A New Approach to QoS Driven Service Selection in Service Oriented Architectures

Valeria Cardellini, Valerio Di Valerio, Vincenzo Grassi, Stefano Iannucci, Francesco Lo Presti
Dipartimento di Informatica, Sistemi e Produzione, Università di Roma "Tor Vergata"
Via del Politecnico 1, 00133 Roma, Italy
{*cardellini, di.valerio, iannucci*}*@ing.uniroma2.it,* {*vgrassi, lopresti*}*@info.uniroma2.it*

***Abstract*—Service selection has been widely investigated by the SOA research community as an effective adaptation mechanism that allows a service broker, offering a composite service, to bind at runtime each task of the composite service to a corresponding concrete implementation, selecting it from a set of candidates which differ from one another in terms of QoS parameters. In this paper we present a load-aware per-request approach to service selection which aims to combine the relative benefits of the well known per-request and per-flow approaches. We present experimental results obtained with a prototype implementation of a service broker. Our results show that the proposed approach is superior to the traditional per-request one and combines the ability of sustaining large volume of service requests, as the per-flow approach, while at the same time offering a finer customizable service selection, as the per-request approach.**

***Keywords*-Quality of Service, Service Oriented Architecture, Service selection.**

I. Introduction

Service Oriented Systems (SOSs) are becoming popular thanks to a widely deployed internetworking infrastructure. SOSs are composed by a possibly large number of heterogeneous third-party subsystems. As a consequence, as they grow in number and size, they also rapidly increase in complexity, which is further complicated by the highly changing execution environment where they have to operate. A major trend to tackle this growing complexity is to design SoSs as runtime self-adaptable systems, to make them able to meet both functional and non functional requirements. The former concern the overall logic to be implemented, while the latter concern the quality of service (QoS) levels that should be guaranteed to the SOSs users. In both cases, self-adaptation can leverage different mechanisms, including the tuning of control parameters of services used in the composition of a SOS, the selection of most suitable services within a set of candidates, or even a modification of the overall SOS composition logic. In this paper, we focus on self-adaptation based on QoS-driven *service selection*. The service selection goal is to determine the binding of each task in the composite service to actual implementations, leaving unchanged the composition logic. The idea at the basis of service selection is to exploit the existence, in the open marketplace as well as in the proprietary service parks, of several services implementing the same functionality with different non functional characteristics and cost [1], [2], [3].

The service selection problem has been widely investigated in recent years. A first generation of service selection solutions implements a *local* approach [4], [5], that time by time associates each running task of a SOS with the best available service that implements that task. However, this local approach can guarantee only local QoS constraints, for example the response time of a given task lower than a given threshold. Second generation solutions implement a *global* approach, where the QoS constraints are guaranteed for the whole execution of the composite SOS rather than for its single tasks. They are more suitable in a scenario where a service provider stipulates global Service Level Agreements (SLAs) with users.

Global approaches face the service selection problem at two granularity levels. At the *per-request* grain [6], [7], [8], [9], [10], [11], the adaptation focuses on each single request submitted to the system and has the goal of fulfilling the QoS constraints of that request. On the contrary, the *per-flow* grain [12], [13], [14], [15] considers the flow of requests of a user rather than the single request, and the adaptation goal is to fulfill the QoS constraints that concern the global properties of that flow, *e.g.*, the average SOS response time or its availability. However, the solutions proposed for both per-request and per-flow granularities are not satisfactory. Indeed, the former exhibits scalability problem under a sustained traffic of requests, because each request is managed independently of all the other concurrent ones. As a consequence, multiple service requests could be assigned to the same concrete service, that could be overloaded. On the other hand, the per-flow approach is not able to ensure QoS guarantees to a single request, and the user perceived QoS could be very different from that stipulated in the SLA.

To overcome these limitations, in this paper we propose a new per-request service selection policy, that we call *load-aware per-request*. The proposed policy exploits the multiple available implementations of each abstract task, and realizes a runtime probabilistic binding. In this way, different concurrent requests to the same abstract task are bound to different concrete services, realizing a randomized load balancing, in a way similar to the per-flow solutions [13]. At the same time, however, the QoS constraints are ensured *for each request* that the user submits. To achieve these goals, we need to properly calculate the probabilities that drive the service binding, taking into account the load submitted and the capacity of each concrete service in such a way to avoid overloading them. Thanks to the randomized load balancing, our policy scales better than other per-request approaches, because it can sustain higher request rates as the number of available concrete services increases: exploiting the multiple available resources is the peculiarity and the added value of our load-aware per-request service selection approach. In particular, the service selection problem is formulated as a Mixed Integer Linear Problem (MILP) and several QoS attributes are considered, like response time, availability, and cost, although others may be added without effort.

We compare our approach with the selection policy presented in [7], which is one of the per-request top performing state-of-the-art approaches. The comparison is carried out by plugging both the policies into the MOSES prototype [16] and analyzing their impact on the overall performance of a real service-oriented system. Our experimental results show that the load-aware per-request, due to its inherent scalability, is more suitable to work in an realistic environment where multiple implementations of each abstract task are expected to be available.

The rest of this paper is organized as follows. In Section II we introduce the system model, the QoS model of the SOS, and the basis idea of randomized load balancing. In Section III we present the MILP optimization problem that determines the optimal selection policy. In Section IV we first briefly describe the MOSES prototype and then demonstrate the effectiveness of our approach through a set of experiments. An overview of the service selection policies proposed so far is discussed in Section V, while the conclusions are finally drawn in Section VI.

II. System Model

We consider a broker that offers to the users a composite service P with different QoS levels and monetary prices, exploiting for this purpose a set of existing concrete services. Hence, the broker is an intermediary between users and concrete services, acting as a service provider towards the users and as a service requestor towards the concrete services used to implement the composite service, as shown in Figure 1). Its main task is to drive the adaptation of the service it manages to fulfill the Service Level Agreements (SLAs) negotiated with its users, given the SLAs it has negotiated with the concrete services and while optimizing a suitable broker utility function, *i.e.*, response time or cost.

Let us denote by $S_i \in \mathcal{S}, i = 1, \ldots, m$ the set of abstract tasks that compose the composite service P (for sake of simplicity, in the following we will use S_i and i interchangeably) and by $\Im_i$ the set of concrete services implementing the abstract task S_i. Within this framework, a core task of the service broker is to determine a proper service selection, that is to find for each abstract task S_i a corresponding implementation $cs_{ij} \in \Im_i$, so that the agreed SLAs are fulfilled. The selection criterion corresponds to the optimization of a given utility goal of the broker.

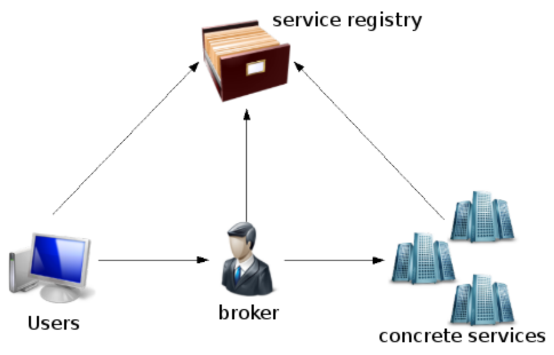


Figure 1. SOA architecture with service broker.

