Real-Time QoS Control for Service Orchestration

Joost Bosman
CWI
Dept. of Stochastics
Amsterdam, The Netherlands

Hans van den Berg
TNO, Dept. Performance
of Network and Systems
The Hague, The Netherlands

University of Twente,
Dept. Computer Science
Enschede, The Netherlands

Rob van der Mei
CWI
Dept. of Stochastics
Amsterdam, The Netherlands

VU University Amsterdam
Dept. of Mathematics
Amsterdam, The Netherlands

***Abstract*—Service orchestration has become the predominant paradigm that enables businesses to combine and integrate services offered by third parties. For the commercial viability of orchestrated services, it is crucial that they are offered at sharp price-quality ratios. A complicating factor is that many attractive third-party services often show highly variable service quality. This raises the need for mechanisms that promptly adapt the orchestration to changes in the quality delivered by third party services.**

In this paper, we propose a real-time QoS control mechanism that dynamically optimizes service orchestration in real time by learning and adapting to changes in third party service response time behaviors. Our approach combines the power of learning and adaptation with the power of dynamic programming. The results show that real-time service re-compositions lead to dramatic savings of cost, while meeting the service quality requirements of the end-users. The challenge here is to respond to significant response-time changes in a timely manner, while not wasting CPU cycles on unnecessary orchestration updates. Experimental results performed in a test-lab environment demonstrate that a few orchestration updates are sufficient to achieve this.

I. Introduction

The evolution towards a service-based economy is currently boosted by the developments in information and communication technology and expected to influence our lifestyle in the future. In particular, over the past years, service-orchestration have become a predominant paradigm among businesses for enabling more efficient and flexible business processes, addressing some of the technological challenges posed by the service-based economy. Service orchestration enables the development of composite services that aggregate services offered by third parties. The emergence of these service chains has opened up tremendous possibilities for creation of new services in many application areas, including for example banking, health, entertainment and travel.

In the competitive market of information and communication services, it is crucial for service providers to be able to offer services at competitive price/quality ratios. Succeeding to do so will attract customers and generate business, while failing to do so will inevitably lead to customer dissatisfaction, churn and loss of business. A complicating factor in controlling quality-of-service (QoS) in service oriented architectures is that the ownership of the services in the composition (sub-services) is decentralized: a composite service makes use of sub-services offered by third parties, each with their own business incentives. As a consequence, the QoS experienced by the (paying) end user of a composite service depends heavily on the QoS levels realized by the individual sub-services running on different underlying platforms with different performance characteristics: a badly performing sub-service may strongly degrade the end-to-end QoS of a composite service. In practice, service providers tend to outsource responsibilities by negotiating Service Level Agreements (SLAs) with third parties. However, negotiating multiple SLAs in itself is not sufficient to guarantee end-to-end QoS levels as SLAs in practice often give probabilistic QoS guarantees and SLA violations can still occur. Moreover probabilistic QoS guarantees do not necessarily capture time-dependent behavior e.g. short term service degradations. Therefore, the negotiation of SLAs needs to be supplemented with *run-time QoS-control* capabilities that give providers of composite services the capability to properly respond to short-term QoS degradations (real-time composite service adaptation). Motivated by this we developed an approach that adapts to (temporary) third party QoS degradations by tracking the response time behavior of these third party services.

In order to demonstrate the effectiveness of our approach we have developed an experimental test-lab environment (depicted in Figure 3 in Section VI). The experimental environment includes an actual HTTP implementation and is able to concurrently orchestrate service requests.

The remainder of this paper is as follows. In Section II we explain our service selection orchestration model. Section III elaborates on literature and related work. We start in section IV with the introduction of our real-time QoS control approach. Next in Section V we introduce the tools for handling empirical distributions. Using the empirical distributions we define a dynamic program that optimizes expected profit. Experiments on our approach are performed in Section VI. Results are

presented and discussed in Section VII. Finally we conclude in Section VIII.

II. Orchestration Model

We consider a composite service that comprises a sequential workflow consisting of N tasks identified by $T_1, \ldots, T_N$. The tasks are executed one–by–one in the sense that each consecutive task has to wait for the previous task to finish. Our solution is applicable to any workflow that could be aggregated and mapped into a sequential one. Basic rules for aggregation of non–sequential workflows into sequential workflows have been illustrated in, e.g. [14], [13], [5]. However, the aggregation leads to coarser control, since decisions could not be taken for a single service within the aggregated workflow, but rather for the aggregated workflow patterns themselves.

