A Scalable Approach to QoS-Aware Self-adaption in Service-Oriented Architectures

Valeria Cardellini[1], Emiliano Casalicchio[1], Vincenzo Grassi[1], Francesco Lo Presti[1], and Raffaela Mirandola[2]

[1] Università di Roma "Tor Vergata", Viale del Politecnico 1, 00133 Roma, Italy
{cardellini,casalicchio}@ing.uniroma2.it,
{vgrassi,lopresti}@info.uniroma2.it
[2] Politecnico di Milano, Piazza Leonardo Da Vinci 32, 20133 Milano, Italy
mirandola@elet.polimi.it

Abstract. In this paper we consider a provider that offers a SOA application implemented as a composite service to several users with different Qos requirements. For such a system, we present a scalable framework to the QoS-aware self-adaptation based on a two layer reference architecture. The first layer addresses the adaptation at the provisioning level: operating at a slower time scale, its role is to identify the set of candidate services to implement the system functionality at the required user QoS. The second layer addresses the adaptation at the service selection level: operating on a faster time scale, its role is to determine at running time the actual services which are bound to each user request while meeting both provider and user QoS. We formulate the adaptation strategy of both layers as suitable optimization problems which can be efficiently solved using standard techniques. Numerical experiments show the effectiveness of the proposed approach.

Keywords: Service-oriented architecture, self-adaptation, quality of service.

1 Introduction

The today increasingly complex software systems operating in a dynamic operational environment ask for management policies able to deal intelligently and autonomously with problems and tasks. Besides, the way software systems are developed is more and more based on the Service Oriented Architecture (SOA) paradigm, which encourages the construction of new applications through the identification, selection, and composition of network-accessible services offered by loosely coupled independent providers. In a "service market", these different providers may offer different implementations of the same functionality (we refer to the former as *concrete services* and the latter as *abstract service*). These competing services are differentiated by their quality of service (QoS) and cost attributes, thus allowing a prospective user to choose the services that best suit

N. Bartolini et al. (Eds.): QShine/AAA-IDEA 2009, LNICST 22, pp. 431–447, 2009.

his/her needs. The QoS contracted by users and providers must meet certain respective obligations and performance expectations which the parties agree upon in the *Service Level Agreement* (SLA) contracts.

The fulfillment of global QoS requirements, such as the application response time and availability, by a SOA system offering a composite application is a challenging task, because it requires the system to take complex decisions within short time periods, in an operational environment characterized by a dynamic and unpredictable nature. A promising way to manage effectively this task is to make the SOA system able to self-adapt at runtime in response to changes in its operational environment, by autonomously reconfiguring itself through a closed-loop approach with feedback [1]. In this way, the system can timely react to environment changes (concerning for example available resources, type and amount of user requests), in such a way to fulfill its requirements at runtime.

Several methodologies have been already proposed for QoS-aware SOA systems able to dynamically self-adapt in order to fulfill non-functional or functional requirements (e.g., [2, 3, 4, 5, 6, 7]). Most of the proposed methodologies address this issue as a *service selection* problem: given the set of abstract services needed to compose a new added value service, the goal is to identify for each abstract service a corresponding concrete service, selecting it from a set of candidates (e.g., [2, 3, 4, 6, 7]). When the operating conditions change (e.g., a selected concrete service is no longer available, or its delivered QoS has changed, or the user QoS requirements have changed), a new selection can be calculated and the abstract services which compose the offered SOA application are dynamically bound to a new set of concrete services.

In this paper, we follow the service selection approach towards self-adaptive SOA systems, but, differently from previous work in the area, we propose a two-layer adaptation strategy carried out by the service broker that offers the SOA application. In our approach, adaptation decisions occur at different time scales in order to exploit the optimal provisioning of the component services and maintain QoS guarantees to various classes of users. Specifically, the first layer operates at a slow time scale and addresses the adaptation task at the *service provisioning* level. Its role is to identify, from a given set of functionally equivalent candidate concrete services, the actual pool of concrete services that will be used to implement the component functionalities such that the aggregated QoS values satisfy the users' end-to-end QoS requirements and, at the same time, the service broker's utility function is maximized. The first layer also determines how much the identified concrete services are being utilized (i.e., it reserves the resource capacities). The solution provided by the first-layer is used on a long term for planning and defining SLAs with the service providers. The second layer operates at a fast time scale and addresses the adaptation at the *service selection* level. Its role is to determine, from the pool identified by the first layer, the actual concrete services which are bound to each incoming user request while meeting both provider and user QoS requirements.

We formulate the adaptation strategies of both layers as suitable optimization problems which can be solved using standard techniques. Specifically, the

second-layer optimization problem is formulated as a Linear Programming (LP) problem and is suitable to be solved at runtime because of its efficiency. On the other hand, the first-layer optimization problem is a Mixed Integer Linear Programming (MILP) one and is known to be NP-hard. However, its solution is required on a larger time scale than the second-layer problem: we estimate that, in a real scenario, the times at which the solution of two problems occurs differ by at least two orders of magnitude. Therefore, our two-layer approach can be deployed directly in a broker-based architecture operating in a highly variable SOA environment, where the scalability and effectiveness in replying to the users are important factors. To the best of our knowledge, this paper represents in the SOA environment the first proposal of a two-layer adaptation strategy operating at different time scales in order to manage dynamically the service provisioning and selection issues.

There is a significant body of research about how to realize the self-adaptation of systems to let them cope with a dynamic operational environment [1]. Existing proposals about how to architect a self-adaptive system share the common view that self-adaptation is achieved by means of a monitor-analyze-act cycle [8]: the system collects relevant events concerning itself and its context, analyzes them to decide suitable adaptation actions, and then act to execute the adaptation decisions. The main classes of approaches proposed in the SOA research community to tackle the dynamic adaptation of a SOA system include QoS-based service selection and workflow recostructuring.