Within this context, our selection policy realizes a probabilistic binding: for each abstract task, it determines a set of concrete services that can be used to implement that task, with a probability associated with each of these services. Therefore, different concrete services can be bound to the same abstract task, depending on these probabilities. Thus, the actual implementation of a composite service may vary from request to request.

The probabilities used by our selection policy are determined by taking into account both the number of concurrent requests that arrive to the system and the capacity of each concrete service, that we assume the service broker knows (we will address this issue in Section IV). This probabilistic binding represents a peculiarity of our per-request approach with respect to other service selection policies with the same granularity. It allows us to realize a randomized load balancing which is aware of the capacity of each concrete service as well as of the system load, thus overcoming the limits of the per-request selection policies proposed so far. In fact, in [7], [5], [8], [6], each abstract task is deterministically bound to only one concrete service that can be overloaded under high load conditions. On the hand, by exploiting multiple service implementations our per-request solution is able to share the load.

However, the service broker has to satisfy the QoS levels agreed in the SLAs with its user. Since different concrete services implementing the same task can differ in their QoS attributes values, we may obtain different composite service implementations with distinct overall QoS attributes, depending on the actual service binding. To guarantee the QoS constraints, it is necessary to choose the services so that all the feasible implementations respect them.

The rest of this section is organized as follows. In Section II-A we introduce the SLA model and the QoS attributes. In Section II-B we explain how the randomized load balancing is realized. Finally, in Section II-C we illustrate how the QoS attributes are calculated along the composite service.

A. SLA Model

The QoS levels offered by the service providers are defined in SLAs. Since our selection policy aims at satisfying every single request, the SLA states conditions that are restricted to the single request. In general, SLA conditions may refer to different kinds of functional and non-functional attributes of the service. We consider in this work the following attributes and relative conditions:

- *response time*: the upper bound on the interval of time elapsed from the service invocation to its completion;
- *availability*: the lower bound on the probability that the service is accessible when invoked;
- *cost*: the upper bound on the price charged for the service invocation.

We model the SLA between the service broker and its users as a tuple $\langle R_{max}, A_{min}, C_{max} \rangle$, where R_{max} and C_{max} represent the upper bound on the response time and cost, respectively, and A_{min} the lower bound on the availability. Since the service broker plays an intermediary role, in its turn it also acts as a service user against the providers of the concrete services it uses to implement the abstract tasks in the service composition. Each of these concrete services is characterized by its own SLA, that we model as an extension of the SLA offered by the broker to its users: response time, availability, and cost maintain the same semantics, but we add a new parameter, L_{ij}, which is a threshold on the maximum request rate that can be submitted to the concrete service cs_{ij}. The cs_{ij} provider satisfies the QoS attributes in the advertized SLA as long as the request rate does not exceed L_{ij}; in case of violation of the load threshold, there is no guarantee on the QoS levels of cs_{ij}.

B. Randomized Load Balancing

The core of our per-request selection policy is the randomized load balancing of the requests directed to each abstract task S_i, so that they are switched to multiple concrete services implementing it. The load balancing is tuned on the basis of the capacity of each concrete service cs_{ij}, defined by the parameter L_{ij}, and of the rate of requests submitted by the users to S_i. As we already mentioned, an abstract task is bound by the broker to a set of concrete services, and each concrete service in this set has an associated probability. So, at abstract task binding time, only one of these services is probabilistically chosen. As a consequence, only a fraction of the incoming requests is switched to a given concrete service cs_{ij}, and this fraction depends on the probability x_{ij} determined by the broker. We define a service selection policy as the set of all these probabilities, that we represent with the vector $\boldsymbol{x} = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m]$, where for each entry $\boldsymbol{x}_i = [x_{ij}], i \in \mathcal{S}, j \in \Im_i$, the constraints $x_{ij} \in [0, 1]$ and $\sum_{j \in \Im_i} x_{ij} = 1$ hold. Our idea is to drive the value of the x_{ij} probabilities, forcing on each x_{ij} an upper bound P_{ij}, so that the fraction of requests switched by the broker to the concrete service cs_{ij} does not overload it. The upper bound P_{ij} is calculated through the ratio $P_{ij} = \dfrac{L_{ij}}{\lambda_i}$, where λ_i is the actual request rate to the abstract task S_i and L_{ij} is the load threshold for cs_{ij}. If P_{ij} is greater than 1, it means that there is no upper bound because cs_{ij} is able to satisfy all the incoming requests to S_i on its own. Vice versa, if P_{ij} is less than 1, cs_{ij} alone cannot satisfy all the requests directed to S_i but it must be backed by other concrete services so that their overall capacity can sustain the submitted load.

C. QoS Attributes of the Composite Service

The QoS attributes of the composite service can be calculated using the service selection policy $\boldsymbol{x}$, but a model of the composite service workflow is also needed. In the following subsections we introduce this model and, afterwards, the QoS attributes computation.

1) Composite Service Graph: We assume that all the workflows of the composite services managed by the service broker have a single initial task, or that they start with a fork-join parallel sequence. Furthermore, we assume that for each conditional branch we know the probability of executing it; similarly, we assume to know the probability of reiterating loops.

The composite service graph is obtained by transforming the workflow of the composite service as in [7]. In particular, loops are peeled, *i.e.*, they are transformed in a sequence of branch conditions, each of which evaluates if the loop has to continue with the next iteration or it has to exit, according to the branch probability introduced above. A pre-requisite for loop peeling is the knowledge of the maximum number of supposed iterations. We calculate this value as the *p-percentile* of the distribution of reiterating the loop. After loop peeling, the composite service can be modeled as a Directed Acyclic Graph (DAG). As in [7], we define:

- *Execution path.* An execution path ep_n is a multiset of tasks $ep_n = \{S_1, S_2, \ldots, S_I\} \subseteq \mathcal{S}$, such that S_1 and S_I are respectively the initial and final tasks of the path and no pair $S_i, S_j \in ep_n$ belongs to alternative branches. We need a multiset rather than a simple set because a single task may appear several times in the execution path. An execution path may also contain parallel sequences, but it does not contain loops, which are peeled. A probability of execution p_n is associated with every execution path and can be calculated as the product of the probabilities of executing the branch conditions included in the path. Similarly, the branch conditions that arise from loop peeling produce other execution paths.
- *Subpath.* A subpath of an execution path ep_n is a sequence of tasks $[S_1, \ldots, S_I]$, from the initial to the end task, that does not contain any parallel sequence. In other words, each branch b of a parallel sequence

identifies a subpath inside the execution path ep_n. We denote a subpath by sp_b^n.

