A Performance Comparison of QoS-driven Service Selection Approaches

Valeria Cardellini, Valerio Di Valerio, Vincenzo Grassi, Stefano Iannucci, and Francesco Lo Presti

DISP, Università di Roma "Tor Vergata", Italy
{cardellini,di.valerio,iannucci}@ing.uniroma2.it
{vgrassi,lopresti}@info.uniroma2.it

Abstract. Service selection has been widely investigated as an effective adaptation mechanism that allows a service broker, offering a composite service, to bind each task of the abstract composition to a corresponding implementation, selecting it from a set of candidates. The selection aims typically to fulfill the Quality of Service (QoS) requirements of the composite service, considering several QoS parameters in the decision. We compare the performance of two representative examples of the per-request and per-flow approaches that address the service selection issue at a different granularity level. We present experimental results obtained with a prototype implementation of a service broker. Our results show the ability of the per-flow approach in sustaining an increasing traffic of requests, while the per-request approach appears more suitable to offer a finer customizable service selection in a lightly loaded system.

1 Introduction

A major trend to tackle the increasing complexity of service-oriented systems (SOSs) is to design them as runtime self-adaptable systems, so that they can operate in highly changing and evolving environments. The introduction of self-adaptation allows a system offering a composite service to meet both functional requirements, concerning the overall logic to be implemented, and non functional requirements, concerning the quality of service (QoS) levels that should be guaranteed to its user. The adaptation in a SOS may take place at two different levels. At the *horizontal* level, the adaptation involves mainly the *service selection*, that determines the binding of each task in the composite service to actual implementations, leaving unchanged the composition logic, while at the *vertical* level the composition logic can be altered [7].

In this paper, we focus on the adaptation at the horizontal level and consider the granularity level at which the adaptation can be performed. With the *per-request* grain, the adaptation concerns a single request addressed to a composite service, and aims at making the system able to fulfill the QoS requirements of

* The original publication is available at http://www.springerlink.com/ in *Towards a Service-based Internet*, LNCS Vol. 6994, pp. 167-178, 2011.

that request, independently of the concurrent requests that may be addressed to the system. With the *per-flow* grain, the adaptation concerns an overall flow of requests, and aims at fulfilling QoS requirements concerning the global properties of that flow.

In this paper, we compare the performance of the per-flow and per-request approaches considering a service broker that offers a composite service to prospective users having differentiated QoS requirements. To this end, we consider two representative methodologies that tackle the service selection at the per-request and per-flow grains and incorporate them into the MOSES (MOdel-based SElf-adaptation of SOA systems) prototype [4], a runtime adaptation framework for a SOS architected as a service broker. We compare the performance of the two methodologies under two workload scenarios characterized by different workload patterns, considering as main performance metric the fulfillment of the composite service's response time agreed by the broker with its users.

Most of the proposed methodologies for service selection focus on the *per-request* case (*e.g.*, [1, 2, 5, 8, 9, 11, 12]) and have been formalized as optimization problems. Zeng et al. [12] present a global planning approach based on integer programming. Ardagna and Pernici [2] model the service composition as a mixed integer linear problem and their technique is particularly efficient for large process instances. Alrifai and Risse [1] combine global optimization with local selection techniques to reduce the optimization complexity. Canfora et al. [5] follow a quite different strategy based on genetic algorithms. Since the per-request service selection problem is NP-hard, heuristic algorithms have been proposed, e.g., [8, 9, 11]. For the per-request approach we focus on the methodology in [2], which is one of the top performing state-of-the-art approaches.

A few works have focused on the per-flow granularity. Beside the proposal in [6], that we use as representative case of the per-flow approach and takes the form of a linear problem, a per-flow methodology is in [3], where service selection is based on a constrained non-linear optimization problem. The work in [3] is also, until now, the only comparison between the per-flow technique therein presented and the per-request proposals in [1, 2]; however, the performance comparison in [3] concerns only the optimization time reduction due to the different problem formulations and is conducted through simulation. On the other hand, in this paper we compare the per-flow and per-request approaches plugging them into the MOSES prototype, thus analyzing their impact on the overall performance of a real service-oriented system.