In the first case, as already outlined above, new service components are selected to deal with changes in the operating scenario [2, 6, 7, 3, 4, 9, 10, 11]. Some of the works dealing with this general problem propose heuristics (e.g., [9, 10] or genetic algorithms in [3]) to determine the adaptation actions. Others propose exact algorithms to this end: [6] formulates a multi-dimension multi-choice 0-1 knapsack problem as well as a multi-constraint optimal path problem; [7] presents a global planning approach to select an optimal execution plan by means of integer programming; in [2,10,11] the adaptation actions are selected through mixed integer programming. A general drawback of most proposals for dynamic adaptation based on service selection is that they pay little attention to efficiency and scalability. The approaches that we presented in [4, 12] and adopt also in this paper address these issues by performing the optimization on a per-flow rather than per-request basis. In these approaches, the solution of the optimization problem holds for all the requests in a flow, and is recalculated only when some significant event occurs (e.g., a change in the availability or the QoS values of the selected concrete services). Moreover, the optimization problem is solved taking into account simultaneously the flows of requests generated by multiple classes of users, with possibly different QoS constraints.

The second class includes research efforts that have instead considered *workflow restructuring*, exploiting the inherent redundancy of the SOA environment to meet the QoS (basically, dependability) requirements [13, 10, 14]. In [12] we proposed a methodology that integrates within a unified framework the two classes of approaches by binding each abstract service to a set of functionally

equivalent concrete services, coordinated according to some spatial redundancy pattern. The two-layer approach we present in this paper can be extended by applying the above methodology.

The rest of the paper is organized as follows. In Section 2 we present the system architecture. In Section 3 we describe the composite service model we refer to, the type of SLA contracts used for the service users and providers, and define the goals of the two optimization problems. In Section 4 we present the mathematical formulation of the optimization problems used in the two-layer adaptation approach. In Section 5 we present the results of some numerical experiments. Finally, we draw some conclusions and give hints for future work in Section 6.

2 System Architecture

The *service broker* acts as a third-party intermediary between service users and providers, performing a role of provider towards the users and being in turn a requestor to the providers of the concrete services. It advertises and offers the composite service with a range of service classes wich imply different QoS levels and monetary prices. To carry out its task, the broker architecture is structured around the following components, as illustrated in Figure 1: the *Workflow Engine*, the *Composition Manager*, the *SLA-P Manager*, the *Selection Manager*, the *SLA Monitor*, the *Optimization Engine*, the *Provisioning Manager*, and the *Data Access Library*. Our envisioned architecture is inspired by existing implementation of frameworks for Web services QoS brokering, e.g., [15, 16].

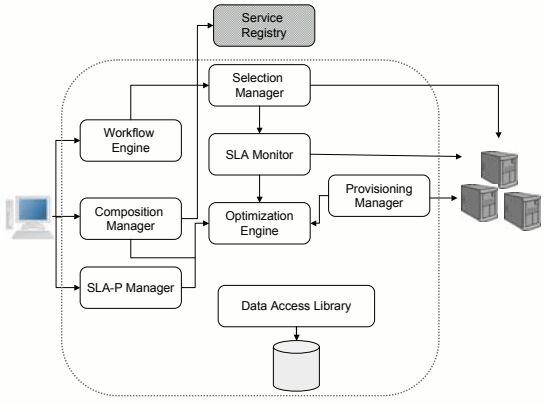

Fig. 1. Broker architecture

The respective tasks of the broker architecture components can be summarized as follows. The main functions of the Composition Manager are the specification of the business process and the discovery of all the service providers

offering functionally equivalent service implementations. The Workflow Engine is the software platform executing the BPEL business process (e.g., ActiveBPEL or ApacheODE) and represents the user front-end for the composite service provisioning. The Workflow Engine interacts with the Selection Manager to allow the invocation of the component services. Indeed, for each service invocation, the Selection Manager binds dynamically the request to the real endpoint that represents the concrete service. The latter is identified through the solution of the service selection optimization problem. Therefore, in the envisioned architecture the Selection Manager is in charge of the adaptation actions of the service selection layer. It also keeps up to date information about the composite service usage profile. Together, the Workflow Engine and the Selection Manager are responsible for managing the user requests flow, once the user has been admitted to the system with an established SLA.

The main task of the SLA-P Manager is the SLA negotiation with the users of the composite service. It is also in charge of the admission control and rate limiting functionalities. The first allows to determine whether a new user can be accepted, given the associated SLA and without violating already existing SLAs. To this end, the SLA-P Manager may trigger a new solution of the service selection problem. The rate limiting functionality is motivated by the need to limit the requests submitted to the composite service to the maximum arrival rate agreed in the SLAs. As a control mechanism for rate limiting, our broker architecture employs the classic token bucket [17]. This mechanism permits burstiness, but bounds it. The SLA-P Manager maintains a separate token bucket for each user, and each token in a bucket enables a single request to the composite service. Upon arrival, a request for the composite service will be sent out with the token bucket of the corresponding user decreased by one, provided there are available tokes for the request. Otherwise, the request is enqueued for subsequent transmission until tokens have been accumulated in the bucket. A SOA middleware architecture that employs the token-bucket algorithm for admission control is presented in [18].

The SLA Monitor collects information about the QoS level perceived by the users and offered by the providers of the used component services. Furthermore, the SLA Monitor signals whether there is some variation in the pool of service instances currently available for a given abstract service (i.e., it notifies if some service goes down/is unavailable).

The Optimization Engine is the broker component that executes the two adaptation algorithms (i.e, service provisioning and service selection), passing to them the updated instance of the optimization problem with the new values of the parameters. The calculated solutions provide indications about the adaptation actions that must be performed to identify the pool of resources (i.e., the concrete services) and to optimize their use with respect to the utility criterion of the broker as well as to the QoS levels agreed with the users.