Therefore, the set of all the execution paths identifies all the possible execution scenarios of the composite service. The QoS constraints must hold for every execution path to guarantee the SLAs the service broker stipulated with its users.

2) QoS Attributes Computation: Given the service selection policy $\boldsymbol{x}$ and the execution paths that arise from the composition logic, we can calculate the QoS attributes of each abstract task and then the overall QoS attributes of the composite service. We are interested both in the average QoS perceived by the users and in its worst case value. As discussed below, we need both these values to maximize the broker utility function and satisfy the QoS constraints.

Let r_{ij} be the response time of the concrete service cs_{ij}, a_{ij} its availability, and c_{ij} its cost. The average QoS values of the abstract task S_i, namely, the average response time R_i, the availability A_i, and the average cost C_i, are given by the following expressions:

$$R_i = \sum_{j \in \Im_i} r_{ij} x_{ij} \quad (1)$$

$$A_i = \sum_{j \in \Im_i} a_{ij} x_{ij} \quad (2)$$

$$C_i = \sum_{j \in \Im_i} c_{ij} x_{ij} \quad (3)$$

The worst case QoS values, denoted by R_i^w, A_i^w, and C_i^w, are given by:

$$R_i^w = \max_{j \in \Im_i} r_{ij} y_{ij} \quad (4)$$

$$A_i^w = \min_{j \in \Im_i} a_{ij} y_{ij} \quad (5)$$

$$C_i^w = \max_{j \in \Im_i} c_{ij} y_{ij} \quad (6)$$

where y_{ij} is a binary variable indicating whether the concrete service cs_{ij} can be ever bound to the abstract task S_i, *i.e.*, $y_{ij} = 1$ if $x_{ij} > 0$ and 0 otherwise.

Using these formulas and the notion of execution paths and subpaths, we can calculate the QoS attributes along each execution path itself, using the aggregation formulas presented in [7]. Note that the same formulas apply both for the average case and for the worst case; therefore, for the sake of simplicity, we show only the latter case. We denote by R_n the maximum response time of the execution path ep_n, with A_n its minimum availability, and with C_n its maximum cost; these are, in other words, the QoS attributes values calculated in the worst case scenario. They are:

$$R_n = \max_{sp_b^n \in ep_n} \sum_{S_i \in sp_b^n} R_i^w \quad (7)$$

$$A_n = \prod_{S_i \in ep_n} A_i^w \quad (8)$$

$$C_n = \sum_{S_i \in ep_n} C_i^w \quad (9)$$

While the cost and the availability are simply obtained, respectively, as a sum and as a multiplication of the QoS attributes of each abstract task in the execution path, the matter is slightly different for the response time. Indeed, the response time of an execution path is equal to the response time of the longest subpath inside the execution path itself.

III. Optimization Problem

Given a composite service P, the goal of the service broker is to find a selection policy $\boldsymbol{x}$ that ensures the QoS constraints for every execution scenario, *i.e.*, for each execution path ep_n that arises from P, while realizing the randomized load balancing. The selection policy $\boldsymbol{x}$ is calculated by solving a suitable optimization problem. We formulate this optimization problem as a Mixed Integer Linear Problem (MILP), with the following decision variables:

- x_{ij}: this variable takes value in the range $[0, 1]$ and represents the probability that the concrete service $cs_{ij} \in \Im_i$ is bound to the abstract task S_i; it is used to drive the randomized load balancing.
- y_{ij}: this variable is equal to 1 if the concrete service cs_{ij} is bound to the abstract task S_i with a given probability defined by x_{ij}, 0 otherwise. It is used to ensure that the QoS constraints are met.

While the QoS constraints are evaluated using the worst case values of the QoS attributes for each abstract task, the objective function is maximized using the average values, because it is the value that is expected along multiple executions of the composite service. In particular, the optimization problem maximizes the aggregated values of QoS calculated over all the possible execution paths that arise from the composite service description, taking into account the relative probability p_n. The aggregated values are obtained using the Simple Additive Weighting (SAW) technique as scalarization method.

In the first phase, each quality dimension along an execution path is normalized according to the following formulas, depending on whether the QoS attribute is a positive (10) or a negative (11) one. A QoS attribute is defined positive (negative) if the greater the value is, the greater (lower) the quality of that attribute. Availability is an example of positive attribute (the higher the availability, the better the quality is), while response time is an example of negative attribute (the lower the response time, the better the quality is).

$$z_n^h(\boldsymbol{x}) = \begin{cases} \dfrac{q_n^h(\boldsymbol{x}) - \min q_n^h}{\max q_n^h - \min q_n^h}, & \text{if } \max q_n^h \neq \min q_n^h \\ 1, & \text{if } \max q_n^h = \min q_n^h \end{cases} \quad (10)$$

$$z_n^h(\boldsymbol{x}) = \begin{cases} \dfrac{\max q_n^h - q_n^h(\boldsymbol{x})}{\max q_n^h - \min q_n^h}, & \text{if } \max q_n^h \neq \min q_n^h \\ 1, & \text{if } \max q_n^h = \min q_n^h \end{cases} \quad (11)$$

In the above formulas, $q_n^h(\boldsymbol{x})$ is the h-th quality dimension value calculated over the execution path ep_n using the selection policy $\boldsymbol{x}$. $\max q_n^h$ and $\min q_n^h$ indicate its maximum and minimum values, respectively, and they can be estimated across several composite service executions.

In the second phase a score is obtained using a weighted sum of the normalized quality attributes, as follows:

$$score_n = \sum_h w_h z_n^h(\boldsymbol{x}) \tag{12}$$

where the weight w_h specifies the relative importance that the service broker must give to a QoS attribute with respect to the others.

Finally, the objective function is obtained using the following weighted formula:

$$F(\boldsymbol{x}) = \sum_n p_n score_n(\boldsymbol{x}) \tag{13}$$

The optimal service selection policy $\boldsymbol{x}$ can be obtained solving the following optimization problem (for sake of simplicity, we use n instead of ep_n):

$$\begin{array}{rlll}
 & \textbf{max } F(\boldsymbol{x}) & & \\
\textbf{subject to:} & \sum_{j \in \Im_i} x_{ij} = 1 & \forall i & (14) \\
 & x_{ij} \leq P_{ij} & \forall i, \forall cs_{ij} \in \Im_i & (15) \\
 & x_{ij} \leq y_{ij} & \forall i, \forall cs_{ij} \in \Im_i & (16) \\
 & r_j y_{ij} \leq R_i^w & \forall i, \forall cs_{ij} \in \Im_i & (17) \\
 & a_j y_{ij} \geq A_i^w & \forall i, \forall cs_{ij} \in \Im_i & (18) \\
 & c_j y_{ij} \leq C_i^w & \forall i, \forall cs_{ij} \in \Im_i & (19) \\
 & \sum_{i \in sp_b^n} R_i^w \leq R_n & \forall sp_b^n \in ep_n, \forall n & (20) \\
 & \sum_{i \in ep_n} log(A_i^w) = A_n & \forall n & (21) \\
 & \sum_{i \in ep_n} C_i^w = C_n & \forall n & (22) \\
 & R_n \leq R_{max} & \forall n & (23) \\
 & A_n \geq log(A_{min}) & \forall n & (24) \\
 & C_n \leq C_{max} & \forall n & (25) \\
 & x_{ij} \in \Re^+ & \forall i, \forall cs_{ij} \in \Im_i & \\
 & y_{ij} \in \{0,1\} & \forall i, \forall cs_{ij} \in \Im_i & \\
 & R_i^w, A_i^w, C_i^w \in \Re^+ & \forall i & \\
 & R_n, C_n \in \Re^+ & \forall n & \\
 & A_n \in \Re^- & \forall n &
\end{array}$$

