Adaptive Request Prioritization in Dynamic Service-oriented Systems

Roman Khazankin, Daniel Schall, Schahram Dustdar
Distributed Systems Group
Vienna University of Technology
Argentinierstraße 8/184-1, A-1040 Vienna, Austria
{lastname}@infosys.tuwien.ac.at

***Abstract*—The availability of scarce resources in a service-oriented system demands for context-aware selection policies that adapt based on service-level agreements (SLAs). One of the open issues is to prioritize service requests in dynamically changing environments where concurrent instances of processes may compete for resources. Here we propose a runtime monitoring approach to observe the actual state of the system. We argue that priorities should be assigned to requests based on potential violations of SLA objectives. While most existing work in the area of quality of service monitoring and SLA modeling focuses typically on purely technical systems, we consider service-oriented systems spanning both software-based services and human actors. Adaptive request scheduling in such systems is challenging due to the poorly predictable behavior of human actors in performing tasks. Our approach helps to cope with these challenges by prioritizing service requests that may cause violations of SLAs and corresponding objectives that are associated with processes.**

***Keywords*-service-oriented systems, monitoring, adaptation, service-level agreements, scheduling, human factors**

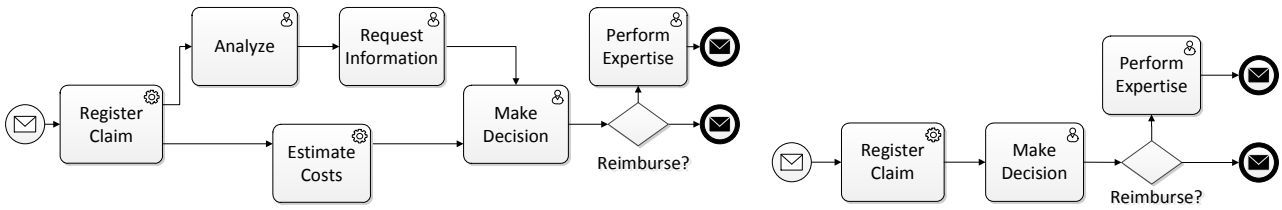
I. Introduction

Service-oriented systems have become an important approach and technological framework to solve problems in distributed computing environments. Challenges in distributed service-oriented systems include the discovery of resources and monitoring of the system's runtime behavior. Capturing the current state of the system is essential in dynamic environments where services are discovered and invoked at runtime. Research in the area of quality of service (QoS) modeling and monitoring (e.g., see [1]) has provided an important building block to observe the runtime state of a service-oriented system. Service-level agreements (SLAs) let service providers specify quality criteria demanded by consumers [2]. Keeping services compliant to SLAs is crucial in a service-oriented system. Usually, if the system is designed properly and acts as expected (e.g., response time and service availability), the SLA is satisfied. However, both internal and external factors can compromise the overall performance of the system. While the strategic actions should be taken to prevent the system from entering undesirable conditions (e.g., through replicating the components, adding resources), the run-time adaptation can also be performed in attempt to minimize the penalties in given situations. In this work, we assume that multiple processes need to access shared resources in a singleton manner. Assume a process consisting of multiple activities, some of them enacted by invoking software services and certain activities performed by human actors. In a service-oriented system, such a scenario could be realized by modeling and enacting compositions using the Business Process Execution Language (BPEL) [3], where human steps are modeled using BPEL4People and WS-HumanTask [4], [5]. Service provided by human actors can be regarded as 'low throughput' services because humans naturally work at lower speeds than software-based services. If human-based low throughput services are part of a process, the order in which processes get the response from such services impacts the process execution times. If the system gets overloaded or some processes are late due to unexpected delays, the SLAs might be violated. However, the control over the ordering of service requests can reduce the penalties or prevent the violations at all.