The paper is organized as follows. In Sect. 2 we analyze the per-request and per-flow service selection approaches. In Sect. 3 we provide an overview of the MOSES system. In Sect. 4 we present the MOSES-based experiments to compare the performance and effectiveness of the two approaches. We conclude in Sect. 5.

2 QoS-driven Service Selection Approaches

We consider a service broker, which offers to prospective users a composite service with a range of different service classes, which imply different QoS levels and

monetary prices, exploiting for this purpose a set of existing concrete services. The broker acts as a full intermediary between users and concrete services, performing a role of service provider towards the users and being in turn a requestor to the concrete services used to implement the composite service. 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 while optimizing a suitable broker utility function, *i.e.*, response time or cost. Within this framework, one of the main broker tasks is to determine a service selection that fulfills the SLAs it negotiates with its requestors, given the SLAs it has negotiated with the providers. The selection criteria correspond to the optimization of a given utility goal of the broker.

In this section we present the per-request and the per-flow approaches to service selection, following the formulations presented in [2] and [6], respectively.

Let us denote by $\mathcal{S}$ the set of abstract tasks that compose the composite process P offered by the broker, where $S_i \in \mathcal{S}$, $i = 1, \ldots, m$, represents a single task, being m the number of tasks composing P. Figure 1(a) shows an example of business process workflow. For each task S_i, we assume that the broker has identified a pool $\Im_i = \{cs_{ij}\}$ of candidate concrete services implementing it.

For each candidate service, the broker negotiates a SLA with its provider, establishing the values of the QoS attributes provided by each concrete service in correspondence with a mean volume of requests generated by the broker for that service. Then, the broker may negotiate a SLA with each requestor, establishing the offered QoS level of the composite service. We consider the following subset

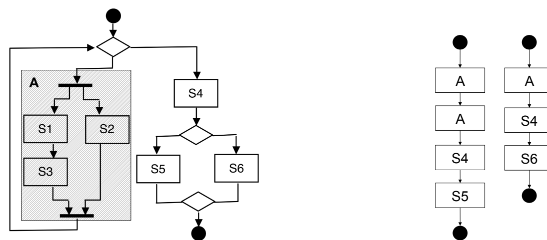


Fig. 1. Example of workflow (left) and execution paths (right)

of representative QoS attributes:

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

Our general model for the SLA between the composite service users and the service broker (acting the provider role) consists of a tuple $\langle R_{max}, A_{min}, C_{max}, L \rangle$, where: R_{max} is the upper bound on the service response time, A_{min} is the lower bound on the service availability, C_{max} is the upper bound on the service cost

per invocation. The provider can also specify the additional parameter L, that indicates that performance thresholds R_{max} and A_{min} will hold provided that the request rate generated by the users does not exceed the load threshold L.

The broker (acting the user role) negotiates and defines SLAs with the providers of the concrete services. For each $cs_{ij} \in \Im_i$, we denote with the tuple $\langle r_{ij}, a_{ij}, c_{ij}, l_{ij} \rangle$ the corresponding SLA, whose parameters have the same meaning of the SLAs negotiated by the broker with the composite service users.

The SLAs stipulated in the per-request and per-flow approaches differ in two aspects. The first one regards the granularity level at which the SLAs with the composite service users are managed by the service broker. In the per-request approach, the broker tries to meet the QoS constraints for *each individual request* submitted to the composite service, irrespective of whether it belongs to some flow generated by one or more users, and taking into account the *worst case* (*i.e.*, the maximum number of iterations in a loop and different branches). On the other hand, in the per-flow approach the service level objectives stated in the SLA concern the average value of the QoS attributes calculated over all the requests pertaining to the *flow of requests* generated by a given user. In the per-flow formulation in [6] the analysis focused on the *average case* rather than the worst one. To compare the two approaches, in this paper we modify the original formulation in [6], so that the per-flow approach takes the worst case into consideration (specifically, the maximum number of invocations to each abstract task rather than the average number). The second difference about the SLAs in the two approaches regards the load threshold, which is not contemplated by the per-request approach. As we will see in Sec. 4, this limits the applicability of the per-request approach, which hardly scales with workload increases.