The constraints (14) guarantee that the sum of the probabilities of choosing the concrete services is equal to 1 for each task S_i. The constraints (15) define the upper bound to the probability of choosing a concrete service cs_{ij}. These two constraints families implement the randomized load balancing policy. The constraints from (17) to (19) express the response time, availability, and cost of every abstract task S_i in terms of the worst concrete services that are selected to implement that task, as we discussed in Section II-C. The constraints (20) evaluate the response time of each execution path ep_n as the response time of its longest subpath sp_b^n, while the constraints (21) and (22) refer respectively to the availability and cost of each execution path ep_n. Finally, the constraints from (23) to (25) are the QoS constraints to be fulfilled. Note that we use the logarithm of the availability instead of the availability in our optimization problem because we need to linearize Equation (8) to put it in our MILP problem.

IV. Experimental Results

In this section, we present the experimental analysis we have conducted using the MOSES prototype to compare the effectiveness of the proposed load-aware per-request approach with respect to the traditional per-request approach proposed by Ardagna and Pernici in [7]. First we briefly describe the MOSES prototype and the experimental environment, then we analyze the experimental results.

A. MOSES Prototype

MOSES, which stands for *MOdel-based SElf adaptation of SOA systems*, is a QoS-driven runtime adaptation framework for service-oriented systems [14]. It is intended to act as a *service broker* which offers to prospective users a composite service with QoS guarantees, exploiting for this purpose the runtime binding between the abstract functionalities of the composite service and a pool of existing concrete services that implement the abstract functionalities. Its main task is to drive the adaptation of the composite service to fulfill the QoS goals stipulated with its users.

We have designed the MOSES prototype with a flexible and modular system architecture, where each module performs a specific functionality and its implementation can be changed without impacting on the rest of the system. In the following, we provide an overview on the MOSES system; a detailed description of the prototype can be found in [16].

We first describe what we refer to as the core MOSES modules (namely, the *BPEL Engine*, the *Adaptation Manager*, and the *Optimization Engine*, together with the *Storage* layer); then, we present the remaining ones that enrich the basic functionalities.

The *Optimization Engine* computes the selection policy $\boldsymbol{x}$ that drives the runtime binding. The modular architecture of MOSES allows us to develop multiple implementations of the service selection optimization policies, possibly using external tools for finding the optimal solution. To this end, the Optimization Engine exposes the same interface to other MOSES modules irrespectively of the specific policy. Examples of already implemented policies are the per-flow optimization policies in [13], [17] and the per-request optimization policy in [7]. Furthermore, we have implemented the load-aware selection policy presented in this paper, that

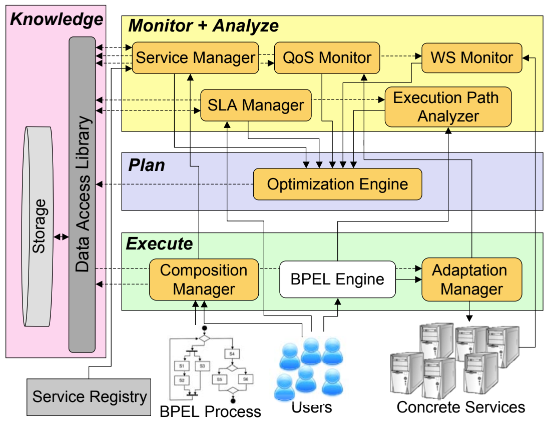


Figure 2. MOSES high-level architecture based on the MAPE-K model

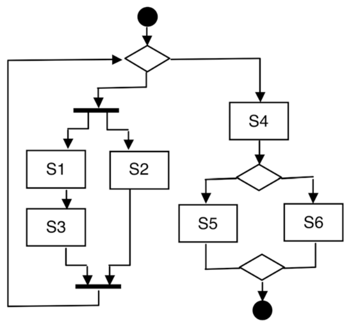


Figure 3. Workflow of the composite service managed by MOSES

tries to mix the best of the previously implemented selection policies.

The *BPEL Engine* executes the composite service, described in BPEL [18], that defines the user-relevant business logic.

The *Adaptation Manager* is the actuator of the adaptation actions determined by the Optimization Engine: it is actually a proxy interposed between the BPEL Engine and any external service provider. Its task is to dynamically bind each abstract task's invocation to the real endpoint according to the optimal solution determined by the Optimization Engine.

MOSES is architected as a self-adaptive system based on the MAPE-K (Monitor, Analyze, Plan, Execute, and Knowledge) reference model for autonomic systems [19]. Figure 2 shows how the MOSES modules implement each MAPE-K macro-component, together with the system inputs (i.e., the composite service and the pool of candidate concrete services). This input is used to build a model (Execute), which in turn is used and is kept up-to-date at runtime (Monitor). The monitored parameters are analyzed (Analyze) in order to know if adaptation actions (*i.e.*, a new service selection policy) have to be taken. A new selection policy $\boldsymbol{x}$ is calculated (Plan) to react to some external significant event, such as a significant violation of the SLA parameters. The BPEL Engine, together with the Adaptation Manager, belong to the Execute macro-component because their task is to execute the logic of the business processes; on the other hand, the Optimization Engine constitutes the Plan macro-component because it is involved in computing the selection policy.

The basic functionalities implemented in the Execute and Plan macro-components are enriched by the modules belonging to the Monitor+Analyze macro-component: they capture changes in the MOSES environment and, if they are relevant, modify at runtime the system model kept in the Storage layer, also triggering the Optimization Engine to make it calculate a new service selection policy. The new service selection policy $\boldsymbol{x}$ will be then calculated using the system model view as updated by the monitoring modules. Specifically, the *Service Manager* and *WS Monitor* respectively detect the addition or removal of concrete services (the latter due either to graceful failures or crashes). The *QoS Monitor* measures the actual values of the QoS attributes provided by MOSES to its users and offered to MOSES by its service providers and detects violations of the service level objectives stated in the SLAs. The *Execution Path Analyzer* tracks variations in the usage profile of the abstract tasks, allowing for example to dynamically update the probability of executing thee conditional branches in the workflow of the service composition. Finally, the *SLA Manager* manages the user registration with the associated SLA, possibly performing an admission control.