The Provisioning Manager is in charge of organizing the service provisioning policy that makes the broker able to meet its utility objective, that is it manages the first-layer adaptation actions in the proposed system architecture. Once the

Optimization Engine has identified through the solution of the service provisioning problem the new subset of component services to be used, the Provisioning Manager negotiates the SLAs with their respective providers.

Finally, the Data Access Library is used by all the modules to access the model parameters of the composite service operations and environment (among which the abstract and the corresponding concrete services with their QoS values, and the values determined by the solution of the optimization problems, as discussed in Section 3). In Figure 1 the lines connecting the components to the Data Access Library have been omitted for clarity.

The SLA-P Manager, SLA Monitor, Selection Manager, and Composition Manager modules are collectively responsible for monitoring, detecting and deciding about the activation of a new adaptation strategy. When one of these modules detects a significant variation of the system model parameters, it signals the event to the Optimization Engine, which executes a new instance of the service provisioning or selection optimization problem and determines a new solution (in case it exists). Specifically, in our two-layer adaptation strategy the triggering to the Optimization Engine can occur either periodically or aperiodically and at different time scales. Given the efficiency of the service selection problem (formulated as LP problem in Section 4), it is suitable for being executed frequently in such a way to react quickly to detected changes. Its solution may be caused by a change in the effective request arrival rates measured by the SLA-P Manager at the exit of the token buckets, by a variation in the QoS levels determined by the SLA Monitor, and by a change in the composite service usage measured by the Selection Manager. If existing, the calculated solution provides indications to the Selection Manager on how to use the pool of available concrete services.

Since the first-layer service provisioning is a time-consuming reaction to detected changes (formulated as MILP problem in Section 4), it has to be invoked moderately and on a larger time scale. Its activation may be either periodic or aperiodic and it corresponds to modifications in the broker utility, in the arrival/departure of users, and also some change in the available resources (i.e., new concrete services identified by the Composition Manager, unreachability of some used concrete service determined by the SLA Monitor). The first-layer solution can be also triggered as a consequence of a second-layer optimization problem withouth a feasible solution. We postpone to a future paper the study of the possible activation schemes of the two layers and their performance impact on the system.

3 System Model

3.1 Composite Service Model

The SOA system managed by the broker offers a composite service, that is, a composition of multiple services in one logical unit in order to accomplish a complex task. We assume that the composite service structure is defined using BPEL [19], the de-facto standard for service workflows specification languages.

Here, without lack of generality, we restrict on the BPEL structured style of modeling, and consider workflows which include, besides the primitive `invoke` activity, all the different types of structured activities: `sequence`, `switch`, `while`, `pick`, and `flow`. Figure 2 shows an example of a BPEL workflow described as a UML2 activity diagram. With the exception of the `pick` construct, this example encompasses all the structured activities listed above.

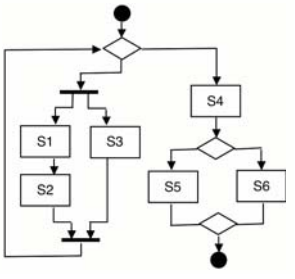

Fig. 2. An example of BPEL workflow

The business process for the composite service defines a set of abstract services $\mathcal{V}$. We denote by $S_i \in \mathcal{V}$ each abstract service (i.e., a functionality needed to compose a new added value service), and by $s_{ij} \in P_i$ a specific concrete service, where P_i is the set of functionally equivalent concrete services that have been identified by the Composition Manager as candidates to implement S_i. For each abstract service S_i, we also denote by $I_i \subseteq P_i$ the pool of concrete services determined by the solution of the service provisioning problem and used at runtime for offering the composite service.

The overall QoS of a composite service implementation depends not only on the QoS of the concrete services that have been bound to the abstract services and on the way they are orchestrated, but also on the usage profile of those services for each given class of users: a rarely invoked service has obviously a smaller impact on the overall QoS than a frequently invoked one, and different classes of users may invoke the same services with different frequencies. To embody this knowledge in our model, we model the usage profile of each service class $k \in K$ (where K denotes the set of the considered classes), by annotating each abstract service S_i with the average number of times V_i^k it is invoked by k-class requests addressed to the composite service. The Selection Manager performs a monitoring activity to keep up to date the V_i^k values.

3.2 SLA Model

Since the broker offering the composite service plays both the provider and requester roles, it is involved in two types of SLA, corresponding to these two roles: we call them SLA-P (provider role) and SLA-R (requester role). In general,

a SLA may include a large set of parameters, referring to different kind of QoS attributes (e.g., response time, availability, and reputation). In this paper, we restrict our attention to the following three attributes (but other attributes could be easily added to our framework without changing the methodology):

- *response time*: the interval of time elapsed from the service invocation to its completion;
- *availability*: the probability that the service completes its task when invoked;
- *cost*: the price charged for the service invocation.

The SLA-R contracted by the broker with the provider of the concrete service $s_{ij} \in I_i$ is specified by an instance of the tuple $\langle r_{ij}, a_{ij}, L_{ij}, c_{ij}, d_{ij} \rangle$, where r_{ij} and a_{ij} are the average response time and logarithm of availability of s_{ij}. In our SLA model, we assume that the price paid by the broker to the provider of s_{ij} is given by the sum of a fixed cost c_{ij} plus a variable cost, which is linearly proportional through d_{ij} to the amount of service capacity L_{ij} reserved by the broker. By solving the service provisioning optimization problem, the broker identifies the pool of concrete services with each of whom it negotiates an active SLA-R. The set of all the active SLAs-R defines the constraints within which the broker can organize the second stage of the adaptation strategy carried out through the service selection.