2.1 Per-request Approach

In the per-request approach we need to identify the concrete service to be bound to each abstract service for all execution paths [2]. The per-request optimization problem is formulated as a Mixed Integer Linear Programming (MILP) problem. We denote with the vector $\boldsymbol{x} = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m]$ the optimal policy for a request to the composite service, where each entry $\boldsymbol{x}_i = [x_{ij}]$, $x_{ij} \in \{0, 1\}$, $i \in \mathcal{S}$, $j \in \Im_i$, denotes the adaptation policy for task S_i and the constraint $\sum_{j \in \Im_i} x_{ij} = 1$ holds. That is, x_{ij} is the decision variable equal to 1 if task S_i is implemented by concrete service cs_{ij}, 0 otherwise. Assume that the per-request policy $\boldsymbol{x}$ determines that for a given request $\boldsymbol{x}_i = [0, 0, 1, 0]$. According to this policy, for S_i the broker binds the request to cs_{i3}.

Following the per-request strategy in [2], we need to consider all the possible *execution paths* derived from the workflow. An *execution path* ep_n is a set 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. An execution path may also contain parallel sequences but it does not contain loops, which are *peeled* (see Fig. 1(b) for two execution paths derived from the workflow of Fig. 1(a)). A probability of execution $freq_n$ is associated with every execution path and can be evaluated as the product of the probabilities

of executing the branch conditions included in the path. Branch conditions that arise from loop peeling produce other execution paths. Therefore, the set of all the execution paths identifies all the possible execution scenarios of the process.

The general goal of the optimization problem is to maximize the aggregated QoS value, considering all of the possible execution scenarios, *i.e.*, all the execution paths arising from the business process. For simplicity's sake, in the formulation below we consider that the service broker's goal is to minimize for each request the response time of the composite service it offers.

$$\text{Problem } \textbf{per-request: min} \sum_{ep_n} freq_n * R_n(\boldsymbol{x})$$

$$\textbf{subject to: } R_n(\boldsymbol{x}) \leq R_{max} \quad \forall ep_n \tag{1}$$

$$\log A_n(\boldsymbol{x}) \geq \log A_{min} \quad \forall ep_n \tag{2}$$

$$C_n(\boldsymbol{x}) \leq C_{max} \quad \forall ep_n \tag{3}$$

$$x_{ij} \in \{0,1\} \;\; \forall j \in \Im_i, \sum_{j \in \Im_i} x_{ij} = 1 \quad \forall i \in \mathcal{S} \tag{4}$$

$R_n(\boldsymbol{x})$, $A_n(\boldsymbol{x})$ and $C_n(\boldsymbol{x})$ denote the response time, availability, and cost of the execution path ep_n. We note that the minimization of the response time is only one of the possible objective functions that can be used, depending on the utility goal of the broker. An alternative expression can be found in [2], where the objective function is formulated using the weighted z-scores of QoS attributes.

2.2 Per-flow Approach

While in the per-request approach the optimization problem atomically considers a single request, in the per-flow it is assumed to have a set K of service classes, with $k \in K \subseteq \mathbb{N}$, for each business process P. Hence, the SLA with each user u of a class $k \in K$ is defined as a tuple $\langle R^k_{max}, A^k_{min}, C^k_{max}, L^k_u \rangle$. The optimization problem takes simultaneously into account the overall flow of requests belonging to the service classes. Anyway, the granularity level of the service classes may be arbitrarly fine, so that each user could have its own service class.