We have implemented the MOSES prototype exploiting the Java Business Integration (JBI) implementation provided by OpenESB and MySQL for the storage layer. We have used Sun BPEL Service Engine to orchestrate the service composition. The Optimization Engine relies on IBM ILOG CPLEX Optimizer [20] as optimization software package to solve the per-request optimization problems.

B. Experimental Setup

The testing environment consists of 3 Intel Xeon quad-core servers (2 Ghz/core) with 8 GB RAM each (nodes 1, 2, and 3), and 1 KVM virtual machine with 1 CPU and 1 GB RAM (node 4); a Gb Ethernet connects all the machines. The MOSES prototype is deployed as follows: node 1 hosts all the MOSES modules in the Execute macro-component, node 2 the storage layer together with the candidate concrete services, and node 3 the modules in the Monitor+Analyze and Plan macro-components. Finally, node 4 hosts the workload generator.

We consider the composite service defined by the workflow in Figure 3, composed of 6 stateless tasks, and assume that 4 concrete services (with their respective SLAs) have been identified for each task, except for tasks S_1 and S_3 for which 5 implementations have been identified. The respective SLA parameters, shown in Table I (top), differ in terms of cost c_{ij}, availability a_{ij}, and response time r_{ij}

Table I
SLA PARAMETERS FOR CONCRETE SERVICES (TOP) AND SERVICE CLASSES (BOTTOM)

cs	r_{ij}	a_{ij}	c_{ij}
cs_{11}	2	0.995	6
cs_{12}	1.8	0.99	6
cs_{13}	2	0.99	5.5
cs_{14}	3	0.995	4.5
cs_{15}	4	0.99	3
cs_{21}	1	0.995	2
cs_{22}	2	0.995	1.8
cs_{23}	1.8	0.99	1.8
cs_{24}	3	0.99	1
cs_{31}	1	0.995	5
cs_{32}	1	0.99	4.5
cs_{33}	2	0.99	4
cs_{34}	4	0.95	2
cs_{35}	5	0.95	1

cs	r_{ij}	a_{ij}	c_{ij}
cs_{41}	0.5	0.995	1
cs_{42}	0.5	0.99	0.8
cs_{43}	1	0.995	0.8
cs_{44}	1	0.95	0.6
cs_{51}	1	0.995	3
cs_{52}	2	0.99	2
cs_{53}	3	0.99	1.5
cs_{54}	4	0.95	1
cs_{61}	1.8	0.99	1
cs_{62}	2	0.995	0.8
cs_{63}	3	0.99	0.6
cs_{64}	4	0.95	0.4

Class k	$R^k_{\max}$	$A^k_{\min}$	$C^k_{\max}$
1	16	0.88	55
2	18	0.85	50
3	20	0.82	45
4	22	0.79	40

(measured in sec). In the following experiments, we used this *baseline* set composed of 26 concrete services as well as an enlarged set of concrete services, constructed by doubling the baseline set (in the following, we refer to the latter as *2x baseline*).

The concrete services are simple stubs, without internal logic; however, their non-functional behavior conforms to the guaranteed levels expressed in their SLA. The perceived response time is obtained by modeling each concrete service as a $M/D/m/PS$ queue implemented inside the Web service deployed in a Tomcat container. The $M/D/m/PS$ model is parameterized in such a way to have an average CPU usage between 65% and 70% when the request rate is equal to 10 req/sec. Table I (bottom) shows the SLAs offered by MOSES to the composite service users according to different service classes.

To issue requests to the composite service managed by MOSES we have developed a workload generator, implemented in C language using the Pthreads library. The workload generator can be configured to issue requests to MOSES class by class at a fixed rate.

We conducted four sets of experiments, aiming at testing the service selection policies under different scenarios and loads. The first set of experiments was performed to point out the scalability problems of the traditional per-request approach; the second set was carried out to compare the performance of the traditional per-request policy versus the load-aware one; the third set was performed to analyze the scalability of the load-aware per-request selection. In the first three sets of experiments, each experiment is composed by several runs lasting 15 minutes each, during which the workload generator generates requests corresponding to the service class 2 (see the k=2 row in Table I (bottom) for its SLA) at a constant rate. The request rate is then increased run by run until the system is stable. Finally, the last set of experiments was performed in order to prove the effectiveness of our load-aware policy. In particular, we generated for every service class a constant request rate for the first half of the experiment, then increasing the request rate only for class 2 (in the second half on the experiment.

The main performance metric we measured is the response time of the composite service. We also measured the CPU utilization of the concrete services to analyze the different effects of the request load distribution among the concrete services that the traditional and the load-aware per-request policies determine.

C. Experimental Results

In the first set of experiments, we ran three load tests on MOSES using the traditional per-request policy in [7]. In the first test, we used the baseline set of concrete services without instrumenting any of the MOSES modules in the Monitor and Analyze macro-components. In the second test, we used the previous configuration, but we exploited the 2x baseline set of concrete services. Finally, in the last test we used the 2x baseline set of concrete services and we added the support of the QoS Monitor, in order to detect SLA violations of the response time of the concrete services and, in positive case, to determine a new service selection policy that exploits different concrete services implementations.

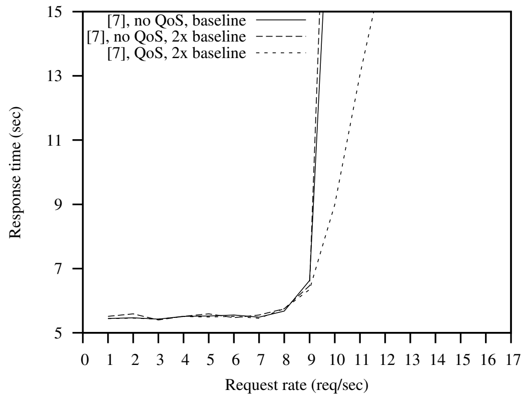


Figure 4. Response time of the traditional per-request service selection policy

Figure 4 shows the average response time perceived by users at different request rates submitted to MOSES. We observe that for all the three tests the response time is nearly constant until the request rate reaches 7 requests/sec. From this point on, the response time of both the tests without QoS Monitor (*[7], no QoS, baseline* and *[7], no QoS, 2x baseline* curves), regardless of the used concrete services set, rapidly grows because the per-request service selection does not exploit the presence of different service implementations, always using the same service identified as the best one. In the test with the QoS Monitor enabled (*[7], QoS, 2x baseline* curve), the response times of the concrete services are collected, their average calculated every 2 sec. and analyzed; if the QoS Monitor finds out that the currently used concrete implementations do not have an adequate performance (*i.e.,* their are violating the response time contractualized in the SLA), it triggers the Optimization Engine to compute a new optimal policy $\boldsymbol{x}$, using the actual response times of the concrete services instead of those declared into the SLAs. As a result, the currently used overloaded services will not be used in the near future, but they will be candidate for re-usage when the new selected concrete services will in their turn become overloaded. However, the introduction of the QoS Monitor provides only a modest performance improvement; even if the QoS Monitor invocation frequency is relatively high (every 2 sec.), the reaction is not quick enough to address higher request rates. We can conclude that the traditional per-request approach is not able to scale out the available services implementations, and thus it is unable to sustain higher request rates than those sustainable by the bottleneck concrete service.

