amazon [4] or RackSpace [5]) or by smaller scale privately managed infrastructures. Cloud solutions are also spreading fast in the academia with the emerging open-source tools, such as Eucalyptus [6] and OpenNebula [7].

However, user demands are frequently overextending the boundaries of a single cloud system. In these cases, they need to handle the differences between the various cloud providers and have to negotiate their requirements with multiple parties. Federated clouds aim at supporting these users by providing a single interface on which they can transparently handle the different cloud providers as they would do with a single cloud system. This paper proposes an architecture to construct federated cloud systems that not only offers a single interface for its users but it automatically manages their virtual machines independently from the currently applied cloud system. We argue that efficient cloud selection in federated clouds requires a cloud monitoring subsystem that determines the actual health status of the available IaaS systems.

We present an architecture that incorporates the concepts of on-demand service deployment, cloud brokering and meta-brokering, supported by an integrated monitoring solution. The meta-brokering component allows the system to interconnect the various cloud brokers available in the system. It is also responsible for selecting a proper execution environment managed by a cloud broker. This selection process relies on a sophisticated monitoring component, which provides up-to-date service availability and infrastructure reliability based on specific monitoring metrics. The cloud broker component is responsible for managing the virtual machine instances of the particular virtual appliances hosted on a specific IaaS provider. Our architecture also organizes virtual appliance distribution with its automatic service deployment component that can decompose and deliver virtual appliances in smaller parts.

Related works have identified several shortcomings in the current cloud infrastructures [2]: e.g. federated clouds face the issue of scalability, self-management and losing
complete control on computing costs. Our solution aims at these problems by allowing users to utilize meta-brokering between public, academic and private cloud systems as a result lowering their operation costs. Our architecture serves as an entry point to this cloud federation by providing transparent service execution for users. The following challenges are of great importance for such a mediator solution: varying load of user requests, enabling virtualized management of applications, establishing interoperability, minimizing cloud usage costs and enhancing provider selection. Therefore the main contributions of this paper are: (i) an advanced, integrated monitoring solution to support meta-brokering decisions for user requirement-based service provisioning, and (ii) a holistic view of interoperable federated clouds managed by a multi-level resource management architecture.

This paper is organized as follows: first, we gather related approaches in Section II. In Section III we introduce our proposed architecture and discuss its main components in three subsections. In Section IV, we present a simplified scenario that we use to exemplify our approach. Finally, we conclude our research in Section V.

II. RELATED WORK

In 2009, Amazon Web Services launched Amazon CloudWatch [8], which is a supplementary service for Amazon EC2 instances that provides monitoring services for running virtual machine instances. It allows gathering information about the different characteristics (traffic shape, load, disk utilization, etc.) of resources, and based on that users and services are able to dynamically start or release instances to match demand as utilization goes over or below predefined thresholds. The main shortcoming is that this solution is tied to a specific IaaS cloud system and introduces a monetary overhead, since the service charges a fixed hourly rate for each monitored instance.

Yigitbasi et. al. [15] introduced a solution for cloud performance monitoring called C-Meter. Using this framework, workloads can be submitted to target clouds to analyze their performances. On the contrary, our monitoring solution examines the real, running applications instead of workloads, and does not necessary require additional deployments.

Regarding federated management of different infrastructures, GridBot [10] represents an approach for execution of bags-of-tasks on multiple grids, clusters, and volunteer computing grids. It has a Workload Manager component that is responsible for brokering among these environments, which is similar to our approach, but we rather target multi-cloud solutions and focus on highly dynamic service executions instead of tasks more suitable for volunteer grids.

M. Schmidt et al. [9] investigate different strategies for distributing virtual machine images within a data center: unicast, multicast, binary tree distribution and peer-to-peer distribution based on BitTorrent. They found the multicast method the most efficient, but in order to be able to distribute images over network boundaries ("cross-cloud") they choose BitTorrent. They also propose to use layered virtual machine images for virtual appliances consisting of three layers: user, vendor and base. By using the layers and a copy-on-write method they were able to avoid the retransmission of images already present at the destination and thus decrease instantiation time and network utilization. The authors only investigated distribution methods within the boundaries of a single data center, going beyond that remained future work.