In the per-flow approach we need to identify the concrete service to be bound to each abstract service for all the service class. For each class k, we denote with the vector $\boldsymbol{x}^k = [\boldsymbol{x}^k_1, \ldots, \boldsymbol{x}^k_m]$ the optimal policy, where each entry $\boldsymbol{x}^k_i = [x^k_{ij}]$, $0 \leq x_{ij} \leq 1$, $i \in \mathcal{S}$, $j \in \Im_i$, denotes the adaptation policy for task S_i and the constraint $\sum_{j \in \Im_i} x^k_{ij} = 1$ holds. That is, the policy define a probabilistic binding between S_i and its implementation in $\Im_i$, whereby each entry x^k_{ij} of $\boldsymbol{x}^k_i$ denotes the probability that the class-k request will be bound to concrete service cs_{ij}. As an example, consider the case $\Im_i = \{cs_{i1}, cs_{i2}, cs_{i3}, cs_{i4}\}$ for task S_i. Assume that the per-flow policy $\boldsymbol{x}$ determines that for a given class k $\boldsymbol{x}^k_i = [0, 0.2, 0.5, 0.3]$. According to this policy, given a class-k request for S_i, the broker binds the request: with probability 0.2 to cs_{i2}, 0.5 to cs_{i3}, and 0.3 to cs_{i4}.

The per-flow approach is formulated as a Linear Programming (LP) problem, and therefore its computational cost is lower than the alternative approach. As

in the per-request formulation, we consider the minimization of the response time, but in this case the latter regards the aggregated flow of requests.

$$\text{Problem } \textbf{per-flow: min} \sum_{k \in K} L^k R^k(L, \boldsymbol{x})$$

$$\textbf{subject to: } R^k(L, \boldsymbol{x}) \leq R^k_{max} \quad \forall k \in K \tag{5}$$

$$\log A^k(L, \boldsymbol{x}) \geq \log A^k_{min} \quad \forall k \in K \tag{6}$$

$$C^k(L, \boldsymbol{x}) \leq C^k_{max} \quad \forall k \in K \tag{7}$$

$$\sum_{k \in K} x^k_{ij} V^k_{\alpha,i} L^k \leq l_{ij} \quad \forall j \in \Im_i, \forall i \in \mathcal{S} \tag{8}$$

$$x^k_{ij} \geq 0 \quad \forall j \in \Im_i, \sum_{j \in \Im_i} x^k_{ij} = 1 \quad \forall i \in \mathcal{S} \tag{9}$$

where: $L = [L^k]_{k \in K}$ and $L^k = \sum_u L^k_u$ is the aggregated class-k users service request rate (being u a user); $R^k(L, \boldsymbol{x})$, $A^k(L, \boldsymbol{x})$, and $C^k(L, \boldsymbol{x})$ the class-k response time, availability, and cost, respectively, under the adaptation policy $\boldsymbol{x} = [\boldsymbol{x}^k]_{k \in K}$. Their expression requires knowledge of $V^k_{\alpha,i}$, which is the α-quantile of the number of times S_i is invoked by class-k requests: for further details we refer the reader to [6]. Here (8) represents the request load assigned to each concrete service and ensures that the load does not exceed the volume of invocations l_{ij} agreed with the service providers. As in the per-request approach, the minimization of the response time is just a possible utility goal of the broker.

3 MOSES System

MOSES is a QoS-driven runtime adaptation framework for service-oriented systems, intended to act as a *service broker* and designed with a flexible and modular system architecture. In the following, we provide an overview of the MOSES system; a detailed description of the per-flow methodology (for whom MOSES has been originally designed) and prototype can be found in [6] and [4], respectively.

We first describe the core MOSES modules and then the remaining ones that enrich the basic functionalities. The *Optimization Engine* computes the optimal solution that drives the runtime binding according to the two alternative approaches in Sect. 2. To achieve a flexible implementation, the Optimization Engine exposes the same interface to the other MOSES modules irrespectively of the specific approach. The *BPEL Engine* executes the business process, described in BPEL, that defines the user-relevant business logic. Finally, 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 functionality is to dynamically bind each abstract task's invocation to the real endpoint identified by the Optimization Engine.