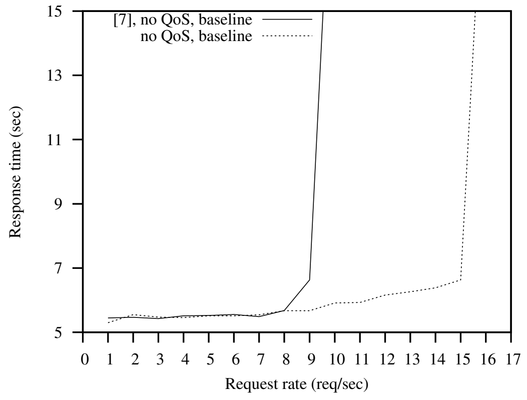


Figure 5. Response time of the traditional versus load-aware per-request service selection policies

The second set of experiments compares the traditional per-request and the load-aware per-request selection policies. These experiments use the baseline set of concrete services and do not involve any Monitor or Analyze MOSES component. Figure 5 compares the average response time according to the request rate submitted to MOSES when using the two different policies. We observe that the response times achieved by the two policies perfectly overlap until the request rate reaches the saturation point of the traditional per-request policy. From this point on, the former (*[7], no QoS, baseline* curve) is not able to exploit the available implementations, while the load-aware policy (*no QoS, baseline* curve) performs better, scaling out the available concrete services. Therefore, the load-aware approach is able to sustain higher request rates than the traditional per-request, given that there are available concrete services to be exploited.

To show the effectiveness of the load balancing, we monitored the CPU usage of the concrete services during the execution of the experiments. Since every concrete service is implemented as a $M/D/m/PS$ queue, the CPU usage has been computed with the formula $\frac{\lambda_{ij} * T_{ij}}{nCPU_{ij}}$, where: λ_{ij} is the request rate directed to the j_{th} implementation of S_i (that is, cs_{ij}), T_{ij} its service time, and $nCPU_{ij}$ the number of CPUs available to that service implementation.

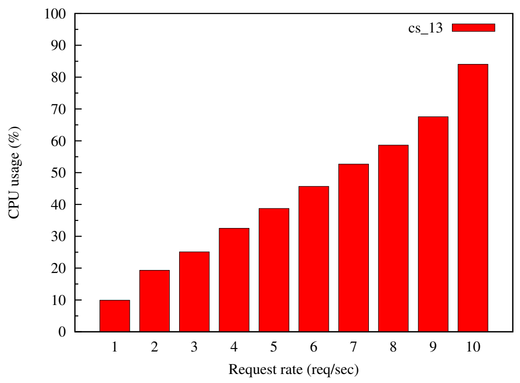


Figure 6. CPU usage of the concrete services under the traditional per-request policy

Figure 6 shows the CPU usage of cs_{13}, which is the single concrete service used by the traditional per-request optimization approach to implement S_1 . It is easy to see that the load almost linearly increases until it reaches the CPU usage equal to 85%, value at which the system becomes unstable (see Figure 5).

Figure 7 shows the CPU usage of the concrete services used by the load-aware policy to implement S_1. Differently from the traditional strategy, with the load-aware policy multiple concrete services can be used to implement the same task. In particular, when the request rate is low (from

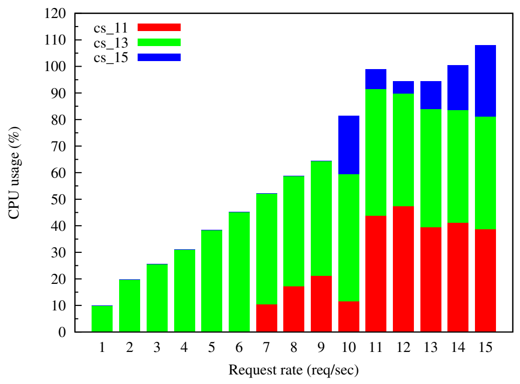


Figure 7. CPU usage of the concrete services under the load-aware per-request policy

1 req/sec to 6 req/sec), there is no need to use multiple concrete services (we recall that each concrete service is modeled so to have an average CPU usage between 65% and 70% when the request rate is equal to 10 req/sec). Therefore, for the low request rate only cs_{13} is used, like in the traditional per-request policy. When the request rate increases from 7 req/sec to 9 req/sec, the concrete services cs_{13} and cs_{11} are both used. From 10 req/sec on, cs_{15} is also used to implement S_1, therefore the load is balanced across three concrete services. We observe that the cumulative load is not monotonically increasing, because the concrete services model different underlying hardware: cs_{11} and cs_{15} have 29 CPUs each, while cs_{13} has 25 CPUs. Therefore, when more load is directed to a concrete service with a larger capacity, the overall load decreases.

We carried out the third set of experiments to show the scalability capabilities of the load-aware per-request policy. In these experiments we used both the baseline and the 2x baseline sets of concrete services, without deploying the QoS Monitor module. Figure 8 shows the scaling capabilities of the load-aware per-request service selection: until there is no need to use more than one concrete service at a time (*i.e.*, until the request rate reaches around 7 req/sec), it does not matter to have a larger number of available implementations; therefore, the test with the baseline set of concrete implementations behaves as the test with 2x baseline. However, at higher request rates, the availability of a larger set of candidate services provides better response times and allows to manage the request rates without incurring in overloading, because the load can be better distributed among the available implementations.

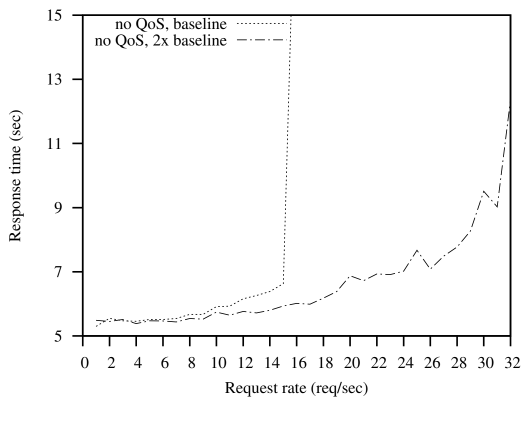


Figure 8. Response time of the load-aware per-request policy under the two sets of concrete services

Finally, we conducted a last set of experiments simulating several concurrent users characterized by different service classes. The goal is to prove the effectiveness of the MOSES adaptation under the load-ware per-request policy despite variations in the submitted workload. To this end, each of the four service classes submits requests at a constant rate equal to 1 req/sec, except class 2 for which we increased the request rate from 1 to 10 req/sec in the second half of the test. Therefore, in the first half of the experiment, the aggregate workload is equal to 4 req/sec, which is also easily manageable by the traditional per-request policy; on the other hand, in the second half of the experiment we submitted an aggregated workload to the system equal to 13 req/sec, which cannot be sustained by the traditional per-request policy. The overall test lasted 1 hour.