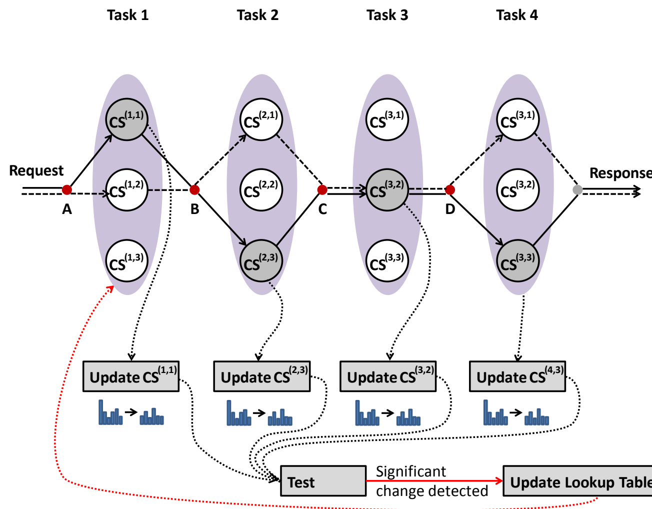

Figure 1: Orchestrated composite web service depicted by a sequential workflow. Dynamic run–time service composition is based on a lookup table. Decisions are taken at points A–D. For every used concrete service the response-time distribution is updated with the new realization. In this example a significant change is detected. As a result for the next request concrete service 2 is selected at task 1.

The workflow is based on an unambiguous functionality description of a service (“abstract service”), and several functionally identical alternatives (“concrete services”) may exist that match such a description [10]. Each task has an abstract service description or interface which can be implemented by external service providers.

The workflow in Fig. 1 consists of four abstract tasks, and each task maps to three concrete services (alternatives), which are deployed by (independent) third–party service providers. For each task T_i there are M_i concrete service providers $\text{CS}^{(i,1)}, \ldots, \text{CS}^{(i,M_i)}$ available that implement the functionality corresponding to task T_i. For each request processed by $\text{CS}^{(i,j)}$ cost $c^{(i,j)}$ has to be paid. Furthermore there is an end–to–end response-time deadline δ_p. If a request is processed within δ_p a reward of R is received. However, for all requests that are not processed within δ_p a penalty V had to be paid. After the execution of a single task within the workflow, the orchestrator decides on the next concrete service to be executed, and composite service provider pays to the third party provider per single invocation. The decision points for given tasks are illustrated at Fig. 1 by A, B, C and D. The decision taken is based on (1) execution costs, and (2) the remaining time to meet the end–to–end deadline. The response time of each concrete service provider $\text{CS}^{(i,j)}$ is represented by the random variable $D^{(i,j)}$. After each decision the observed response time is used for updating the response time distribution information of the selected service. Upon each lookup table update the corresponding distribution information is stored as reference distribution. After each response the reference distribution is compared against the current up-to date response time distribution information.

In our approach response-time realizations are used for learning an updating the response-time distributions. The currently known response-time distribution is compared against the response-time distribution that was used for the last policy update. Using well known statistical tests we are able to identify if an significant change occurred and the policy has to be recalculated. Our approach is based on fully dynamic, run–time service selection and composition, taking into account the response–time commitments from service providers and information from response-time realizations. The main goal of this run–time service selection and composition is profit maximization for the composite service provider and ability to adapt to changes in response-time behavior of third party services.

By tracking response times the actual response-time behavior can be captured in empirical distributions. In [14] we apply a dynamic programming (DP) approach in order to derive a service-selection policy based on response-time realizations. With this approach it is assumed that the response-time distributions are known or derived from historical data. This results in a so called lookup table which determines what third party alternative should be used based on actual response-time realizations.