The main execution sequence for a composite service request managed by MOSES differs according to the service selection approach. With the per-request one, every core module is involved in the execution, as depicted in Fig. 2(a): the

user issues a process invocation to the BPEL Engine which, in turn, requests to the Optimization Engine the optimization problem solution, considering the specific SLA parameters agreed with the user for that request. The optimal solution, that encompasses all the abstract tasks that will be invoked during the request execution, is kept in the Storage layer, so that it can be retrieved for each abstract task to concrete implementation binding that occurs during the processing of that request. When the BPEL Engine reaches an `invoke` activity, it contacts the Adaptation Manager, which retrieves the needed runtime binding information from the Storage and invokes the selected concrete service.

The per-flow approach follows a different pattern: the optimization problem solution is not computed synchronously at the receipt of every request for the composite service, but rather only for the flow to whom that request belongs and only when some monitoring module determines its need to react to some change occurred in the MOSES environment. The corresponding sequence diagram is thus simplified, because it does not include the gray shaded box in Fig. 2(a).

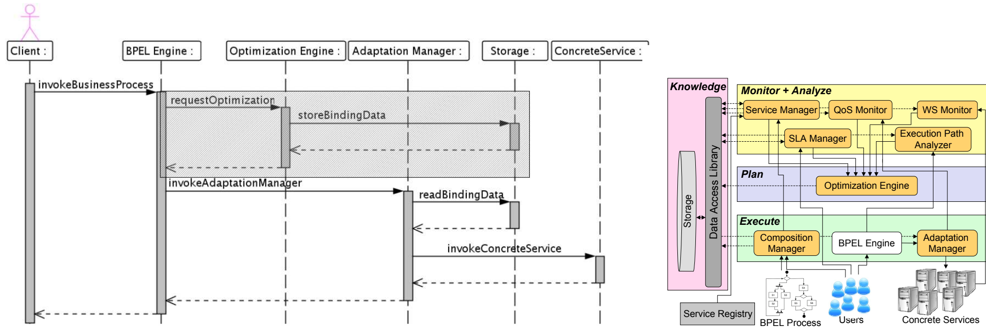


Fig. 2. MOSES system: request execution flow (left) and high-level architecture (right)

In a system subject to a quite sustained request rate, performing a per-request solution of the optimization problem could cause an excessive computational load, especially for a large-scale optimization problem, being the problem formulated as MILP. To mitigate this issue, we have improved the per-request execution sequence by introducing the caching of each calculated solution of the optimization problem corresponding to a given instance of the system model. Therefore, if a request matches with a cached solution (in terms of SLA and system parameters), similarly to the per-flow approach, the binding is retrieved from the Storage layer without involving 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 [10]. Figure 2(b) shows how the MOSES modules implement each MAPE-K macro-component, together with the system inputs (*i.e.*, the business process and the set of concrete services). This input is used to build a model (Execute), which is kept up-to-date at runtime (Monitor). The monitored parameters are

analyzed (Analyze) in order to know if adaptation actions have to be taken; if needed, a new adaptation policy is calculated (Plan).

The modules in the Monitor+Analyze macro-component capture changes in the MOSES environment and, if they are relevant, modify at runtime the stored system model and trigger the Optimization Engine. The *Service Manager* and *WS Monitor* detect the addition or removal of concrete services, respectively. The *QoS Monitor* detects violations of the service level objectives stated in the SLAs between MOSES and the service providers. The *Execution Path Analyzer* tracks variations in the usage profile of the abstract tasks. The *SLA Manager* manages the arrival/departure of a user with the associated SLA, eventually performing a contract admission control.

4 Experimental Comparison

In this section, we present the experimental analysis we have conducted using the MOSES prototype to compare the per-flow and per-request approaches.

4.1 Experimental Setup

The MOSES prototype is based on the Java Business Integration (JBI) implementation called OpenESB and the relational database MySQL. We use Sun BPEL Service Engine for the business process logic, and MATLAB and CPLEX to solve respectively the per-flow and per-request optimization problems. We refer to [4] for a detailed description of the MOSES prototype.