We denote by K the set of QoS classes offered by the broker. Each class $k \in K$ is characterized in terms of bounds on the expected response time $R^k_{\max}$ and availability $A^k_{\min}$ as well as the service costs: a fixed component c^k and a variable component which is proportional to a rate d^k per unit per request per unit of time. A user u requesting a given class of service k has to define the maximum load L^k_u it will generate. The SLA-P established by the broker with the requestor u for the QoS class k is therefore a tuple $\langle R^k_{\max}, A^k_{\min}, L^k_u, c^k, d^k \rangle$.

As discussed in Section 2, our broker architecture implements the token bucket mechanism for request rate limiting. The bucket of each user is refilled at rate L^k_u, until the bucket reaches its capacity. We denote by λ^k_u the effective arrival rate processed by the system.

3.3 Service Selection Model

The goal of the Selection Manager is to determine, for each QoS class, the concrete service s_{ij} that must be used to fulfill a request for the abstract service $S_i \in \mathcal{V}$. The selection can be modelled by associating with each S_i a vector $\boldsymbol{x}_i = (\boldsymbol{x}^1_i, ..., \boldsymbol{x}^{|K|}_i)$, where $\boldsymbol{x}^k_i = [x^k_{ij}]$ and $s_{ij} \in I_i$. Each entry x^k_{ij} of $\boldsymbol{x}^k_i$ denotes the probability that the class-k request will be bound to the concrete service s_{ij}. With this model, we assume that the Selection Manager can probabilistically bind to different concrete services the requests (belonging to a same QoS class k) for an abstract service S_i. The deterministic selection of a single concrete service corresponds to the case $x^k_{ij} = 1$ for a given $s_{ij} \in I_i$.

As an example, consider the case $I_i = \{s_{i1}, s_{i2}, s_{i3}\}$ and assume that the adaptation policy $\boldsymbol{x}_i^k$ for a given class k specifies the following values: $x_{i1}^k = x_{i2}^k = 0.3$, $x_{i3}^k = 0.4$. This strategy implies that 30% of the class-k requests for service S_i are bound to service s_{i1}, 30% are bound to service s_{i2} while the remaining 40% are bound to s_{i3}. From this example we can see that, to get some overall QoS objective for a given class flow of requests, the Seleciton Manager may switch different requests to different providers (using $\boldsymbol{x}_i^k$ to drive the switch).

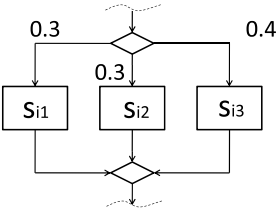

Fig. 3. Flow partitioning among different providers

The Selection Manager determines the values of the x_{ij}^k by invoking the Optimization Engine. The goal is to determine an overall selection strategy $\boldsymbol{x} = (\boldsymbol{x}_1, ..., \boldsymbol{x}_{|\mathcal{V}|})$ which maximizes a suitable QoS objective function $F(\boldsymbol{x})$. The optimization problem takes the following general form:

find $\boldsymbol{x}$ **which maximizes** $F(\boldsymbol{x})$
subject to: Class-k QoS due to strategy $\boldsymbol{x}^k$ does not violate class-k SLA, $k \in K$;
the load induced by strategy $\boldsymbol{x}$ on provider s_{ij} does not
exceed L_{ij}, $s_{ij} \in I_i$, $S_i \in \mathcal{V}$.

In our setting, the optimization problem takes the form of a LP problem. The details will be spelled out in Section 4.

3.4 Service Provisioning Model

The goal of the Provisioning Manager is to determine from the set of candidate concrete services the subset that will be used to implement the system functionalities and the capacity to be reserved on each selected concrete service. We model this selection with two vectors. The first vector is $\boldsymbol{y} = [y_{ij}]_{s_{ij} \in P_i}$, $i \in \mathcal{V}$, $y_{ij} \in \{0, 1\}$: $y_{ij} = 1$ if service $s_{ij} \in P_i$ is included in the pool I_i; otherwise, $y_{ij} = 0$. We also define the vector $\boldsymbol{L} = [L_{ij}]_{s_{ij} \in P_i}$, $i \in \mathcal{V}$. L_{ij} is the service capacity the application reserves with the concrete service s_{ij}. $L_{ij} = 0$ if $y_{ij} = 0$ and $L_{ij} \geq 0$ if $y_{ij} = 1$.

The Provisioning Manager determines the values of the y_{ij} and L_{ij} by invoking the Optimization Engine. The goal is to determine the service pool and capacity which minimize a suitable cost function $C(\boldsymbol{y}, \boldsymbol{L})$. The optimization problem takes the following general form which we will detail in the next section:

find $(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{L})$ **which minimizes** $C(\boldsymbol{y}, \boldsymbol{L})$
subject to: Class-k QoS due to strategy $\boldsymbol{x}^k$ does not violate class-k SLA, $k \in K$;
the service pool $\boldsymbol{y}$ and capacity $\boldsymbol{L}$ are such that the load
induced by strategy $\boldsymbol{x}$ on provider s_{ij} does not exceed
L_{ij}, $s_{ij} \in I_i$, $S_i \in \mathcal{V}$ for any possible class request arrival rate.

To understand the role of $\boldsymbol{x}$ in this problem observe that for $(\boldsymbol{y}, \boldsymbol{L})$ to be feasible there must be at least one redirection strategy $\boldsymbol{x}$ such that: 1) the load induced by $\boldsymbol{x}$ on any provider does not exceed the capacity reserved on that provider for any class request arrival rate; and 2) the QoS of each class is not violated. On the other hand, we are not interested in optimizing (or even identifying) such strategy as long as one actually exists.

4 Optimization Problems

In this section, we first present how to compute the QoS attributes of the composite service. We then detail the instances of the optimization models we presented in Section 3.