III. Literature and Related Work

We extend our approach in [14] such that we can learn an exploit response-time distributions on the fly. The use of classical reinforcement-learning techniques would be a straight forward approach. However, our model has a special structure that complicates the use of the classical Temporal Difference learning (TD) learning approaches. The solution of our DP formulation searches the stochastic shortest path in a stochastic activity network [5]. This DP can be characterized as a hierarchical DP [8], [1]. Therefore classical Reinforcement

Learning (RL) is not suitable and hierarchical RL has to be applied [1]. Also changes in response-time behavior are likely to occur which complicates the problem even more. Both the problem structure and volatility are challenging areas of research in RL. Typically RL techniques solve complex learning and optimization problems by using a simulator. This involves a Q value that assigns utility to state–action combinations. Most algorithms run off-line as a simulator is used for optimization. RL has also been widely used in on–line applications. In such applications, information becomes available gradually with time. Most RL approaches are based on environments that do not vary over time. We refer to [8] for a good survey on reinforcement learning techniques.

In our approach we tackle both the hierarchical structure, and time varying behavior challenges. To this end we are using empirical distributions and updating the lookup table if significant changes occur. As we are considering a sequence of tasks, the number of possible response time realizations combinations explodes. By discretizing the empirical distribution over fixed intervals we overcome this issue.

In the literature the problem of QoS–aware optimal service composition of Service Oriented Architecture (SOA) services has been well–studied (see e.g. [2], [11]). The main problem addressed in these papers is how to select one concrete service per abstract service for a given workflow, in such a way that the QoS of the composite service (as expressed by the respective SLA) is guaranteed, while optimizing some cost function. Once established, this composition would remain unchanged the entire life–cycle of the composite web service. In reality, SLA violations occur relatively often, leading to providers' losses and customer dissatisfaction. To overcome this issue, it is suggested in [3], [12], [9] that, based on observations of the actually realised performance, re–composition of the service may be triggered. During the re–composition phase, new concrete service(s) may be chosen for the given workflow. Once re–composition phase is over, the (new) composition is used as long as there are no further SLA violations. In particular, the authors of [3], [12], [9] describe *when* to trigger such (re–composition) event, and *which adaptation actions* may be used to improve overall performance.

A number of solutions have been proposed for the problem of *dynamic, run–time* QoS–aware service selection and composition within SOA, [4], [7], [14], [6]. These (proactive) solutions aim to adapt the service composition dynamically at run–time. However, these papers do not consider the stochastic nature of response time, but its expected value. Or they do not consider the cost structure, revenue and penalty model as given in this paper.

IV. Real time QoS control

In this section we explain our real-time QoS control approach. The main goal of this approach is profit maximization for the composite service provider, and ability to adapt to changes in response-time behavior of third party services. We realize this by monitoring/tracking the observed response-time realizations. The currently known empirical response-time distribution is compared against the response-time distribution that was used for the last policy update. Using well known statistical tests we are able to identify if an significant change occurred and the policy has to be recalculated. Our approach is based on fully dynamic, run–time service selection and composition, taking into account the response–time commitments from service providers and information from response-time realizations. We illustrate our approach using Figure 2. The execution starts with an initial lookup table at step (1). This could be derived from initial measurements on the system. After each execution of a request in step (2) the empirical distribution is updated at step (3). A DP based lookup table could leave out unattractive concrete service providers. In that case we do not receive any information about these providers. These could become attractive if the response-time behavior changes. Therefore in step (4), if a provider is not visited for a certain time, a probe request will be sent at step (5b) and the corresponding empirical distribution will be updated at step (6a). After each calculation of the lookup table, the current set of empirical distributions will be stored. These are the empirical distributions that were used in the lookup table calculation and form a reference response-time distribution. Calculating the lookup table for every new sample is expensive and undesired. Therefore we propose a strategy where the lookup table will be updated if a significant change in one of the services is detected. For this purpose the reference distribution is used for detection of response-time distribution changes. In step (5a) and step (6a) the reference distribution and current distribution are retrieved and a statistical test is applied for detecting change in the response-time distribution. If no change is detected then the lookup table remains unchanged. Otherwise the lookup table is updated using the DP. After a probe update in step (5b) and step (6b) we immediately proceed to updating the lookup table as probes are sent less frequently. In step (7) and step (8) the lookup table is updated with the current empirical distributions and these distributions are stored as new reference distribution. By using empirical distributions we are directly able to learn and adapt to (temporarily) changes in behavior of third party services.