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 hosted 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 hosted the workload generator. We

Table 1. SLA parameters for concrete services (left) and service classes (right)

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	r_{ij}	a_{ij}	c_{ij}
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_{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	r_{ij}	a_{ij}	c_{ij}
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	14	0.9	39
2	17	0.88	35
3	19	0.86	32
4	22	0.84	29

consider the workflow of Fig. 1(a), 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 Tab. 1(left), differ in terms of cost c_{ij}, availability a_{ij}, and response time r_{ij} (in sec). The concrete services are simple stubs; however, their non-functional behavior conforms to the guaranteed levels expressed in their SLA. The perceived response time is obtained by modeling each service as a M/G/1/PS queue implemented inside the Web service deployed in the Tomcat container. For all concrete services the load threshold l_{ij} is equal to 10 req/sec and the response time knee is beyond it.

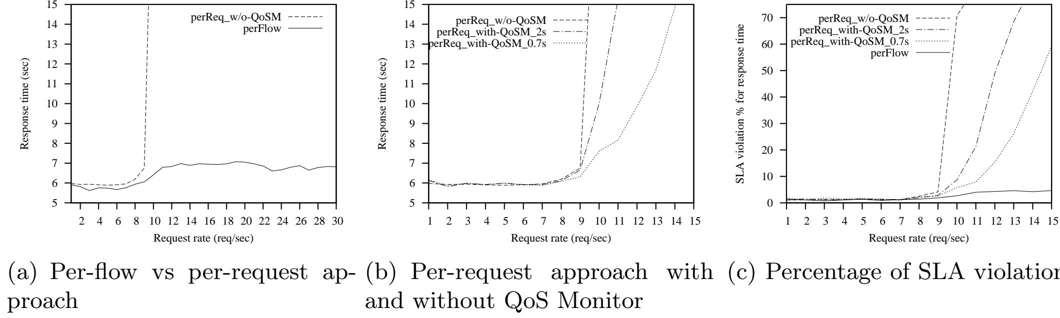
On the user side, we assume a scenario with four classes of the composite service managed by MOSES. The SLAs negotiated by the users are characterized by a range of QoS requirements as listed in Tab. 1(right), with users in class 1 having the most stringent performance requirements (being willing to pay the highest cost) and users in class 4 the least stringent ones (being willing to save money). The usage profile of the service classes is given by the following values for the maximum number of service invocations: $V^k_{\alpha,1} = V^k_{\alpha,2} = V^k_{\alpha,3} = 3$, $V^k_{\alpha,4} = 1$, $k \in K$; $V^k_{\alpha,5} = 0.7$, $V^k_{\alpha,6} = 0.3$, $k \in \{1, 3, 4\}$; $V^2_{\alpha,5} = V^2_{\alpha,6} = 0.5$, being $\alpha = 0.96$.

To issue requests to the composite service managed by MOSES we have developed a workload generator in C language using the Pthreads library. It mimics the behavior of users that establish SLA contracts before accessing the composite service. For the per-flow approach, upon the arrival of a new contract there is a preliminary invocation to the SLA Manager for the admission control: a new contract is accepted if the **per-flow** problem can be solved given the SLA requested by the new user and the SLAs agreed by MOSES with its currently admitted users. On the other hand, for the per-request approach there is no admission control, because each request is treated independently of other concurrent requests. Once its SLA contract has been accepted, the user u starts issuing requests to the composite service at a rate L^k_u until the contract ends.

4.2 Experimental Results

To compare the per-flow and per-request service selection approaches, we consider two different workload scenarios. In the *first scenario*, we consider each service class per time (*i.e.*, in a specific experiment the requests pertain only to one of the service classes in Table 1(right)) and we stress the MOSES system by progressively increasing the request rate. To this end, we set for all the contracts a fixed duration equal to 100 sec and L^k_u=1 req/sec, while the contract interarrival rate ranges from 0.01 to 0.3 contr/sec for each step of the overall experiment: this setting corresponds to an overall request arrival rate L^k from 1 to 30 req/sec. Each single step (corresponding to a given request rate) lasts 15 minutes. At each step, to avoid overwhelming a just started GlassFish instance, which has a significant setup time, the workload generator does not immediately issue requests at the required request rate but within a ramp (set to 100 sec), during which the request rate is linearly incremented until it reaches the desired value.