4.1 QoS Metrics

For each class $k \in K$ offered by the broker, the overall QoS attributes are the expected response time R^k and the expected availability A^k. To compute these quantities, let $Z_i^k(\boldsymbol{x})$, $Z \in \{R, A\}$, denote the QoS attribute of the abstract service $S_i \in \mathcal{V}$. We have $Z_i^k(\boldsymbol{x}) = \sum_{s_{ij} \in I_i} x_{ij}^k z_{ij}^k$ where z_{ij}^k, $z \in \{r, a\}$ is the corresponding QoS attribute offered by the concrete service s_{ij} which can implement S_i. We now derive closed form expressions for the QoS attributes of the composite service we will later use in the formulation of the optimization problem.

Availability. The (logarithm of the) availability QoS metric is an additive metric [20]. Therefore, for its expected value we readily obtain

$$A^k(\boldsymbol{x}) = \sum_{i \in \mathcal{V}} V_i^k A_i^k(\boldsymbol{x}) = \sum_{i \in \mathcal{V}} V_i^k \sum_{s_{ij} \in I_i} x_{ij}^k a_{ij}$$

where V_i^k is the expected number of times S_i is invoked for a class-k request.

Response Time. The response time metric is additive only as long as the composite service does not include `flow` structured activities. In such cases, we readily have:

$$R^k(\boldsymbol{x}) = \sum_{i \in \mathcal{V}} V_i^k \sum_{s_{ij} \in I_i} x_{ij}^k r_{ij}. \tag{1}$$

In the general case, instead, we need to account for the fact that the response time of a `flow` activity [19] is given by the largest response time among its component activities. Hence, in the general case, the response time is not additive and (1) does not hold. In this case, we derive an expression for the response time $R^k(\boldsymbol{x})$ by recursively computing the response time of the constituent workflow activities as shown in [4] which we will later use in the actual problem formulation.

4.2 Second-Layer Problem: Service Selection Optimization

In this section we detail the service selection optimization problem. The goal is to determine the variables x_{ij}^k, $i \in \mathcal{V}$, $k \in K$, $s_{ij} \in I_i$ which maximize a suitable QoS function. We assume that the broker wants, in general, to optimize multiple QoS attributes (which can be either mutually independent or possibly conflicting), rather than just a single one, *i.e.*, the response time. We thus consider as objective function $F(\boldsymbol{x})$ an aggregate QoS measure given by a weighted sum of the (normalized) QoS attributes. More precisely, let $Z(\boldsymbol{x}) = \frac{1}{\sum_{k\in K}\lambda^k}\sum_{k\in K}\lambda^k Z^k(\boldsymbol{x})$, where $Z \in \{R, A\}$ is the expected overall response time and availability, respectively, and $\lambda^k = \sum_u \lambda_u^k$ is the instantaneous aggregate flow of class-k requests. We define the objective function as follows:

$$F(\boldsymbol{x}) = w_r \frac{R_{\max} - R(\boldsymbol{x})}{R_{\max} - R_{\min}} + w_a \frac{A(\boldsymbol{x}) - A_{\min}}{A_{\max} - A_{\min}} \tag{2}$$

where $w_r, w_a \geq 0$, $w_r + w_a = 1$, are weights for the different QoS attributes. $R_{\max}$ ($R_{\min}$), and $A_{\max}$ ($A_{\min}$) denote, respectively, the maximum (minimum) value for the overall response time and the (logarithm of) availability. We will describe how to determine these values shortly.

The Optimization Engine task consists in finding the variables x_{ij}^k, $i \in \mathcal{V}$, $k \in K$, $s_{ij} \in I_i$, which solve the following optimization problem:

$$\begin{aligned}
&\textbf{max } F(\boldsymbol{x}) \\
\textbf{subject to: } & R^k(\boldsymbol{x}) \leq R_{\max}^k \quad k \in K && (3)\\
& R_{l'}^k(\boldsymbol{x}) \leq R_l^k(\boldsymbol{x}) \quad l' \in d(l), l \in \mathcal{F}, k \in K && (4)\\
& R_l^k(\boldsymbol{x}) = \sum_{i\in\mathcal{V}, i \prec_{dd} l} \frac{V_i^k}{V_l^k} \sum_{s_{ij}\in I_i} x_{ij}^k r_{ij} + \\
& \qquad + \sum_{h\in\mathcal{F}, h\prec_{dd} l} \frac{V_h^k}{V_l^k} R_h^k(\boldsymbol{x}), l \notin \mathcal{F}, k \in K && (5)\\
& A^k(\boldsymbol{x}) \geq A_{\min}^k \quad k \in K && (6)\\
\sum_{k\in K} & x_{ij}^k V_i^k \lambda^k \leq L_{ij} \quad i \in \mathcal{V}, s_{ij} \in I_i && (7)\\
& x_{ij}^k \geq 0, \quad s_{ij} \in I_i, \sum_{s_{ij}\in I_i} x_{ij}^k = 1 \quad i \in \mathcal{V}, k \in K && (8)
\end{aligned}$$

Equations (3)-(6) are the QoS constraints for each service class on response time and availability, where $R_{\max}^k$ and $A_{\min}^k$ are respectively the maximum response time and the minimum (logarithm of the) availability that characterize the QoS class k. The constraints for the response time take into account the fact the response time of a `flow` activity is given by the largest response time of its component activities. This is reflected in the constraints (4)-(5), where $\mathcal{F}$ denotes the set of `flow` activities in the composite service. Inequalities (4), in particular,

allow us to express the relationship among the response time R_l^s of a `flow` activity and that of its component activities $R_{l'}^s$. For each `flow` activity l, $d(l)$ is the set of top-level activities/services which are nested within l; $i \prec_{dd} l$ means that service i occurs within activity j in the BPEL code and, within j, i does not appear within a `flow` activity (see [4] for details). Equations (7) are the SLA-R constraints and ensure that the application does not exceed the volume of invocations agreed with the service providers. Finally, Equations (8) are the functional constraints.