To address these challenges, we propose a dynamic adaptation approach and heuristic prioritizing algorithm that analyzes the current state of the service-oriented system at runtime and prioritizes service requests according to assigned process SLAs. We assume that the execution state of all the processes in a service-oriented system is accessible, and that the penalty functions of SLAs are provided. The main idea behind the algorithm is that the priority of a service execution is given to those processes that are expected to produce the highest penalties if this service is delayed for them. To illustrate our approach, we discuss insurance claim processes. The contributions of the paper are as follows:

- We present the conceptual approach for service request prioritization in orchestration engines.
- We design a priority assignment algorithm to cope with the challenges in dynamic SOA.
- We present a detailed evaluation of our approach to demonstrate its effectiveness and efficiency.

The paper is organized as follows: Section II overviews related work with the focus on adaptation techniques in service-oriented systems. In section III we present a motivating scenario and in section IV the adaptation model. Experiments are shown in section V. The paper is concluded in section VI.



(a) Comprehensive insurance. (b) Liability insurance.

Figure 1. Claim processes for different insurance cases.

II. Related Work

Our approach is aiming at minimizing SLA penalties via prioritizing the requests and assigning the available service throughput. Such an objective constitutes a scheduling problem. Among the variants of this problem, the resource-constrained multi-project scheduling problem (RCMPSP) [6] is the most conformable to ours. However, those studies do not address the service-oriented architecture, and, thereby, related concepts such as SLA or QoS.

A priority scheduling method for process engines is proposed in [7]. It analyzes the execution status in the process engine and dynamically assigns the priorities to service requests alike to our approach. Instead of penalty functions, it considers utility functions. However, this work assumes that services support prioritized execution of requests with either *High* or *Low* priority. Such an assumption has two strong disadvantages: firstly, it severely reduces the scope of application, as services do not support prioritized execution in general, and, secondly, even if a service distinguishes between requests with high and low priorities, it would still not be able to distinguish between requests with the same priority which is crucial in case of multiple concurrent requests and low service throughput. Unlike it, our approach uses essentially different prioritization algorithm and request scheduling (via proxy schedulers), so it does not have these disadvantages.

SLA violation and prevention in service compositions through adaptation is addressed by various researchers. For example, [8] proposes a general adaptation framework for monitoring and preventing SLA violations by performing various actions upon the service composition, like changing the service bindings or composition structure. In contrast to it, our approach does not address the composition changes, but request prioritization among different compositions.

Various escalation mechanisms to avoid breaking the workflow deadlines are discussed in [9]. The prioritization of tasks or cases which is highly related by implication, is briefly discussed. However, the paper does not consider SLAs and penalties, and no rationale regarding the actual implementation of the method is given.

Trade-offs between costs and profits of various service composition adaptations are discussed in [10]. The adaptation proposed by us does not imply any costs besides the performance overhead used for the analysis.

The approaches like [11], [12] use dynamic binding to improve the QoS of process instances, whereas our approach does not assume the existence of several service endpoints.

III. Scenario

To illustrate our approach, we discuss a motivating scenario where processes are designed and executed in the context of insurance claim handling. We look at different kinds of insurance processes: the first one dealing with a *comprehensive* insurance plan and the second with *liability* coverage. Both cases are depicted in Fig. 1.

The comprehensive plan ensures that damage (for example accidents or vandalism) is being paid by the insurance company. In certain European countries, liability is the minimum insurance coverage everyone must have due to government regulations. As an example, if A is responsible for the damage of B, then A's insurance company must pay for B's damage. Fig. 1(a) shows the process for the comprehensive insurance plan. People obtaining coverage through this plan may be regular or premium customers. For premium customers, the insurance company wants to provide better quality of service as for regular customers. For example, faster processing of the insurance claim. The process is initiated as soon as the customer issues an insurance claim. The registration of the claim is performed automatically by a software service. In the next step the process splits into two parallel branches. Based on the issued claim, a software service is invoked (step *Estimate Cost*) to perform an automatic calculation of the expected costs. A person from the insurance company analyzes the received claim and typically requests further information from the customer. After both branches have finished, a decision is made by a supervisor. The outcome may be to reimburse the customer or not. In the first case, an expert reviews the case by visiting the customer to obtain precise understanding of the damage upon which actual calculations are made. The alternate case terminates by sending a (auto-)generated notification to the customer.