For space reasons we focus our analysis on the most sensitive SLA parameter to the workload increase, *i.e.*, the response time, obtained by the requests of class 1, which has the most stringent SLA requirements. Figure 3(a) shows the re-



(a) Per-flow vs per-request approach

(b) Per-request approach with and without QoS Monitor

(c) Percentage of SLA violation

Fig. 3. Scenario 1: response time of the composite service for class 1

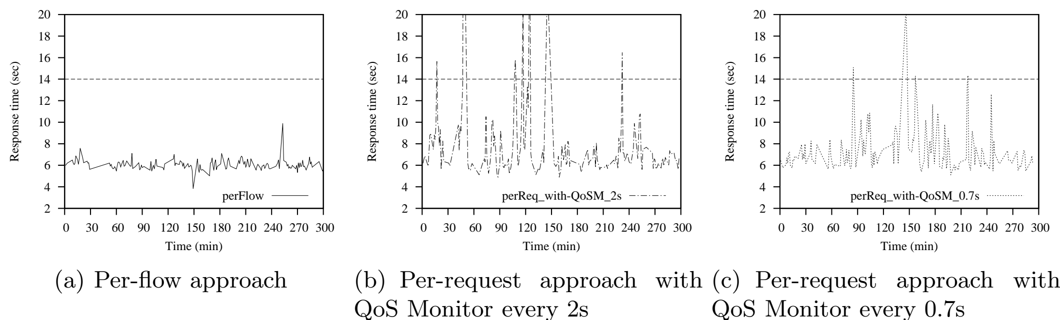
sponse time of the composite service achieved by the two alternative approaches for an increasing request rate and with the MOSES monitoring modules disabled (except the SLA Manager). We observe that while the per-flow response time remains well below the agreed SLA value (equal to 14 sec), for the per-request approach (denoted by perReq_w/o-QoSM) the response time increases exponentially approximately at the concrete services' load threshold (set to 10 req/sec). In a lightly loaded system, the per-request approach is effective to address the adaptation to each single request. However, when the workload increases, it incurs in stability and management problems, since it takes adaptation actions just for a single request, independently of the other concurrent requests. Therefore, the concrete services identified as the best ones by the per-request deterministic policy are overwhelmed by the requests. On the other hand, the probabilistic per-flow policy chooses the best implementations only until their load threshold is not exceeded (see (8) of the **per-flow** problem); at that point, it distributes the requests among a subset of (possibly all) the available concrete services. This behavior is evident in Fig. 3(a), where the response time increases from around 6 to 7 sec at the concrete services' load threshold. The stable behavior of the per-flow approach is counterbalanced by an amount of dropped SLA contracts; the rejection percentage ranges from 7% (for 12 req/sec) to 59% (for 30 req/sec).

To improve the performance of the per-request approach, we activate the QoS Monitor, so that after a SLA violation the agreed values of the concrete services' parameters are updated in the system model with the measured values and the triggered Optimization Engine calculates a new solution of the **per-request** problem. The SLA violation is detected when the data monitored during one time window exceed by 20% the SLA agreed by MOSES with the service providers. We can see in Fig. 3(b) that the monitoring activity and the subsequent reaction improve the per-request behavior: when the best implementation for a given task

becomes overloaded, the requests are shifted towards another concrete service determined by the new adaptation policy. However, the improvement is achieved at a cost of having a very reactive system, characterized by a quite frequent monitoring activity because the monitored data are analyzed either to 2 or even 0.7 sec, denoted by perReq_withQoSM_2s and perReq_withQoSM_0.7s in Fig. 3(b).

Let us now consider how in the first scenario the SLA is satisfied: Fig 3(c) shows the percentage of violations for the response time agreed with the users. While under the per-flow approach only few requests suffer from a SLA violation, the percentage dramatically increases for the per-request service selection, even with a frequent monitoring activity.