Using a lookup table based on empirical distributions could result in the situation that certain alternatives are never invoked. When other alternatives break down this alternative could become attractive. In order to deal with this issue we use probes. A probe is a dummy request that will provide new information about the response time for that alternative. As we only receive updates from alternatives which are selected by the dynamic program, we have to keep track of how long ago a certain alternative has been used. For this purpose to each concrete service provider a probe timer $U^{(i,j)}$ is assigned with corresponding probe time–out $t_p^{(i,j)}$. If a provider is not

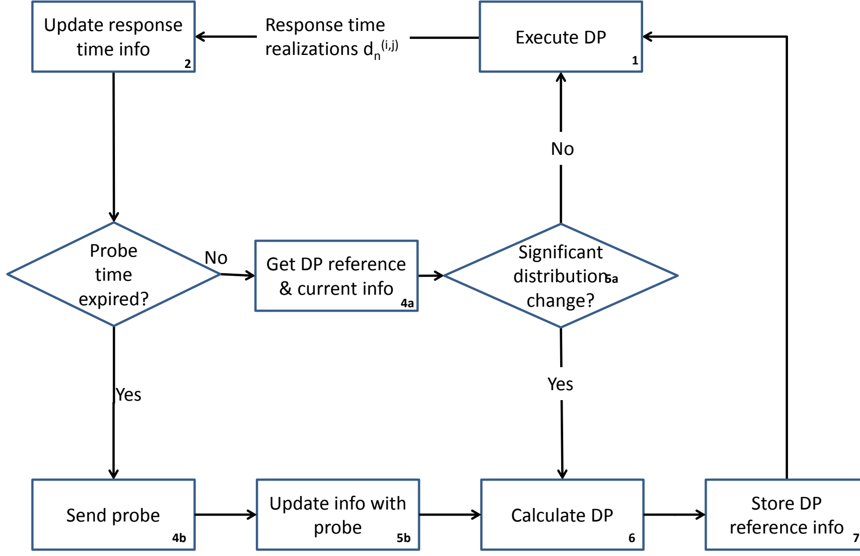

Figure 2: Real-time QoS control approach.

visited in $t_p^{(i,j)}$ requests ($U^{(i,j)} > t_p^{(i,j)}$) then the probe timer has expired and a probe will be collected incurring probe cost $c_p^{(k,j)}$. If for example, in Figure 1, the second alternative of the third task has not been used in the last ten requests, the probe timer for alternative two has value $U^{(3,2)} = 10$. After a probe we immediately update the corresponding distribution. No test is applied here as probes are collected less frequent compared to processed requests.

V. Algorithms

In this section we elaborate on the algorithms and detection mechanisms that are used in the closed loop control approach. These include dynamic programming (DP) and the corresponding empirical distribution discretization.

A. Empirical distribution discretization approach

A natural approach for tracking changes is that we define a sliding window of W samples. Let $\mathcal{X}_n = \{d_{n-W+1}, d_{n-W+2}, \ldots, x_n\}$ be the current set of samples in the sliding window after inserting the nth sample d_n. These sets are used to determine the empirical distribution that serves as an input for the dynamic program.

Let h be the discretization step size. Let T^* be the end to end deadline: $T^* = \left\lceil \frac{\delta_p}{h} \right\rceil$. Furthermore we define $q_k^{(i,j)}$ as the discretized empirical distribution of concrete service alternative j at task j at remaining budget $b = hk$. Using the discretization approach of [5] we discretized the empirical distributions as follows for $i = 1, \ldots, N, \quad j = 1, \ldots, M_i$, $k = 0, \ldots, T^*$:

$$q_k^{(i,j)} = \begin{cases} \sum_{t=1}^{W} \mathbb{1}_{\{h[k-0.5] < d_{n-t}^{(i,j)} \leq h[k+0.5]\}}, & \text{if } k < T^*, \\ \sum_{t=1}^{W} \mathbb{1}_{\{d_{n-t}^{(i,j)} > h[k-0.5]\}}, & \text{otherwise.} \end{cases} \tag{1}$$

Here is $d_t^{(i,j)}$ the t-th response-time realization for service alternative j at task i,
and $\mathbb{1}_{\{A\}}$ is the indicator function over A which is 1 if A is true and 0 otherwise.