The second process example is shown in Fig. 1(b). In contrast to the comprehensive insurance example in Fig. 1(a), we assume in this scenario that the person filing the claim is *not* a customer of the insurance company. The process

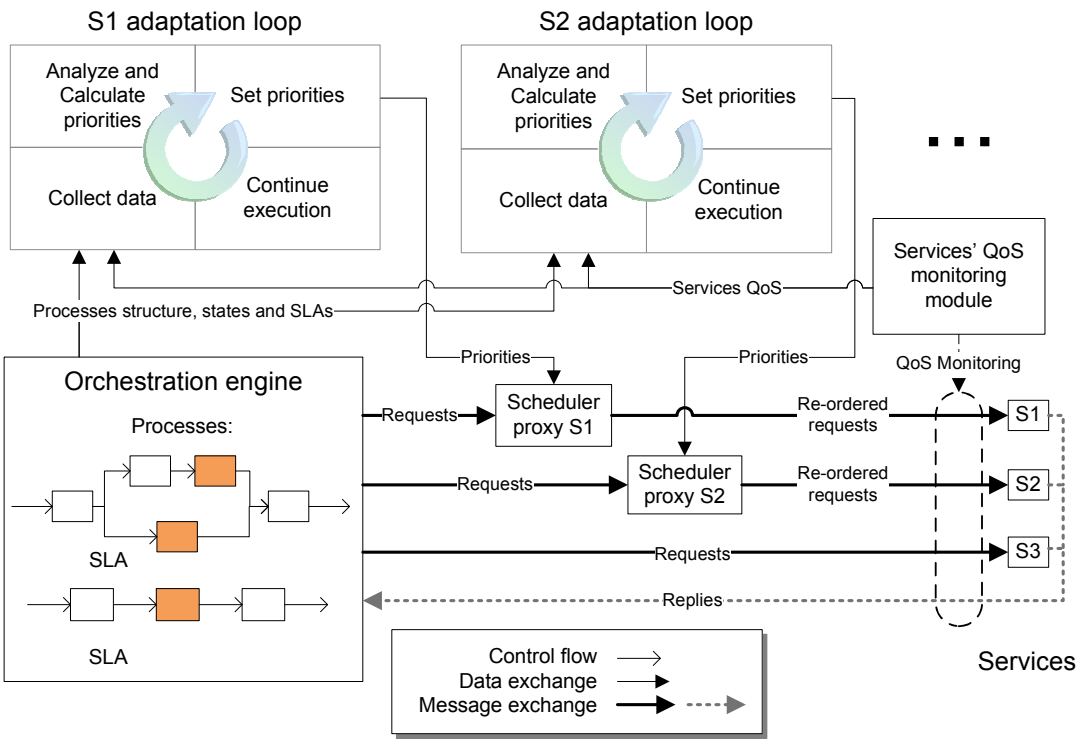


Figure 2. The overall architecture of the approach.

is therefore simpler because the person filing the insurance claim only receives limited support (e.g., help desk) and also limited service-level guarantees are given. The process is initiated in the same manner as in the comprehensive insurance plan example. Afterwards, a decision is made based on received information. The next steps are again equivalent to the steps (*Perform Expertise* and notification) of the first process.

What these processes have in common is that they access shared resources. For example, by invoking a service in the step *Perform Expertise*. If a process invokes this service, other processes (instances) may need to wait until free resource capacities are available. However, this could cause violations in SLA objectives. Thus, careful scheduling of requests is needed to satisfy customer needs.

IV. Adaptation Model

In this section we describe the conceptual architecture of our solution, the prioritization algorithm, and the remarks regarding the deployment of such a solution in a real SOA. We assume that all services in the system are atomic (not composite), as we can decompose all such services.