Figures 9(a)- 9(d) show that the perceived response times are far below the response times agreed in the SLAs and represented by the horizontal lines; this can be explained by observing that the average behavior is very different from the worst case considered in the formulation of the optimization policy. This latter issue could be addressed by considering SLAs where the response time constraint is specified in terms of bounds on the percentile.

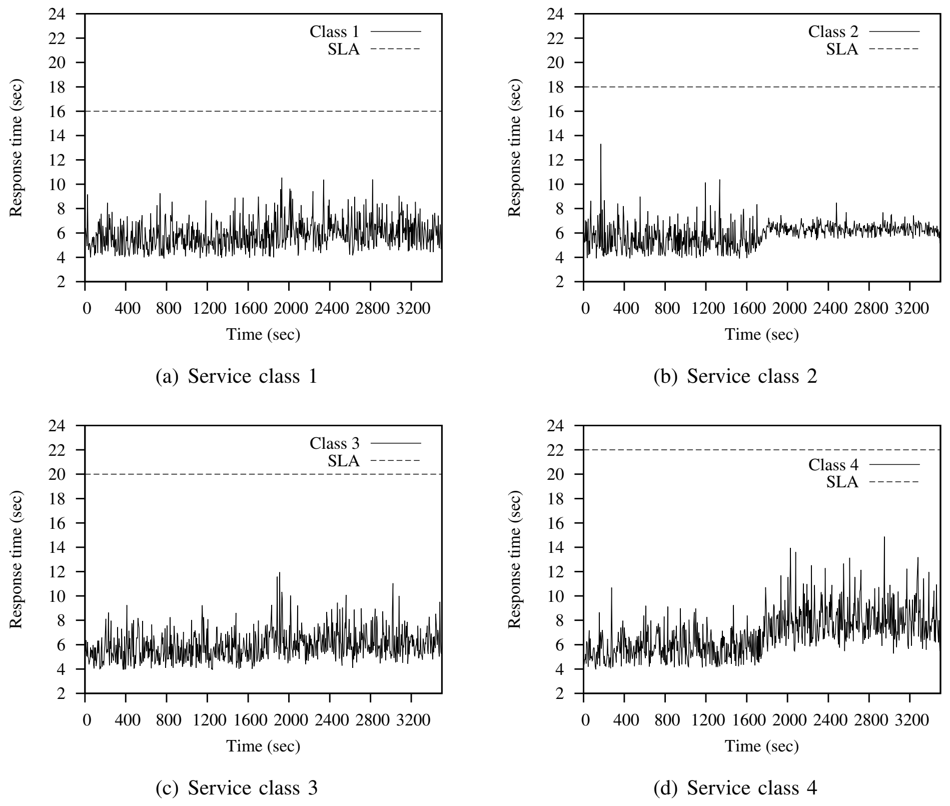
Table II
AVERAGE RESPONSE TIMES OF THE LOAD-AWARE PER-REQUEST POLICY FOR ALL SERVICE CLASSES UNDER LIGHT AND HEAVY LOADS

Class	Light load	Heavy load
1	5.514 sec	6.254 sec
2	5.485 sec	6.350 sec
3	5.509 sec	6.357 sec
4	5.794 sec	8.112 sec

Table II shows the average response times perceived by the users when issuing requests either to a light loaded or to an heavy loaded system according to the service class. When the system is subject to a light load, there are not appreciable differences among the service classes. On the other hand, when the load increases, the average response time perceived by class 4 (which is the one with the least stringent SLA) suffers more the load increase. The motivation is that class 4 requests can only exploit a limited number of concrete services, because of the lowest maximum cost in the SLA (see Table I (bottom)); therefore, to satisfy the cost constraint they cannot be distributed among all the available concrete services.

V. RELATED WORK

The service selection problem has been widely investigated in the last years and many solutions can be found in literature. They can be broadly classified into two categories, depending on whether they implement a *local* or a *global* approach. In the local approach, *e.g.*, [4], [5], [7], only QoS constraints on the execution of single abstract tasks can be predicated: the concrete services are selected one at the time by associating the running abstract task to the best candidate concrete service which supports its execution. On the other hand, the global approach, *e.g.*, [5], [6], [7], [8], [9], [10], [11], [12], [13], [15], [17], [21], aims to ensure the QoS constraints on the whole execution of the composite service.



(a) Service class 1

(b) Service class 2

(c) Service class 3

(d) Service class 4

Figure 9. Response time of the load-aware per-request policy for all service classes over time

Most of the proposed methodologies for service selection focus on the global approach and adopt in particular the *per-request* granularity level, formalizing the service selection as an instance of suitable optimization problems [5], [6], [7], [8], [9], [10], [11], [21]. At the per-request granularity level, the service selection concerns each single request submitted to the service oriented system and has the goal to fulfill the QoS constraints of that specific request, independently of the concurrent requests that may be addressed to the system. Zeng et al. [5] present a global planning approach to select an optimal execution plan by means of integer programming. Their QoS model consider the following attributes: price, availability, reliability, and reputation. Ardagna and Pernici [7] model the service composition as a MILP problem and their technique is particularly efficient for long running process instances. Their approach is formulated as an optimization problem handling the whole application instead of each execution path separately. The proposal by Alrifai and Risse [6] is slightly different, as the global approach is combined with local selection techniques to reduce the optimization complexity. In this solution, the global constraints are reduced to local constraints using integer programming in such a way that satisfying the latter also ensure the former. Another approach to address the complexity of the optimization problem is proposed by Alrifai et al. in [21], where the set of the available concrete services is pruned on the basis of the skyline notion before resolving the optimization problem itself. Canfora et al. [8] follow a quite different strategy for optimal selection, relying on genetic algorithms. They define an iterative procedure to search for the best solution of a given problem among a constant size population without the need for linearization required by integer programming. Since the per-request service selection problem is NP-hard, other heuristic policies have been also proposed (e.g., see [9], [10], [11]).

The common factor to all the solutions discussed so far is that each abstract task is only bound, from time to time, to a single concrete service. It seems reasonable to suppose that, for a given class of requests, the same optimal binding between an abstract task and a corresponding concrete service holds until a significant change detected in the execution environment triggers the calculation of a new binding. Hence, the per-request policies have the drawback of possibly overloading the selected services during the time interval that interlapses between two subsequents changes, because each request is handled independently of all the others.