Actually $q_k^{(i,j)}$ is a histogram where kth bin is bounded by $\left[\, h(k-0.5); h(k+0.5) \,\right)$ and where the sum of the frequencies is normalized to 1. There is a tradeoff in choosing the bin size. When h has a small value the histogram will consist of many bins. In the case that a few large bins are used, too much information about sample location is lost.

B. DP formulation

Using the discretized empirical distributions backward recursion formulas can be formulated. We start with the terminal reward function for $b = 0, \ldots, T^*$:

$$P_b^{(N+1,*)} = \begin{cases} R & \text{if } b > 0, \\ -V & \text{otherwise.} \end{cases} \tag{2a}$$

Using this function we iterate backwards using the following equations for $i = 1,\dots,N,\quad j = 1,\dots,M_i,\quad b = 0,\dots,T^*$:

$$P_b^{(i,*)} = \max_j \Big\{ - c^{(i,j)} + R_b^{(i,j)} + V_b^{(i,j)} \Big\}, \tag{2b}$$

$$R_b^{(i,j)} = \sum_{k=0}^{b} q_k^{(i,j)} P_{k-b}^{(i+1,*)}, \text{and} \tag{2c}$$

$$V_b^{(i,j)} = \sum_{k=b+1}^{T^*} q_k^{(i,j)} P_0^{(i+1,*)}. \tag{2d}$$

Here, the term $P_b^{(i,*)}$ represents the expected profit per request given time budget b at task i under the optimal dynamic programming decision strategy. The term $R_b^{(i,j)}$ represents the expected reward, when concrete service j (assigned to task i) is executed for the given time budget value b. Finally the term $V_b^{(i,j)}$ represents the expected penalty for exceeding the overall deadline at task i while executing concrete service j for the given time budget value b. The expected reward and penalty functions take into account the impact of future decisions as represented by terms relating to $P_b^{(i+1,*)}$ in (2c) and (2d).

While applying formulas (2b)–(2d), the corresponding decisions (actions) $A_k^{(*,i)}$ can be obtained by storing the maximum arguments for $i = 1,\dots,N,\quad j = 1,\dots,M_i,\quad k = 0,\dots,T^*$:

$$A_k^{(i,*)} = \operatorname*{argmax}_j \Big\{ - c^{(i,j)} + R_k^{(i,j)} + V_k^{(i,j)} \Big\}.$$

VI. Experimental setup

In this section we will first explain our test environment and finally describe the experiments.

A. Test environment

Our test lab environment is based on Java 8 and NIO (asynchronous IO) and consists of four types of componens, depicted in Figure 3. (1) In the Orchestrator module there are three components. The Dispatcher component receives composite service requests from the Load generator and orchestrates the corresponding sub-service requests. These sub service requests are dispatched via the Service registry interface. All sub services can register their functionality at the Service registry interface. (2) A Sub service has a (HTTP request) service interface and a Control interface. Requests are served after a pre-configured random delay. Using the Control interface any desired response-time distribution can be configured. All our DP algorithms are implemented in Matlab. (3) Matlab is connected using an Experiment Control Interface (via TCP sockets) to the Load generator and Orchestrator components. The Orchestrator forwards experiment setup requests to the connected Sub services. (4) The Load generator generates requests for the composite service (Orchestrator). The requests are generated using a Poisson process.

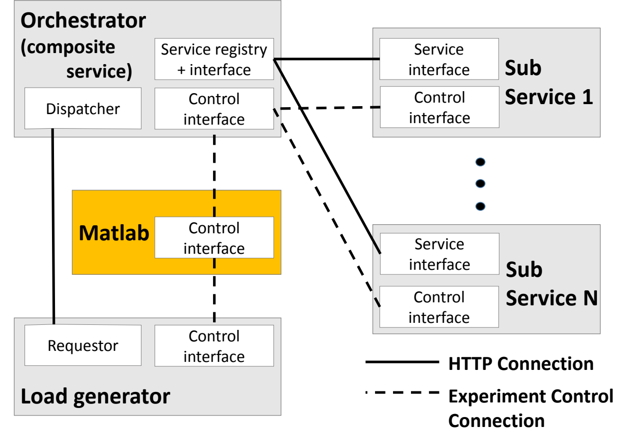

Figure 3: Test lab environment.