A. Conceptual Architecture

The architecture of the approach is depicted by Fig. 2. Normally, when a process invokes a service, a request message is sent to the service endpoint, so the order in which requests are processed by a service is determined by this service's implementation which is unaware of processes running, SLAs, or other context information. In our approach, a scheduler proxy is created and assigned to each service whose request priorities are being adapted. The scheduler proxy intercepts requests to the service and reorders them according to prior defined priorities. The scheduler is aware of the service's QoS (through the monitoring module), so it dispatches requests towards the service depending on the available free capacities. It ensures that the priorities are obeyed. The priorities are periodically set by an adaptation loop created for the service. As the re-ordering is performed before the requests are sent to a service, the actual location of the service does not play a role.

The adaptation loop consists of three phases:

1) *Collection of context and monitoring information.* The structure, the current execution state, and penalty functions (from SLAs) of all currently running processes as well as the QoS information are collected from the orchestration engine and from the monitoring module respectively. We use a deterministic QoS model, so the throughput and response time are considered single values. There exist various approaches for QoS monitoring (e.g., [1], [13]) which is out of the paper's focus.
2) *Calculation of request priorities.* The collected data is passed to the algorithm (described in Listing 1) which calculates priorities for forthcoming and recently made

Input : Service S, its response time S_{RT}, set of processes P, for each process p penalty function $L_p(t)$
Output: Ranked requests

```
1  for process p in P do
2      R_p = pending requests of S in p
3      R_e = requests of S predicted to be made during S_RT/2 period in p
4      R = R_p ∪ R_e
5      Assume that replies of all requests in R are received after S_RT, predict time t of finish for process p
6      l_0 = L_p(t)  // Default penalty
7      for request r in R do
8          Assume that a reply of r is received after S_RT * 2 and replies of all other requests in R are received after
           S_RT, predict time t_r of finish for p
9          l_r = L_p(t_r)  // Penalty for current request
10         d_r = l_r − l_0  // Difference between default penalty and the penalty for current request
11         Add the tuple of r, d_r, and request time k (either real or predicted) to list D
12     end
13 end
14 Sort D descending by d_r then ascending by k
15 Return D
```

Listing 1: Prioritization algorithm.

requests to the service.

3) *Scheduler update.* The corresponding scheduler is updated with the calculated priorities, so the requests to the service can be ordered accordingly.

Iterations in the adaptation thread are performed with the period of the half of the service's response time. This value equals to prediction period (See Listing 1, line 3). The value was selected empirically. As it was evaluated in experiments, if the period was greater, the algorithm performed poorer as sometimes service throughput was unused too long while waiting for predicted requests, however, the lesser period did not improve the performance of the algorithm.

B. Deployment in a Service-oriented System

Although not the main focus of this work, we give a short analysis of the mapping and deployment of our conceptual architecture in real SOA environments. We assume that there is single and accessible (in-house) orchestration engine. Our approach would also work with multiple deployed orchestration engines in the environment. However, for simplicity of discussions, we assume only a single engine. In order to enable the deployment of the proposed architecture, the orchestration engine should be extended to supply the adaptation loops with process state information. Many SOAs have moved towards a bus-oriented messaging backbone. An enterprise service bus (ESB) should be configured to support scheduler proxies. We expect these extensions to be implemented as plugins for corresponding SOA components, however, such an implementation fully depends on the underlying technologies and software being used.

C. Prioritization Algorithm

The prioritization algorithm is outlined in Listing 1. The algorithm predicts forthcoming calls of the service and prioritizes the corresponding requests together with the pending requests according to the penalty difference which appears if the receiving of request's reply is delayed. The algorithm uses predictions which are performed straightforwardly, adding together response times of the services to be called according to the process structure. As for flows and conditions, the most delayed branche's time is selected. Currently we address only *sequence*, *flow* and *condition* process constructs.