The maximum and minimum values of the QoS attributes in the objective function (2) are determined as follows. $R_{\max}$ and $A_{\min}$ are simply expressed respectively in terms of $R_{\max}^k$ and $A_{\min}^k$. For example, the maximum response time is given by $R_{\max} = \frac{1}{\sum_{k \in K} \lambda^k} \sum_{k \in K} \lambda^k R_{\max}^k$. Similar expression holds for $A_{\min}$. The values for $R_{\min}$ and $A_{\max}$, instead, are determined by solving a modified optimization problem in which the objective function is the QoS attribute of interest, subject to the constraints (7)-(8).

We observe that the proposed Optimization Engine problem is a Linear Programming problem which can be efficiently solved via standard techniques. The solution thus lends itself to on-line operations.

4.3 First-Layer Problem: Service Provisioning Optimization

We now turn our attention to the Provisioning Manager optimization problem. The goal is to determine the value of the variables y_{ij} and L_{ij}, $s_{ij} \in P_i$, $i \in \mathcal{V}$, which minimize the broker cost function. We consider as objective function $F(\boldsymbol{y}, \boldsymbol{L})$ the following simple cost function:

$$F(\boldsymbol{y}, \boldsymbol{L}) = \sum_{s_{ij} \in P_i, i \in \mathcal{V}} c_{ij} y_{ij} + d_{ij} L_{ij} \tag{9}$$

where c_{ij} represents a fixed/flat cost to be paid for *using* concrete service s_{ij} and d_{ij} is the cost for unit of capacity of service s_{ij}, reserved by the broker.

The Optimization Engine task consists in finding the y_{ij} and L_{ij}, $s_{ij} \in P_i$, $i \in \mathcal{V}$ (and also x_{ij}^k, $s_{ij} \in P_i$, $i \in \mathcal{V}$, $k \in K$) which solve the following optimization problem:

$$\begin{aligned}
&\textbf{min } F(\boldsymbol{y}, \boldsymbol{L}) \\
&\textbf{subject to:} \\
&\quad \text{QoS constraints } (3) - (6) \\
&\sum_{k \in K} x_{ij}^k V_i^k L^k \leq L_{ij} \quad i \in \mathcal{V}, s_{ij} \in P_i, && (11) \\
&\quad L_{ij} \leq M_{ij} y_{ij}, \quad i \in \mathcal{V}, s_{ij} \in P_i && (12) \\
&\quad x_{ij}^k \leq y_{ij} \quad i \in \mathcal{V}, k \in K && (13) \\
&\quad x_{ij}^k \geq 0, s_{ij} \in P_i, \sum_{s_{ij} \in P_i} x_{ij}^k = 1 \quad i \in \mathcal{V}, k \in K && (14) \\
&\quad y_{ij} \in \{0, 1\}, \quad i \in \mathcal{V}, s_{ij} \in P_i && (15)
\end{aligned} \tag{10}$$

The constraints (3)-(6) are the QoS constraints as in the service selection optimization. The constraints (11) are the provider capacity constraints which require that the reserved capacity L_{ij}, $s_{ij} \in P_i$, $i \in \mathcal{V}$ must accommodate any request load for the concrete service s_{ij} (under service selection strategy $\boldsymbol{x}$), where $L^k = \sum_u L^k_u$ denotes the maximum class k request rate. Finally, equations (12)-(14) are the functional constraints. (12) requires that $y_{ij} = 1$ for L_{ij} be greater than 0; similarly, (13) requires $y_{ij} = 1$ for x^k_{ij} be greater than 0. In (11) we also introduce the constant M_{ij} which denotes the maximum capacity that can be reserved on provider s_{ij} (M_{ij} thus captures the finiteness of provider s_{ij} resources).

The proposed optimization problem is a MILP problem. It is known to NP-hard with the complexity being exponential in the number of integer variables, which is $O(max_{i\in\mathcal{V}}|P_i| \times |\mathcal{V}|)$.

5 Numerical Experiments

In this section, we illustrate the behaviour of the proposed two-layer adaptation strategy through the simple abstract workflow of Figure 2. We consider a broker which offers two QoS classes *gold* and *silver*, denoted by the superscript 1 and 2, respectively. Table 1 summarizes the two classes QoS attributes. The gold class guarantees to its users low response times at a high cost, while the silver class offers a cheaper alternative with higher response times. We consider the following values for the the number of service invocations: $V^k_1 = V^k_2 = V^3_k = 1.5$, and $V^k_4 = 1$ for $k = 1, 2$, $V^1_5 = 0.7$, $V^1_6 = 0.3$, and $V^2_5 = V^2_6 = 0.5$. We assume that for each abstract service there are four providers which implements it. The concrete services differ in terms of response time, cost and availability. Table 2 summarizes their system parameters. They have been ordered so that for each abstract service $S_i \in \mathcal{V}$, s_{ij} represents the *better*, albeit more expensive, service, with respect $s_{ij\prime}$, $j\prime > j$. For all services, we assume $M_{ij} = 10$.

We study now the broker behaviour over a period of 1000 time units during which we have a fixed set of users. The associated peak request rate for the two service classes is assumed equal to $(L^1, L^2) = (4, 7)$. We assume that during this period the only meaningful event is the unavailability of service s_{14} from time 400 onward. First we consider the behaviour of the Provisioning Manager. The role of the manager is to identify the optimal - cost wise - set of concrete services to implement the abstract services and the associated capacities. For the given workflow, the solution of the optimization problem is illustrated in Table 3, which reports the set I_i of concrete services selected for each abstract service

Table 1. Composite service class attributes

QoS Class	$R^k_{\max}$	$A^k_{\min}$	c^k	d^k
gold	12	log(0.95)	10	5
silver	20	log(0.9)	6	3