With respect to monitoring the provisioned services, the existing technical approaches found in the literature to gather the required data can be classified into two big categories. On one hand, some proposals rely on the use of monitoring directives as follows: Introducing monitoring directives into the services themselves using Aspect Oriented Programming (AOP), and weaving the monitoring code into the execution process, which is commonly defined in BPEL [16], [17]. The advantages of this solution are a result of those of AOP, which isolates the monitoring code from the business logic as an aspect, providing low coupling and the ability to add/modify the monitoring rules without affecting the core code of the service. However, in the context of deploying the service over cloud infrastructures, changes over the monitoring rules would require dynamic weaving processes on runtime, which might be somehow difficult if the cloud does not provide the required artifacts for inserting these directives on the execution chain of the service engine. For instance, Zhou et al. [18] make usage of Model-Driven techniques to automatically generate monitoring code for Axis. As advantage, this solution seems to be more efficient than the previous one since there is no weaving process. However, this approach depends on the technology used for service deployment, in this case the engine, where the service is installed.

On the other hand, other proposals use a proxy that intercepts the messages to add monitoring capabilities to the system without the need to be so intrusive into the service or its engine and hence, being independent of the technologies chosen in the implementation of the services [19], [20]. In this case, the same monitoring tool can be used for all kind of services deployed in a cloud. Its main drawback is that if the architecture is not properly built, the proxy can generate a bottleneck affecting negatively the response time of the monitored services.

III. INTEGRATED MONITORING APPROACH FOR SERVICE PROVISIONING IN CLOUDS

Figure 1 shows the architecture of the Integrated Monitoring Approach for Seamless Service Provisioning (IMA4SSP). The figure reveals the interfaces of our components and their relations with the currently available IaaS systems. Our solution offers interoperable access to a federated cloud environment through the interface of the "meta-broker" component. This component is capable to decide
between the use of various “cloud brokers” based on metrics gathered from our “service monitoring” subsystem. Cloud brokers extend the current IaaS functionality by analyzing and dispatching service requests. Based on service demand patterns, they also use the “service deployment” component to deploy or decommission the requested services as virtual machines in specific IaaS systems.

We restrict our solution to support standard and stateless web services described by WSDLs [11]. In this architecture users are able to execute services deployed on cloud infrastructures transparently, in an automated way. The “Generic Service Registry” (GSR – see Figure 1) contains information on these services (including WSDLs and their virtual machine images or virtual appliances – VA). When a service is deployed on a new host, the service deployment component registers its new endpoint to the GSR. Upon decommissioning, these endpoint registrations are removed from the registry. During operation, the SALMon [14] monitoring subsystem allows the components in IMA4SSP to order regular testing on the deployed services according to pre-defined metrics based on the service availability data from the registry.

In our system, users send service calls as request submissions to the Meta-Brokering component (later realized by Generic Meta-Broker Service – GMBS). “Federated call submissions” specify the requested service with a WSDL, the operation to be called, and its possible input parameters. The GMBS checks if the service is registered to the GSR, and if so, it selects a suitable CloudBroker (CB) for further submission, otherwise rejects the request. Based on service usage patterns (e.g. average service response time, call frequency) the GMBS orders the monitoring of the deployed service from SALMon. The monitoring results allow sophisticated matchmaking algorithms based on static data gathered from the GSR and on dynamic information of special metrics (referred as “query cloud metrics” in Figure 1) gathered by SALMon and the cloud brokers. GMBS forms a cloud federation by enabling the autonomous management of the interconnected cloud infrastructures through cloud brokers.

CloudBrokers are dedicated to specific IaaS systems and offer a queue for incoming service calls. They also manage one virtual machine queue for each virtual appliance. Virtual machine queues represent the resources that currently can serve a specific service call. VM queues allow CBs to schedule members of the incoming service queue to specific virtual machines (“Call ⇔ VM Association”). The main goal of CB is to “manage the virtual machine queues” (instantiate and destruct them using service deployment – see Figure 1) according to their respective service demand. The default
virtual machine scheduling is based on the currently available requests in the incoming service queue, their historical execution times, and the number of running VMs. The secondary task of CB involves the dynamic creation and destruction of the various queues. In the following subsections we detail the main components of the architecture.