B. Experiments

To test the real-time QoS control approach defined in Section IV we define an experimental workflow that can be used for investigating the impact of the closed-loop control approach parameters. We emphasize that this experimental set-up is tailored for evaluating responsiveness of our real-time QoS control approach with respect to response-time distributions and does not limit us in tracking other systems with different response-time models.

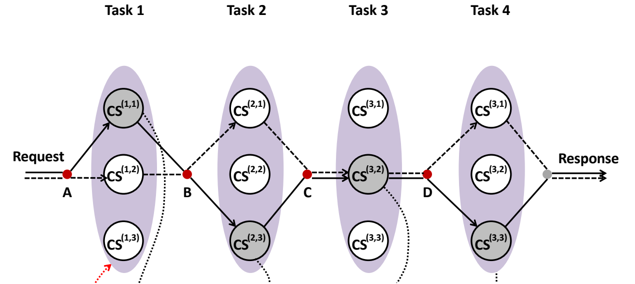

Figure 4: Experimental work flow.

Figure 4 represents our experimental work flow consisting of four tasks. For each task three concrete service alternatives are available. At each alternative j for task T_i the response-time distribution is summarized in terms of mean $\mu^{(i,j)}$ and variance $\left(\sigma^{(i,j)}\right)^2$.

The parameters $\mu^{(i,j)}$, $\sigma^{(i,j)}$, $c^{(i,j)}$ of our experiments are summarized in Table I. There is one cheap reasonably slow service, one faster but more expensive service and a equally expensive service that has slow performance. The latter is intattractive and therefore the algorithms should not include this alternative in the lookup table.

We consider a run consisting of 10000 requests arriving according to a Poisson process with a rate of 100 requests per second. Furthermore, we consider a deadline of $\delta_p = 500ms$.

Table I: Task concrete service alternatives

	Task i		
Concrete service↓	c	μ	σ
Alternative $(i, 1)$	1	250	40
Alternative $(i, 2)$	5	62.5	60
Alternative $(i, 3)$	5	300	40

The default setup is such that using DP at least 95% requests can be served within the deadline. Each successful response to a request will gain a reward of $R = 25$ money units while for a failed request $V = 40$ money units has to be paid.

To test the effectiveness of our approach we interchange the equally priced services $(i, 2)$ and $(i, 3)$ for $i = \{1, 2, 3, 4\}$ after 5000 requests. We denote with N_{swap} the position in the workflow where services are interchanged. The interchange event is represented by the red line in Figures 5-7. Our approach should detect this and adopt the lookup table correspondingly. We compare three approaches: (1) the sliding window approach based on Kolmogorov Smirnov statistic for change detection, (2) the empirical distribution based approach without change point detection and (3) theoretical from the policy based on the *known in beforehand* interchange of services and knowledge of the actual response time distributions. We vary W_p in the range $\{25, 50, 100\}$. The probe interval is fixed to $t_p = 50$ requests. Furthermore we vary the test significance level with $\alpha = \{0.01, 0.05\}$.

VII. Results

This section contains the results from the experiments as described in Section VI. We demonstrate the impact of two parameters the position where the alternatives interchange N_{swap} and window size W. In all Figures 5-7 the vertical axis represents the moving average over 600 request revenues (monetary units). This approximately corresponds to 6 seconds. The horizontal axis represents the total number of requests that has been served. Normally (after startup effects) the moving average should be near to 5. However when subservice response times degrade the target deadline δ_p is more likely to be violated. As a result revenues will drop due to deadline violation penalties.

With Figure 5 we represent two benchmark cases. (1) In Figure 5a the lookup table is not updated at all resulting in a significant drop in revenue when services are interchanged after 5000 requests. The orchestrator is not recovering at all as the lookup table is not updated anymore. (2) The theoretical *known in beforehand* case is represented in Figure 5b. Here the interchange of services is not observable because of the full response time behavior knowledge (at any time).