Table 2. Concrete services QoS attributes

s_{ij}	r_{ij}	a_{ij}	c_{ij}	d_{ij}
s_{11}	1	log(0.999)	3	2
s_{12}	1.5	log(0.995)	4	1.5
s_{13}	1.5	log(0.99)	3	1.5
s_{14}	3.5	log(0.98)	2.5	1
s_{21}	2	log(0.999)	4	1.5
s_{22}	4	log(0.99)	2	1.5
s_{23}	1	log(0.99)	4.5	1
s_{24}	5	log(0.95)	1	1

s_{ij}	r_{ij}	a_{ij}	c_{ij}	d_{ij}
s_{31}	1	log(0.999)	4	1.5
s_{32}	1	log(0.99)	2	1.5
s_{33}	2	log(0.99)	4.5	1
s_{34}	3	log(0.99)	1	1
s_{41}	0.5	log(0.999)	0.6	2
s_{42}	1	log(0.995)	0.5	1
s_{43}	1	log(0.99)	0.4	1.5
s_{44}	2	log(0.99)	0.3	1

s_{ij}	r_{ij}	a_{ij}	c_{ij}	d_{ij}
s_{51}	2	log(0.999)	1	2
s_{52}	2	log(0.995)	0.7	1
s_{53}	2.2	log(0.99)	0.5	1.5
s_{54}	3	log(0.99)	0.2	1.5
s_{61}	1.8	log(0.999)	0.5	1.5
s_{62}	2	log(0.995)	0.4	1
s_{63}	2	log(0.99)	0.3	1
s_{64}	4	log(0.99)	0.2	1.5

$S_i \in \mathcal{V}$ and the capacity L_{ij} reserved in each concrete service. A first solution (Table 3 (left)) is first computed at the beginning of the period (for a minimum cost equal to 93.8). The solution guarantees enough resources to sustain peak rate traffic, *i.e.*, $(L^1, L^2) = (4, 7)$ at the required QoS of each class. Observe that since the different abstract services are characterized by different frequencies of invocations, the overall capacity to be reserved differs from service to service, *e.g.*, S_1 requires an overall capacity of 16.5, while S_5 requires only a capacity of 6.3. At time 400, we assume that service s_{14} becomes unavailable. In our example, this forces the Provisioning Manager to execute again the provisioning optimization problem (it is not possible to serve the requests for the abstract service S_1 with the sole concrete service s_{13}) and adjusts the SLA with the providers accordingly. Table 3 (right) shows the new solution where, essentially, the concrete service s_{13} replaces s_{14} and the reserved capacity of some providers are slightly modified.

Table 3. SLA Manager solution. Service pool and reserved capacities.

Service Sets	Reserved Capacity
$I_1 = \{s_{13}, s_{14}\}$	$L_{13} = 6.5$, $L_{14} = 10$
$I_2 = \{s_{23}, s_{24}\}$	$L_{23} = 9.2$, $L_{24} = 7.8$
$I_3 = \{s_{32}, s_{34}\}$	$L_{32} = 10$, $L_{34} = 6.5$
$I_4 = \{s_{42}, s_{44}\}$	$L_{42} = 6.9$, $L_{44} = 4.1$
$I_5 = \{s_{52}\}$	$L_{52} = 6.3$
$I_6 = \{s_{63}\}$	$L_{63} = 4.7$

Service Sets	Reserved Capacity
$I_1 = \{s_{12}, s_{13}\}$	$L_{12} = 8.35$, $L_{13} = 8.15$
$I_2 = \{s_{23}, s_{24}\}$	$L_{23} = 8.93$, $L_{24} = 7.57$
$I_3 = \{s_{32}, s_{34}\}$	$L_{32} = 10$, $L_{34} = 6.5$
$I_4 = \{s_{42}, s_{44}\}$	$L_{42} = 6.36$, $L_{44} = 4.64$
$I_5 = \{s_{52}\}$	$L_{52} = 6.3$
$I_6 = \{s_{63}\}$	$L_{63} = 4.7$

We now turn our attention to the Selection Manager. Differently from the Provisioning Manager, the Selection Manager adaptation role is to determine at running time the actual services to be bound to each user request. To illustrate its behaviour we consider the sample path arrival rates for the two classes shown in Figure 4 (the sample paths have been generated by superposition of several regulated sources, with each source being a two state on-off source). We assume

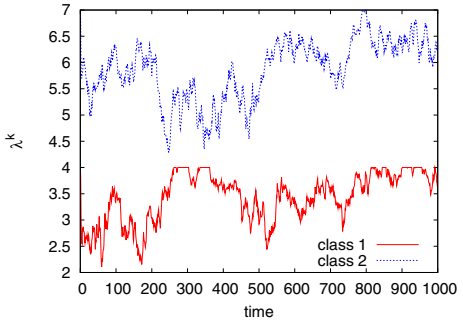

Fig. 4. Sample path arrival rate λ^k

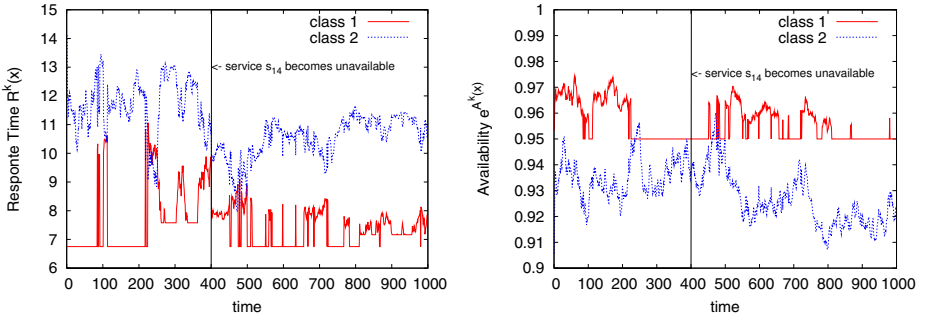

Fig. 5. Selection Manager solution. QoS metrics: response time $R^k(\mathbf{x})$ (left); Availability $e^{A^k(\mathbf{x})}$ (right).

that the Selection Manager uses the measured actual aggregate arrival rates $\lambda^k \leq L^k$, $k = 1, 2$ and solves the service selection optimization problem to settle the vector $\boldsymbol{x}$, according to which randomly determines the concrete service to select. Different values of λ^k, $k = 1, 2$ result into different optimal vectors which in turn yield different QoS metrics.