A. Meta-brokering approach for interoperating clouds

As we already mentioned in the beginning of this section, brokering takes place at two levels in this architecture: the service call is first submitted to a meta-brokering component implemented and names as the Generic Meta-Broker Service (GMBS – which is a revised and extended version of the Grid Meta-Broker Service described in [12]), where a high-level decision is made to which cloud infrastructure the call should be forwarded. Then the service call is placed in the queue of the selected cloud broker, where a lower level brokering is carried out to select the VM that performs the actual service execution.

Now, let us turn our attention to the role of GMBS. This meta-brokering service has five major components. The Meta-Broker Core is responsible for managing the interaction with the other components and handling user interactions. The MatchMaker component performs the scheduling of the calls by selecting a suitable broker. This decision making is based on aggregated static and dynamic data stored by the Information Collector (IC) component in a local database. The Information System (IS) Agent is implemented as a listener service of GMBS, and it is responsible for regularly updating static information from the GSR repository on service availability, dynamic information on service and cloud reliability provided by SALMon (further discussed in Section III-C), and aggregated dynamic information collected from the CloudBrokers (CB) including average VA deployment- and service execution time. The Invoker component forwards the service call to the selected CB and receives the service response.

Each CB is described by an XML-based Broker Property Description Language (BPDL) document containing basic broker properties (e.g. name), and the gathered dynamic properties. The scheduling-related attributes are typically stored in the PerformanceMetrics field of BPDL. More information on this document format can be read in [12]. Namely, the following data are stored in the BPDL of each CB:

- Static availability information on specific virtual appliances in native repositories collected from the GSR;
- average VA deployment time and average service execution time for each VA queried from the cloud brokers;
- and dynamic reliability information expressed by metrics collected from SALMon.

The scheduling process first filters the CBs by checking VA availability in the native cloud repository, then a rank is calculated for each broker based on the collected dynamic data. Finally, the CB with the highest rank is selected for forwarding the service call.

B. Cloud brokering and automatic service deployment

The CloudBroker, which is an extended version of the system described in [13], handles and dispatches service calls to resources and performs resource management within a single IaaS system. It dynamically creates and destroys virtual machines and VM queues of different virtual appliances. Virtual machine creation is supported in the GSR by storing additional static requirements (e.g. its minimum disk, CPU or memory requirements) about each appliance’s future instances.

A VM queue lists resources capable of handling specific service calls, thus instances of a specific VA. New resource requests are inserted to the queue of the appropriate VA, while the need for resource destruction is indicated by the shortening of the queue. Resource entries are managed by the VM Handler that is designed to interact with the public interface of a specific IaaS system. It translates queue changes as VM creation and destruction requests to the IaaS system.

The service call queue stores incoming service requests and a reference in the GSR to a VA for each request. There is a single service call queue in each CloudBroker, while there are many VM queues. Dynamic requirements for the VA may be specified with the service call: additional resources (CPU, memory and disk), and an UUID to identify service calls originating from the same requestor. If dynamic requirements are present, then the CloudBroker creates a new VM queue for them and starts the newly requested VM. Most IaaS systems offer predefined classes of VMs (CPU, memory and disk capacity) not adjustable by the user, therefore the CloudBroker selects the VM class that offers the requested extra resources. This may lead to allocating excess resources in some cases (e.g. the VM class that meets the extra CPU requirement offers twice the requested memory). The CloudBroker also schedules service call requests to VM’s and manages the VM lifecycle. If a service call cannot be associated to any VM, the CloudBroker may decide to start a new VM to serve the request. The VM creation and destruction decisions are based on the following:

- The number of running VM’s available to handle the service call;
- the number of waiting service calls for the VA in the service call queue;
- the average execution time of service calls;
- the average deployment time of VAs;
- and SLA constraints (e.g. total budget, deadline);
- the billing period of the IaaS system.