Figures 6 and 7 demonstrate the effectiveness of our real time QoS control approach. When the interchange occurs a drop in revenues can be observed. This drop lasts until the change has been detected. After the lookup table has been adopted to the

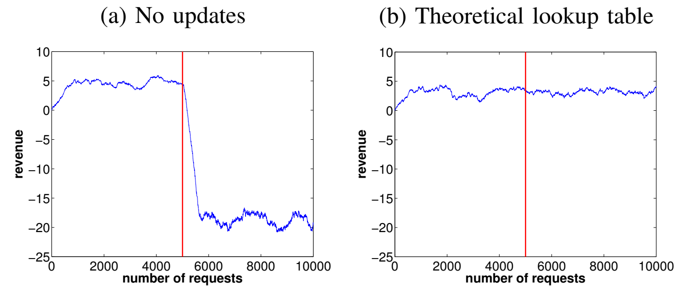

Figure 5: Benchmark cases.

new situation the performance recovers to the level before the services interchanged. The impact of parameter settings can be observed from the width of the gap. Wider gaps correspond to parameter settings that result in slower recovery from changes in service behavior.

Regarding the effect of service interchange position N_{swap} we observe that the gap is wider at the first position and last position (depicted by Figures 6a and 6d). In the intermediate positions (Figures 6b-6c) the gap is smaller. This is where the lookup table approach mostly benefits from response time information from previous service invocations. On the other hand, at the start of the workflow no remaining response time budget information is obtained for the current composite request while after the last position a long response time can not be compensated anymore.

In Figure 7 we observe the effect of window size on our approach. In Figure 7a we observe a significantly wider gap then in Figures 7b and 7c. As one could expect a larger window size results in slower responses to deteriorations. A larger window size requires more samples to fully change the empirical distribution to the new situation.

VIII. Discussion

We modeled and implemented a real-time QoS control approach where dynamic programming is applied on empirical distributions resulting from the actual realized response-time distributions of concrete service providers. Our approach is robust to changes in the sense that it adapts to changes in response-time distributions of concrete service alternatives. To achieve this we use a sliding window approach on the empirical distribution. When using our approach there is a trade–off between parameters that we need to optimize. These parameters are the sliding window W and the change point detection test significance α. The constraints are here determined by computational power and probe cost. Typically we would like to update our lookup table every request and probe frequently. However it takes time to compute a new strategy. We should choose probe time–outs such that we can exploit information about improved service without spending

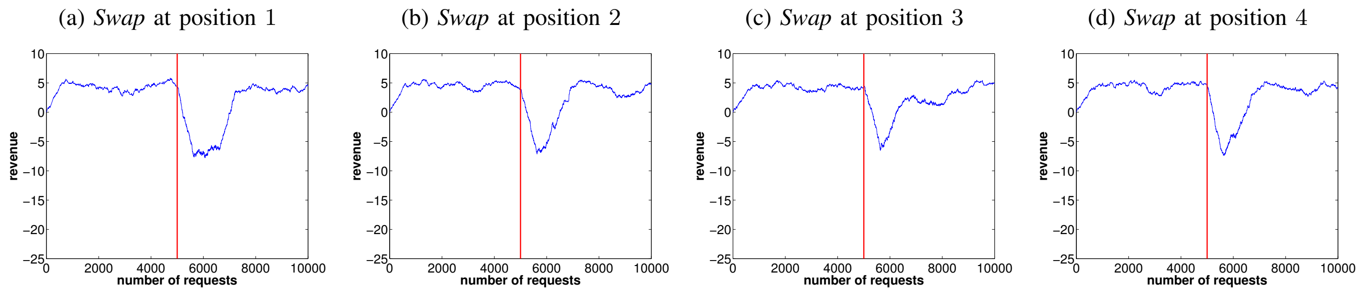

Figure 6: Effect of service change position in the chain (W=50, $\alpha = 0.01$). The vertical red line indicates the change in service response time distributions.