In the *second scenario* we consider a mixed workload in which MOSES offers simultaneously the composite service to all the service classes in Table 1(right). We assume exponential distributions of parameters λ_k and $1/d_k$ for the contract inter-arrival time and duration and a Gaussian distribution of parameters (μ_k, σ_k) for the request inter-arrival L_u^k. Each user u generates its requests to the composite service according to an exponential distribution with parameter L_u^k. The values of the workload model parameters are $d_k = 100$ and $(\mu_k, \sigma_k) = (3, 1)$ $\forall k$; λ_k, d_k, and μ_k values have been set so that for Little's formula $L_k = \lambda_k \mu_k d_k$ and therefore on average $L = (L^k) = (1.5, 1, 3, 1)$. For space reason, we analyze



(a) Per-flow approach (b) Per-request approach with QoS Monitor every 2s (c) Per-request approach with QoS Monitor every 0.7s

Fig. 4. Scenario 2: response time of the composite service over time for class 1

only how the response time of the composite service varies over time for the most demanding class 1, as shown in Figs. 4(a)-4(c) (the horizontal line is the agreed response time, as reported in Tab. 1). Although in the second scenario the system is only subject to a moderate workload intensity (the average overall request rate is 6,37 req/sec, being 20,4 req/sec the peak and 4,29 req/sec the standard deviation), we find that the response time level achieved by the per-flow approach has a much more stable trend and does not suffer from the SLA violations of the per-request service selection. The percentage of dropped contracts by the per-flow approach is 12%.

5 Conclusions

In this paper, we have compared the per-flow and per-request approaches to address the service selection issue for a service broker which offers a composite service with different QoS levels. Our results show that in a lightly loaded system, it is effective to tailor the service selection to each single request, independently of other concurrent requests, to customize the system with respect to that single request. On the other hand, in a system subject to a quite sustained flow of requests, performing a per-request selection could incur in stability problems, since the "local" decisions could conflict with the decisions independently determined for other concurrent requests. Furthermore, the solution of the per-request problem at a frequent rate could cause an excessive computational load due to its MILP formulation. In the latter scenario, the per-flow approach is likely to be more effective, even if it loses the potentially finer customization features of the per-request approach and can drop SLA contracts when there are not enough system resources. We plan to extend the performance comparison to other representative proposals for service selection and to address the lack of a "global" system view that currently affects the per-request approach.

References

1. Alrifai, M., Risse, T.: Combining global optimization with local selection for efficient qos-aware service composition. In: Proc. WWW '09. pp. 881–890 (2009)
2. Ardagna, D., Pernici, B.: Adaptive service composition in flexible processes. IEEE Trans. Softw. Eng. 33(6), 369–384 (2007)
3. Ardagna, D., Mirandola, R.: Per-flow optimal service selection for web services based processes. J. Syst. Softw. 83(8) (2010)
4. Bellucci, A., Cardellini, V., Di Valerio, V., Iannucci, S.: A scalable and highly available brokering service for SLA-based composite services. In: Proc. ICSOC '10. LNCS, vol. 6470, pp. 527–541. Springer (Dec 2010)
5. Canfora, G., Di Penta, M., Esposito, R., Villani, M.: A framework for QoS-aware binding and re-binding of composite web services. J. Syst. Softw. 81(10) (2008)
6. Cardellini, V., Casalicchio, E., Grassi, V., Lo Presti, F., Mirandola, R.: Flow-based service selection for web service composition supporting multiple qos classes. In: Proc. IEEE ICWS '07. pp. 743–750 (2007)
7. Hassine, A.B., Matsubara, S., Ishida, T.: A constraint-based approach to horizontal web. In: Proc. ISWC '06. pp. 130–143 (2006)
8. Liang, Q., Wu, X., Lau, H.C.: Optimizing service systems based on application-level QoS. IEEE Trans. Serv. Comput. 2, 108–121 (2009)
9. Menascé, D.A., Casalicchio, E., Dubey, V.: On optimal service selection in service oriented architectures. Perform. Eval. 67, 659–675 (Aug 2010)
10. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Trans. Auton. Adapt. Syst. 4(2), 1–42 (2009)
11. 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)
12. Zeng, L., Benatallah, B., Dumas, M., Kalagnamam, J., Chang, H.: QoS-aware middleware for web services composition. IEEE Trans. Softw. Eng. 30(5) (2004)