D. Illustrative Example

To illustrate the work of the algorithm, the algorithm steps for two instances of comprehensive insurance claim scenario process are described. Let the services have the same QoS as in experimental setting (See Sec. V) and let both process instances have SLA penalty functions L_S(t,3,10). Let the processes have the states as depicted in Fig. 3. Instance 1 was delayed for some reason. The Information request service (IRS) was called 0.05 sec ago there, so the process is waiting for its response; the Cost estimation service has already returned the response. In instance 2 the Decision making service (DMS) was called 0.01 sec ago. Given that DMS's request priorities are being adapted, the analysis step of its adaptation loop's next iteration would perform as shown in Listing 2. Finally, when D is sorted, the priority of c_1 is considered higher than the priority of c_2.

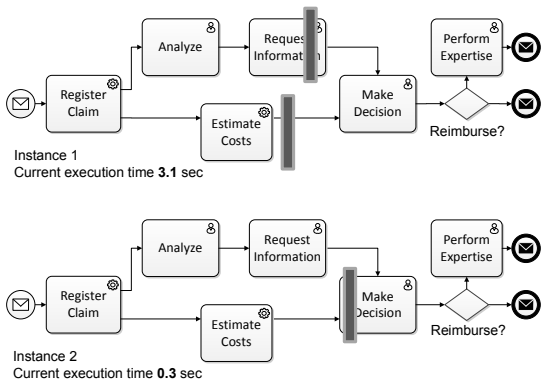


Figure 3. An example of process states.

// Process instance 1:

1 Instance 1 has no pending calls of DMS, however, as the IRS is expected to respond in 0.05 sec (as its response time is 0.1 sec), then the call c_1 to DMS is predicted in 0.05 sec.

2 Default process finish time is calculated: $3.1 + 0.05 + 0.2 + 0.5 = 3.85$ sec

3 Default penalty is calculated: $L_S(3.85, 3, 10) = 0$

4 c_1 is assumed to respond in $0.2 * 2 = 0.4$ sec, Process finish time is calculated: $3.1 + 0.05 + 0.4 + 0.5 = 4.05$ sec

5 The penalty for delayed c_1 is calculated: $L_S(4.05, 3, 10) = 10$

6 The penalty difference for c_1 is calculated: $d_{c_1} = 10 - 0 = 10$

7 $< c_1, 10, 0.05 >$ is added to D

//

// Process instance 2:

8 Instance 2 has a pending DMS call c_2. No other DMS calls are predicted.

9 Default process finish time is calculated: $0.3 + 0.05 + 0.2 + 0.5 = 1.05$ sec

10 Default penalty is calculated: $L_S(1.05, 3, 10) = 0$

11 c_2 is assumed to respond in $0.2 * 2 = 0.4$ sec, Process finish time is calculated: $0.3 + 0.05 + 0.4 + 0.5 = 1.25$ sec

12 The penalty for delayed c_2 is calculated: $L_S(1.25, 3, 10) = 0$

13 The penalty difference for c_2 is calculated: $d_{c_2} = 0 - 0 = 0$

14 $< c_2, 0, -0.01 >$ is added to D

Listing 2: Algorithm steps for the example.

V. Experiments and Discussion

We have implemented an orchestration engine simulator which mimics the QoS characteristics of services and the execution of processes. It simulates the temporal behavior of the system and supports basic process elements: *sequence*, *flow*, and *condition* (executes with given probability). To demonstrate the advantage of our approach, we simulate unexpected overloads and delays in a service-oriented system under various circumstances. We scaled the realistic response times of the services for simulation from days to seconds. So the half of a simulated second corresponds to half of a day in real setting.

A. Setup

In our setup, several process types are repeatedly instantiated in the system according to the frequency $F(t)$, as shown in Fig. 4.

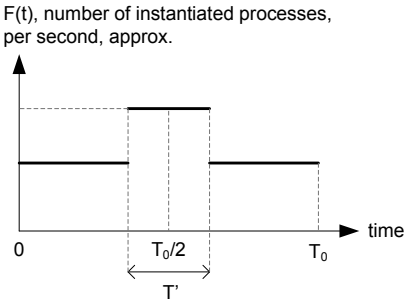


Figure 4. Experiment model.