This drawback is partially solved by those proposals that adopt the *per-flow* granularity level [12], [13], [15], [17], where the focus is on the overall flow of requests of a user, rather than on a single request. Under the per-flow granularity, the service selection goal is to fulfill the QoS constraints that concern the global properties of that flow, *e.g.*, the average response time of the composite service. In [15] the service selection problem is addressed with a LP formulation: an abstract task is probabilistically bound at runtime to several concrete services thus realizing a request load balancing. However, the actual load submitted to each concrete service is not taken into account. Also in [13], [17] the service selection problem is formulated as a LP problem that probabilistically binds each abstract task to multiple corresponding concrete services, but, differently from the work in [15], in these proposals the load submitted to each concrete service is accounted. The system incoming workload is also taken into account by Ardagna and Mirandola in [12], but the service selection is formulated as a constrained non-linear optimization problem.

Although the solutions to the service selection problem presented in [12], [13], [17] take into account the load balancing issue and can scale better than the per-request approaches because of the corresponding formulation of the optimization problem, they work on a per-flow basis. As a consequence, the QoS constraints are ensured on average and in the long term, but no QoS guarantee is given to each single submitted request. In this paper we have proposed a new approach to service selection, that combines the different advantages of the per-request and per-flow approaches proposed so far: it scales similarly to the per-flow ones with respect to the submitted request load, but allows to ensure the QoS constraints on a per-request basis.

VI. Conclusions

In this paper we have presented a load-aware per-request policy to address the service selection issue for a service broker which offers a composite service with QoS constraints. The proposed policy realizes a randomized load balancing of the requests submitted to each abstract task, exploiting the multiple concrete implementations available in the open service market-place. To avoid overloading the chosen concrete services, the load balancing is tuned by taking into account the capacity of each concrete service and the load submitted to the abstract task. In particular, a probability is assigned to each concrete service in a proportional way to its capacity and, for each single request, the concrete services to be invoked are selected according to these probabilities.

Using a prototype implementation, we have compared our approach with one of the top performing per-request service selection policies, presented in [7]. Our experimental results show the scalability of the load-aware per-request policy: it can sustain higher request rates than the per-request policy in [7], because it allows to concurrently exploit multiple concrete services using the load balancing mechanism, while the approach in [7] uses only one concrete service at a time, *i.e.*, it directs all the concurrent requests in the same QoS class to the same concrete service. The experimental results also show that for a service broker using our policy the maximum sustainable load grows with the number of available concrete services, while this does not happen with the policy in [7]. Indeed, the latter is completely insensitive to the number of available implementations for each abstract task, while the load-aware service selection has proved suitable to work in a real operative scenario.

As future work we will consider QoS constraints which specify bounds on the percentile of the response time, because the user perceived QoS can be better expressed in terms of percentile rather than mean values. Furthermore, due to the complexity of the MILP formulation of the proposed policy, we will consider to apply mechanisms that allow to prune the set of available concrete services in such a way to speed-up the problem resolution in case of large problem instances.

References

[1] H. R. Motahari-Nezhad, J. Li, B. Stephenson, S. Graupner, and S. Singhal, "Solution reuse for service composition and integration," in *Proc. WSCA '09*, 2009.

[2] E. D. Nitto, C. Ghezzi, A. Metzger, M. P. Papazoglou, and K. Pohl, "A journey to highly dynamic, self-adaptive service-based applications," *Autom. Softw. Eng.*, vol. 15, no. 3-4, pp. 313–341, 2008.

[3] M. P. Papazoglou, "Service-oriented computing: Concepts, characteristics and directions," in *Proc. IEEE WISE '03*, 2003.

[4] Z. Maamar, Q. Z. Sheng, and B. Benatallah, "Interleaving web services composition and execution using software agents and delegation," in *Proc. WSABE '03*, 2003.

[5] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnamam, and H. Chang, "QoS-aware middleware for web services composition," *IEEE Trans. Softw. Eng.*, vol. 30, no. 5, 2004.

[6] M. Alrifai and T. Risse, "Combining global optimization with local selection for efficient qos-aware service composition," in *Proc. WWW '09*. ACM, 2009, pp. 881–890.

[7] D. Ardagna and B. Pernici, "Adaptive service composition in flexible processes," *IEEE Trans. Softw. Eng.*, vol. 33, no. 6, pp. 369–384, 2007.

[8] G. Canfora, M. Di Penta, R. Esposito, and M. Villani, "A framework for QoS-aware binding and re-binding of composite web services," *J. Syst. Softw.*, vol. 81, no. 10, pp. 1754–1769, 2008.

[9] Q. Liang, X. Wu, and H. C. Lau, "Optimizing service systems based on application-level qos," *IEEE Trans. Serv. Comput.*, vol. 2, pp. 108–121, 2009.

[10] D. A. Menascé, E. Casalicchio, and V. Dubey, "On optimal service selection in service oriented architectures," *Perform. Eval.*, vol. 67, pp. 659–675, Aug. 2010.

[11] T. Yu, Y. Zhang, and K.-J. Lin, "Efficient algorithms for web services selection with end-to-end qos constraints," *ACM Trans. Web*, vol. 1, no. 1, pp. 1–26, 2007.

[12] D. Ardagna and R. Mirandola, "Per-flow optimal service selection for web services based processes," *J. Syst. Softw.*, vol. 83, no. 8, 2010.

[13] V. Cardellini, E. Casalicchio, V. Grassi, F. Lo Presti, and R. Mirandola, "Flow-based service selection for web service composition supporting multiple qos classes," in *Proc. IEEE ICWS '07*, 2007, pp. 743–750.

[14] V. Cardellini, E. Casalicchio, V. Grassi, S. Iannucci, F. Lo Presti, and R. Mirandola, "Moses: A framework for qos driven runtime adaptation of service-oriented systems," *IEEE Trans. Softw. Eng.*, 2011, accepted for publication.

[15] A. Klein, F. Ishikawa, and S. Honiden, "Efficient qos-aware service composition with a probabilistic service selection policy," in *Proc. ICSOC '10*, ser. LNCS, vol. 6470. Springer, Dec. 2010, pp. 182–196.

[16] A. Bellucci, V. Cardellini, V. Di Valerio, and S. Iannucci, "A scalable and highly available brokering service for SLA-based composite services," in *Proc. ICSOC '10*, ser. LNCS, vol. 6470. Springer, Dec. 2010, pp. 527–541.

[17] V. Cardellini, E. Casalicchio, V. Grassi, F. Lo Presti, and R. Mirandola, "QoS-driven runtime adaptation of service oriented architectures," in *Proc. ACM ESEC/SIGSOFT FSE*, 2009, pp. 131–140.

[18] OASIS, "Web Services Business Process Execution Language Version 2.0," Jan. 2007.

[19] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, pp. 1–42, 2009.

[20] "IBM ILOG CPLEX Optimizer," http://www.ibm.com/software/integration/optimization/cplex-optimizer/.

[21] M. Alrifai, D. Skoutas, and T. Risse, "Selecting skyline services for qos-based web service composition," in *Proc. WWW '10*. ACM, 2010, pp. 11–20.