In Figure 5 we show how the expected composite service QoS metrics vary over time for the two classes under the assumption the Selection Manager minimizes the service response time, *i.e.*, $w_r = 1$. Both service response time and availability vary with the request rates but are always within the performance bound defined by the class SLA metrics. Not surprisingly, users experience better response time and service availability for lower request rates since a large fraction - if not all - of the requests are bound to the best services in the pool. Observe that after t=400, the response time for both service classes improves significantly. This can be explained by observing that the unavailability of service s_{14}, which provides the cheapest - but slowest - service, forces the broker to include in the pool the more expensive, but faster, service s_{12}, which results into overall better response times.

6 Conclusions

This paper deals with a two-layer approach for QoS-aware adaptation of SOA systems. The basic guideline we have followed in its definition has been to devise an adaptation strategy that is efficient and scalable to make realistic its use in taking runtime decisions in a rapidly changing environment. This efficiency is achieved by decomposing the service provisioning and service selection optimizations into two independent phases occuring at different time scales. The service selection problem can be solved on a fast time scale at each detected significant change which stems from the system's self or context. The sustainable frequent rate of solution derives from the formulation as a constrained optimization problem that can be efficiently solved via standard techniques and tools for linear programming. The more time consuming service provisioning problem can be solved on a slower time scale because it addresses the identification of the pool of concrete services to be used by the broker for the SLA management with the service providers. Besides being efficient, the proposed approach is also flexible, because it can be simultaneously used to serve the requests of multiple classes of users.

Our future work will address the issues concerning the implementation of the two-layer adaptation approach, such as the temporal aspects of change (e.g.., the monitoring and detection of significant changes that trigger the decision on what needs to be changed). The implementation of a system prototype we are currently working on will allow us to validate the proposed approach through a real set of experiments.

Acknowledgments

This work is supported by the Italian PRIN 2007 project "D-ASAP: Dependable Adaptable Software Architectures for Pervasive Computing".

References

1. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Trans. Auton. Adapt. Syst. 4(2), 1–42 (2009)
2. Ardagna, D., Pernici, B.: Adaptive service composition in flexible processes. IEEE Trans. Softw. Eng. 33(6), 369–384 (2007)
3. Canfora, G., Penta, M.D., Esposito, R., Villani, M.L.: A framework for qos-aware binding and re-binding of composite web services. J. Syst. Softw. 81(10), 1754–1769 (2008)
4. Cardellini, V., Casalicchio, E., Grassi, V., Lo Presti, F.: Flow-based service selection for web service composition supporting multiple qos classes. In: ICWS 2007, pp. 743–750. IEEE Computer Society, Los Alamitos (2007)
5. Maximilien, E.M., Singh, M.P.: Toward autonomic web services trust and selection. In: ICSOC 2004, pp. 212–221. ACM, New York (2004)
6. Yu, T., Zhang, Y., Lin, K.J.: Efficient algorithms for web services selection with end-to-end qos constraints. ACM Trans. Web 1(1), 1–26 (2007)

7. Zeng, L., Benatallah, B., Dumas, M., Kalagnamam, J., Chang, H.: QoS-aware middleware for web services composition. IEEE Trans. Soft. Eng. 30(5) (2004)
8. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer 36(1), 41–50 (2003)
9. Menascé, D.A., Casalicchio, E., Dubey, V.: On optimal service selection in service oriented architectures. Perform. Eval. (2009)
10. Guo, H., Huai, J., Li, H., Deng, T., Li, Y., Du, Z.: Angel: Optimal configuration for high available service composition. In: ICWS 2007, pp. 280–287. IEEE Computer Society, Los Alamitos (2007)
11. Qu, Y., Lin, C., Wang, Y., Shan, Z.: Qos-aware composite service selection in grids. In: GCC 2006, pp. 458–465. IEEE Computer Society, Los Alamitos (2006)
12. Cardellini, V., Casalicchio, E., Grassi, V., Lo Presti, F., Mirandola, R.: Qos-driven runtime adaptation of service oriented architectures. In: ESEC/FSE 2009, pp. 131–140. ACM, New York (2009)
13. Chafle, G., Doshi, P., Harney, J., Mittal, S., Srivastava, B.: Improved adaptation of web service compositions using value of changed information. In: ICWS 2007, pp. 784–791. IEEE Computer Society, Los Alamitos (2007)
14. Stein, S., Payne, T.R., Jennings, N.R.: Flexible provisioning of web service workflows. ACM Trans. Internet Technol. 9(1), 1–45 (2009)
15. Menascé, D., Ruan, H., Gomaa, H.: QoS management in service oriented architectures. Perform. Eval. 7-8(64) (2007)
16. Dan, A., Davis, D., Kearney, R., Keller, A., King, R., Kuebler, D., Ludwig, H., Polan, M., Spreitzer, M., Youssef, A.: Web services on demand: WSLA-driven automated management. IBM Systems J. 43(1) (2004)
17. Tang, P., Tai, C.: Network traffic characterization using token bucket model. In: IEEE Infocom 1999 (1999)
18. Liu, Y., Tan, M., Gorton, I., Clayphan, A.J.: An autonomic middleware solution for coordinating multiple qos controls. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 225–240. Springer, Heidelberg (2008)
19. OASIS: Web Services Business Process Execution Language Version 2.0 (2007), `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`
20. Cardoso, J., Sheth, A.P., Miller, J.A., Arnold, J., Kochut, K.J.: Modeling quality of service for workflows and web service processes. Web Semantics J. 1(3) (2004)