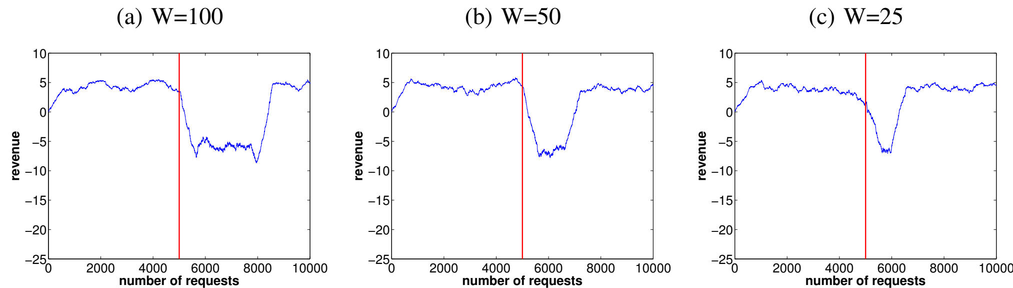

Figure 7: Effect of window size W at position 1 with $\alpha = 0.01$. The vertical red line indicates the change in service response time distributions.

too much cost on probing and using too much computational power. Experimental results indicate that in an environment with changing response-time behavior our real-time QoS control approach has a significant advantage compared to a static lookup table. Moreover our approach has the strong advantage that it learns and exploits response-time behavior on the fly.

Tuning window size W and α creates a second layer of control where these parameters are adapted to optimal values. The update of these parameters is typically on a larger time scale that is not in the scope of our experiments. Furthermore the age of samples should be considered. A discounting approach on the age of the samples could potentially improve the responsiveness as newer samples have bigger impact. This is a interesting direction for further research.

References

[1] A. G. Barto and S. Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(4):341–379, 2003.

[2] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani. An approach for qos-aware service composition based on genetic algorithms. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1069–1075. ACM, 2005.

[3] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani. A framework for qos-aware binding and re-binding of composite web services. *Journal of Systems and Software*, 81(10):1754–1769, 2008.

[4] V. Cardellini, E. Casalicchio, V. Grassi, and F. L. Presti. Adaptive management of composite services under percentile-based service level agreements. In *Service-Oriented Computing: 8th International Conference, ICSOC 2010, San Francisco, CA, USA, December 7-10, 2010. Proceedings*, volume 6470, page 381. Springer-Verlag New York Inc, 2010.

[5] G. L. Choudhury and D. J. Houck. Combined queuing and activity network based modeling of sojourn time distributions in distributed telecommunication systems. *The Fundamental Role of Teletraffic in the Evolution of Telecommunications Networks (Eds. J. Labetoulle and JW Roberts), Proc. ITC*, 14:525–534, 1994.

[6] P. Doshi, R. Goodwin, R. Akkiraju, and K. Verma. Dynamic workflow composition using markov decision processes. *International Journal of Web Services Research*, 2(1):1–17, 2005.

[7] A. Gao, D. Yang, S. Tang, and M. Zhang. Web service composition using markov decision processes. In *Advances in Web-Age Information Management: 6th International Conference, WAIM 2005, Hangzhou, China, October 11-13, 2005, Proceedings*, volume 3739, page 308. Springer, 2005.

[8] A. Gosavi. Reinforcement learning: a tutorial survey and recent advances. *INFORMS Journal on Computing*, 21(2):178–192, 2009.

[9] P. Leitner. Ensuring cost-optimal sla conformance for composite service providers. In *ICSOC/ServiceWave 2009 PhD Symposium*, page 43, 2009.

[10] C. Preist. A conceptual architecture for semantic web services. *The Semantic Web–ISWC 2004*, pages 395–409, 2004.

[11] T. Yu, Y. Zhang, and K. J. Lin. Efficient algorithms for web services selection with end-to-end qos constraints. *ACM Transactions on the Web (TWEB)*, 1(1):6, 2007.

[12] L. Zeng, C. Lingenfelder, H. Lei, and H. Chang. Event-driven quality of service prediction. *Service-Oriented Computing–ICSOC 2008*, pages 147–161, 2008.

[13] H. Zheng, W. Zhao, J. Yang, and A. Bouguettaya. Qos analysis for web service composition. In *2009 IEEE International Conference on Services Computing*, pages 235–242. IEEE, 2009.

[14] M. Živković, J. W. Bosman, J. L. van den Berg, R. D. van der Mei, H.B. Meeuwissen, and R. Núnez-Queija. Run-time revenue maximization for composite web services with response time commitments. In *Advanced Information Networking and Applications (AINA), 2012 IEEE 26th International Conference on*, pages 589–596. IEEE, 2012.