The type of instantiated process is chosen randomly (all types are considered equiprobable). The approximate number of instantiated processes per second is increased from F_0 to F' for a period T' in the middle of the overall process instantiation timespan T_0. The unexpected additional load is thus simulated. The inaccuracy of response time is simulated as well: the actual response time of a service is calculated as $RT + RT * k * R$ where RT - expected response time, k - inaccuracy factor, R - normally distributed random value.

We apply this system behavior for 6 series of experiments (E1-E6) based on the motivating scenario (Sec. III). The experiments are described in Table II. The processes types $T1$ and $T2$ correspond to comprehensive insurance claim and motor vehicle liability insurance claim processes. The QoS values used for services simulation are presented in Table I (the set of services maps to the steps of motivating scenario processes). All experiments use response time inaccuracy factor of *0.3*. The conditions in both processes are assumed to be true in 70% of cases. In E5 and E6, the analysis service happens to be delayed by 0.5 sec in 10% of cases.

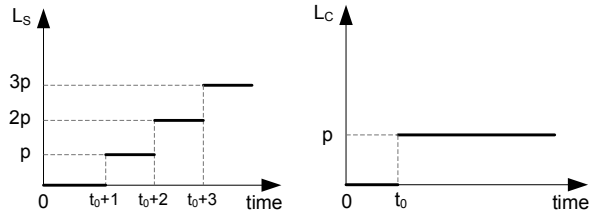
Table I
Quality of Service

Name	**Response time** [sec]	**Throughput**
Analysis service	0.15	5
Expertise service	0.50	5
Decision making service	0.20	5
Information request service	0.10	10
Estimation service	0.10	100
Registration service	0.01	100

In our simulation, throughput indicates the number of simultaneous requests that can be served by a service. As penalty functions, we used staged L_S and constant L_C functions (see Fig. 5).

$$L_S(t,t_0,p) = \begin{cases} 0 & \text{if } t < t_0 \\ (trunc(t) - t_0) * p & \text{if } t >= t_0 \end{cases} \quad (1)$$

$$L_C(t,t_0,p) = \begin{cases} 0 & \text{if } t < t_0 \\ p & \text{if } t >= t_0 \end{cases} \quad (2)$$



(a) Staged. (b) Constant.

Figure 5. Penalty functions.

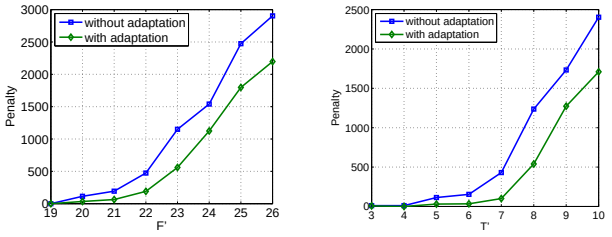
Each experiment was performed 2 times: first time with no adaptation with requests served in First-In-First-Out manner, and the second time with the adaptation enabled. Penalties were measured for each process. As the simulation involves various random factors (process instantiation, process type selection, error and unexpected delay injection, conditions), we made sure that such experiments get the same values returned by random generators. The results of experiments are depicted in figures 6-8.

Table II
EXPERIMENTS PERFORMED

Name	Process types: penalty functions	T'	F'
E1	T1 : L_S(t,3,10) only	5	19 - 26
E2		3 - 10	20
E3	T1 : L_S(t,3,10), T2 : L_C(t,8,20)	5	22 - 29
E4		3 - 9	25
E5	T1 : L_S(t,3,10), T1 : L_S(t,3,15),	6	19 - 26
E6	T1 : L_S(t,3,20), T2 : L_C(t,8,20)	3 - 9	22

B. Discussion

All experiments demonstrate a considerable reduction of penalties of 30-80%. In the following we show pairs of figures: the left figure showing SLA penalties by varying **F'** and the right figure by varying **T'**.