If a destruction is needed, shutdown is performed shortly before the end of the billing period with regard to the average
decommission time in the IaaS.

IaaS systems require virtual appliances to be stored in their native repositories, because only the previously stored appliances are usable to instantiate virtual machines. The architecture organizes the distribution of user created appliances with the help of the Automatic Service Deployment (ASD – [23]) component. To optimize service executions in highly dynamic service environments, the system organizes virtual appliance distribution by automatically decomposing and replicating appliances. To support the rebuilding of decomposed VAs, the ASD requires appliances to embed minimal manageable virtual appliances (MMVA). These special appliances meet the following properties:

- Provide content management interfaces to add, configure and remove new appliance parts;
- Offer monitoring interfaces to analyze the current state of its instances (e.g. provide access to their CPU load, free disk space and network usage);
- And, it is optimally sized: only those files present in the appliance that are required to offer the previously two properties.

As a result, the ASD only replicates MMVAs to native repositories. Then, the VM Handler controls virtual appliance rebuilding using minimal manageable virtual appliances. Consequently, the VM Handler applies the following strategy when it instantiates a virtual appliance that is not available in the native repository. First, it instantiates a virtual appliance that is not available in the native repository. First, it instantiates the MMVA. Then, the it requests the MMVA’s content management interfaces to download the appliance parts – not present in the native repository – from the GSR. Therefore, the appliance is rebuilt in the virtual machine instantiated for the MMVA. Finally, the VM is ready to serve the scheduled requests from the service call queue.

C. Enhanced service health monitoring with SALMon

SALMon [14] is a service monitoring framework that has been integrated into our proposed architecture in order to gather reliability information on the managed IaaS clouds. It is focused on monitoring the QoS of software services, and is able to evaluate them accordingly to stated conditions and notify the results to the interested parties, which in this case is the Information System (IS) agent from the GMBS.

One of the main characteristics of SALMon is that it combines both passive monitoring and testing approaches, being able to configure each method accordingly to the preferences of the user. In our integrated monitoring solution, SALMon is used for testing purposes, in order to gather the QoS of the constituent services deployed in the cloud. This approach consists of periodically invoking a set of methods of the target service and calculate the QoS over the obtained results. Another important characteristic is that the architecture is able to support any kind of service technology (e.g. SOAP-based web services, RESTFul, etc.), which in the context of the heterogeneity of the cloud is an important aspect to address. The framework has been implemented as a Service-Based Application itself, hence it enables an easy integration with other frameworks. SALMon provides the following two services: the Monitor, responsible to retrieve the value of the required quality metrics of the services; and the Analyzer, which is in charge of the evaluation of conditions over these metrics. These services have the required capabilities in order to monitor services running at different cloud infrastructure providers.
In our IMA4SSP solution, only the monitor service of SALMon is used in order to provide run-time values of the dynamic QoS. To offer the required support for any kind of service, the monitor manages several measure instruments. A Measure Instrument is a component that implements the logic needed in order to obtain the value of a concrete basic quality metric (e.g., Current Response Time, Current Availability, Accuracy of a service or operation). Derived quality metrics are calculated from the set of basic quality metrics retrieved from the measure instruments using an aggregator function in a defined time interval (maximum, minimum, average). Since measure instruments are the core components that actually retrieve the values of the basic metrics, these components are technologically dependent on the kind of service they are monitoring. In this sense, the Monitor service stays above the technological details, and just creates the different measure instruments to obtain the QoS.

The architecture of SALMon is depicted in Figure 2. As stated, the main artifacts are the Analyzer and the Monitor Service. The Analyzer service makes usage of the Monitor service to obtain the QoS and evaluate the satisfaction of conditions, whereas the Monitor creates and manages several Measure Instruments to actually gather the QoS. In the passive monitoring approach, instead of invoking the services directly, the user would invoke a proxy that has these Measure Instruments deployed. In the testing approach, the Monitor creates and manages the Tester component, which is responsible to invoke the service periodically. These invocations are performed through the same proxy and using hence the same measure instruments to retrieve the QoS.