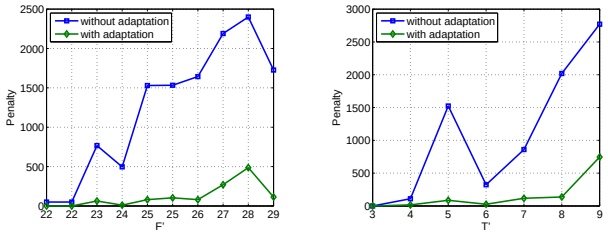


(a) E1 (b) E2

Figure 6. One process type, without delays.

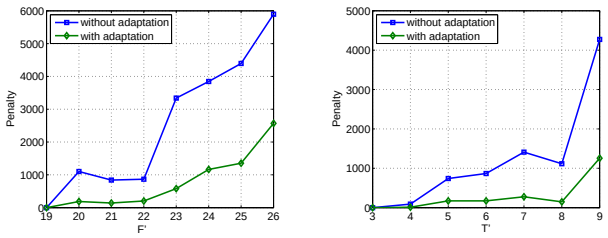
In experiments E1 and E2 (see Fig. 6) the absolute difference between penalties is relatively constant which is explained by the similarity of executed processes: only one process type is instantiated, no difference among instances in form of service delays, the only difference is the variety of response times resulted by the inaccuracy factor. Thus, these experiments give very limited freedom for re-prioritization. Still, the adaptation reduces penalty considerably.

In experiments E3-E6 (see Fig. 7 and Fig. 8) the reduction is greater than in E1-E2 because of the possibility to postpone the service calls in T2 processes at no expense ($t_0 = 8$ for L_C).



(a) E3 (b) E4

Figure 7. Two process types, without delays.



(a) E5 (b) E6

Figure 8. Four process types, with delays.

This is revealed mostly in E3 and E4 as approximately half of the processes were of type T2. In E5 and E6 the reduction is lesser than in E3 and E4, because only quarter of processes were of type T2. In contrast to E1-E2, the absolute difference between penalties in E3-E6 grows with the load increment, as the process pool contains various types of processes which causes the dissimilarity of re-prioritization impact, and, thus, increases the algorithm's efficiency. The non-monotonicity of penalty growth in E3-E6 is caused by the random factors in process generation and instantiation mechanism. To summarize these observations, the performed experiments clearly show the advantage of using adaptation for prioritizing requests in case of unexpected overload or response delays.

Of course, the are limits for applying the adaptation. These limits are reached when the time needed to perform an analysis iteration of the orchestration engine state becomes comparable with the response times of the services whose

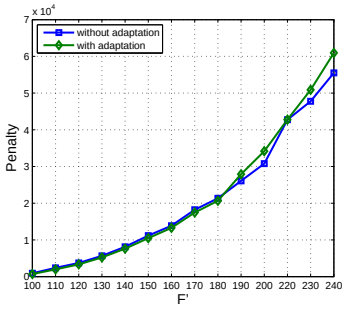


Figure 9. Large values.

request priorities are being adapted. For example, in the experiment with large F' shown in Fig. 9, the method becomes inefficient on $F' > 190$ (the maximal size of the process pool is about 600 processes). However, this limit would scale together with response times of the services, and, in our opinion, will be hardly reachable in a real setting.

VI. Conclusion and Future Work

The problem of SLA penalties reduction in the context of unexpected system overload or service response delays is considered in the paper. The architecture for request scheduling in service-oriented systems and the request prioritization algorithm are proposed. A realistic motivating scenario was taken as a basis for evaluation. The proposed solution was evaluated for the scenario implemented in an orchestration engine simulator. The results of evaluation demonstrate the considerable (30-80%) penalty reduction, thus, showing the clear advantage of the approach. Generally, our approach has no special requirements for SOA system, so it has no obstacles to be applied in practice.

In our future work we plan to adapt and evaluate existing scheduling algorithms to request prioritization problem and to compare their efficiency to our algorithm's. Also we plan to extend the model to allow different services to share the resources, so, for example, if one human is assigned to perform different tasks represented by different services, the system will be aware that the call of one service would impact the QoS of another. Also, we plan to extend our approach towards crowdsourcing scenarios [14], [15]. Systems such as Amazon Mechanical Turk[16] make the capabilities of an open workforce available by letting requesters issue human-intelligent task requests. The challenge in such environments is the limited predictability of resource availability (human workers) and behavior (e.g., task acceptance behavior). Apart from algorithmic aspects (scheduling and assignment of workers), models and specifications (such as WS-HumanTask [17] and BPEL4People [18]) for composing services need to be extended to cope with the dynamics inherent to open Web-based environments. For example, providing adaptive escalation models based on different SLAs or extending temporal aspects that can be modeled in WS-HumanTask with penalty functions.

Acknowledgements

This work was supported by the European Union FP7 project COIN (No. ICT-2008-216256) and the Vienna Science and Technology Fund (WWTF), project ICT08-032.

References

[1] O. Moser, F. Rosenberg, and S. Dustdar, "Non-intrusive monitoring and service adaptation for ws-bpel," in *Proceeding of the 17th international conference on World Wide Web*, ser. WWW '08. New York, NY, USA: ACM, 2008, pp. 815–824.

[2] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-oriented computing: a research roadmap," *Int. J. Cooperative Inf. Syst.*, vol. 17, no. 2, pp. 223–255, 2008.

[3] Web Services Business Process Execution Language Version 2.0, http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html, Apr. 2007.

[4] F. Leymann, "Workflow-based coordination and cooperation in a service world," in *CoopIS*. Springer Berlin Heidelberg, 2006, vol. 4275.

[5] N. Russell and W. M. Aalst, "Work distribution and resource management in bpel4people: Capabilities and opportunities," in *Proceedings of the 20th international conference on Advanced Information Systems Engineering*, ser. CAiSE '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 94–108.

[6] S. Hartmann and D. Briskorn, "A survey of variants and extensions of the resource-constrained project scheduling problem," *European Journal of Operational Research*, vol. 207, no. 1, pp. 1 – 14, 2010.

[7] Q. Tian, L. Li, L. Jin, and X. Bai, "A novel dynamic priority scheduling algorithm of process engine in soa," *Web Services, IEEE International Conference on*, vol. 0, pp. 711–718, 2009.

[8] P. Leitner, A. Michlmayr, F. Rosenberg, and S. Dustdar, "Monitoring, prediction and prevention of sla violations in composite services," in *Web Services (ICWS), 2010 IEEE International Conference on*, 2010, pp. 369 –376.

[9] W. Van Der Aalst, M. Rosemann, and M. Dumas, "Deadline-based escalation in process-aware information systems," *Decision Support Systems*, vol. 43, no. 2, pp. 492–511, 2007.

[10] P. Leitner, W. Hummer, and S. Dustdar, "Cost-based optimization of service compositions," Vienna University of Technology, Tech. Rep., 2011.

[11] M. Ferber, S. Hunold, and T. Rauber, "Load balancing concurrent bpel processes by dynamic selection of web service endpoints," in *Parallel Processing Workshops, 2009. ICPPW '09. International Conference on*, 2009, pp. 290 –297.

[12] C. Patel, K. Supekar, and Y. Lee, "A QoS oriented framework for adaptive management of web service based workflows," in *Database and Expert Systems Applications*. Springer, 2003, pp. 826–835.

[13] L. Zeng, H. Lei, and H. Chang, "Monitoring the QoS for Web services," *Service-Oriented Computing–ICSOC 2007*, pp. 132–144, 2010.

[14] J. Howe, "The rise of crowdsourcing," http://www.wired.com/wired/archive/14.06/crowds.html, Jun. 2006.

[15] D. Brabham, "Crowdsourcing as a model for problem solving: An introduction and cases," *Convergence*, vol. 14, no. 1, p. 75, 2008.

[16] Amazon Mechnical Turk, http://www.mturk.com, Feb. 2011.

[17] M. Ford *et al.*, "Web Services Human Task (WS-HumanTask), Version 1.0." 2007.

[18] A. Agrawal *et al.*, "WS-BPEL Extension for People (BPEL4People)." 2007.