IV. SIMPLIFIED USAGE SCENARIO OF IMA4SSP

As we discussed in the previous subsection, SALMon is capable of monitoring services running at various cloud infrastructure providers. The monitoring target can be defined using specific monitoring metrics. In order to exemplify the operation of our proposed IMA4SSP solution, we describe a simplified usage scenario shown in Figure 3 that we have set up as a proof of concept installation. In this scenario SALMon uses a special test service called Minimal Metric Monitoring Service (M3S) for monitoring cloud reliability, instead of monitoring different service methods stored in the Generic Service Registry. This reference M3S test service has two methods representing three monitoring metrics:

1) Availability: a generalized ping test (e.g. getting the WSDL of the test service) this shows if the service is up and running;
2) Computational capability: measured by a compute method that performs a 5 minute-operation (the result is normalized compared to a reference hardware setup) the response time of this method represents the computational speed of the cloud;
3) Data transfer capability: measured by a transfer method that uploads and downloads a 10 MB file to a predefined public storage location the response time of this method shows the transfer speed of the cloud.

This monitoring test service is prepared in advance, and pre-deployed on the managed IaaS providers. Meanwhile SALMon is configured to monitor the methods of the deployed M3S test services, and the monitoring test cases are set in a way that minimum, maximum, average and latest metric values can be gathered and fetched by GMBS. In a real world private cloud infrastructure we experienced that service methods of the deployed VAs are not always reachable from outside the cloud (though some providers make it available on request). Therefore we separated the monitoring and data management components of SALMon, and created a VM from the monitoring part and a public service from the data management part. For seamless operation, we place the SALMon VM inside the cloud and perform the interactions with the M3S there.

As shown in Figure 3, during operation SALMon performs the monitoring of the M3S methods continuously (Step 1) in an IaaS cloud, and reports the monitored metric values to a central database (Step 2). The IS Agent of GMBS regularly gets the monitored values and updates them in the appropriate BPDV fields of the responsible CloudBroker (Step 3).

Since keeping the monitoring VMs in the cloud can be costly, we have extended the IS Agent component of GMBS to initiate the deployment and decommission of these VMs in order to minimize monitoring costs. The monitored metric values reported to the DDBB have timestamps, therefore these data become outdated after a predefined time interval. When the IS Agent finds that the retrieved metric value of a cloud is outdated, it contacts the VM Handler part of the
responsible CloudBroker, and initiates a M3S VM than a SALMon VM deployment, and calls the appropriate method of the deployed VM to start monitoring. When metric values with new timestamps are read from the DDBB, the IS Agent contacts the VM Handler again, to decommission the monitoring VMs.

V. CONCLUSION

In this paper, we have presented an architecture that offered federated cloud management and utilized a sophisticated service monitoring approach to evaluate basic cloud health status. The architecture uses the Generic Meta-Broker Service as the entry point for the users of the cloud federation. The GMBS service decides the most suitable cloud to perform the service requests of the user by investigating the current state of the clouds according to the Generic Service Registry and the health metrics collected by the SALMon service monitoring subsystem. We also presented the concept of the CloudBroker that is capable of handling service requests and managing virtual machines within a single IaaS cloud system. Finally, we discussed a simplified scenario for exemplifying the operation of our proposed solution using a minimal metric monitoring service.

Our future work targets the evaluation of the IMA4SSP architecture with real user scenarios including ordinary services deployed at different cloud providers. If ordinary services are supported, the architecture should also take into consideration the possible expenses of monitoring. Therefore, we plan to investigate approaches for metric collection for services with long call processing times. Finally, health metrics on ordinary services will also enable better decisions in the lower level components of the architecture. Consequently, we will also explore the integration options of the monitoring system to the CloudBroker and Automatic Service Deployment components of the architecture.

ACKNOWLEDGMENT

The research leading to these results has received funding from the European Community’s Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube), and from the SCI-BUS FP7 project under grant agreement 283481.

REFERENCES


