A Cost Efficient Scheduling Strategy to Guarantee Probabilistic Workflow Deadlines

Thomas Bach Muhammad Adnan Tariq Boris Koldehofe Kurt Rothermel
Institute for Parallel and Distributed Systems,
University of Stuttgart, Germany
Email: ⟨firstname.lastname⟩@ipvs.uni-stuttgart.de

***Abstract*—Today, workflows are widely used to model business processes. A recent trend is to use them to model applications in heterogeneous, large-scale distributed systems. In such systems, many, possibly mobile, providers offer independent and interchangeable services that can be used to satisfy the different activities of a workflow. Due to varying server loads, failures, and changing network characteristics, the response time of these services is highly volatile. Thus, it is hard to ensure the timely and reliable execution of workflows depending on such services. A common approach is to invoke several services in parallel to increase the probability of success. This, however, can easily lead to overprovisioning and high cost when needlessly invoked services have to be compensated. In this paper, we investigate the search space between parallel and sequential invocation of services. We propose to invoke independent services staggered over time to ensure timely workflow execution at minimal cost. Evaluations show that our approach reduces the execution cost by up to $85\,\%$ while it guarantees to fulfill activity deadlines with $99.9\,\%$ probability.**

I. Introduction

In many areas, such as logistics [1], healthcare [2], manufacturing, and urban mobility [3], workflows have become the standard tool to model business processes. Workflows consist of interrelated activities that trigger services and other applications over time. Their modular structure allows for optimal refinement of business processes even across different hardware and software platforms.

With the advent of cloud computing and the recent advances in mobile and pervasive computing, business processes are migrating to massively distributed and pervasive environments, where workflows are using different services to fulfill their activities. In such environments, services can be running virtually everywhere in the network. They can range from traditional web-services running in the cloud, to services running on computing nodes nearby users' in the fog [4] [5], or even on the users mobile devices [6]. Thus, workflows have become fully distributed business applications that need to coordinate and ensure the correct execution of their components running on heterogeneous nodes in the network.

This ongoing trend has brought business processes close to the users. In modern processes, users are integrated and given the ability to interactively influence the workflow execution. Such tight interactions have raised a strong need for low response times as users do not want to wait long for the system to respond. An increased delay in response time quickly translates to loosing customers [7] and, therefore, money. Google, for example, reports that a 400 millisecond delay resulted in $0.59\,\%$ less searches per user [8]. This clearly states the need to guarantee the responsiveness of a businesses process within a certain time frame.

978-1-4799-5804-7/15/$31.00 © 2015 IEEE

As low response times are hard to achieve, they have become the greatest vulnerability of modern distributed, service oriented applications. Each workflow-level operation (activity), such as organizing a business trip, invokes a service which in turn may need to invoke other workflows or services, (e.g. hotel booking or airline ticket reservation). As the response time of each activity is in the order of a few milliseconds, the tail of the response time distribution of each service becomes important. If only a fraction of the sequentially invoked services take longer to respond, the delay adds up and leads to a long holdup in the execution of the overall workflow.

Nevertheless, response time characteristics of services used to execute business processes differ greatly, dependent on the available resources and load of the network [9]. For instance, a service in the cloud may be very reliable in general. However, if invoked from a mobile device, its responsiveness may be poor due to packet loss, network latency, and network partitioning [10] [11]. Studies confirm that even the characteristics of services available in fixed networks, i.e., traditional web-services, differ, dependent on their location and the origin of the request [12], [13]. Furthermore, the response time distribution of such services is highly volatile [14]. Measurements gathered by Zhang et. al. [15] also show that typical response time distributions have a long tail. This means that in most cases ($\sim 70\,\%$) a request is answered within milliseconds. However, up to $\sim 15\,\%$ of the responses take more than 5 seconds [12], [13]. As we already experience such problems in fixed infrastructure networks, it is anticipated that these issues will become even more important for mobile and pervasive environments in the future [6].

Redundancy is a common approach to decrease response time (and cope with differed kinds of infrastructure failures [11]). There already exist approaches that redundantly *schedule* several services, to execute a single activity of a workflow. One approach is to invoke several interchangeable services (in terms of functionality) in parallel [11], [16] to reduce the response time of an activity. However, this also induces high cost in terms of communication and execution overhead. In addition, if one activity is fulfilled by several services, but can only be executed once (like booking a ticket), the unnecessary service invocations have to be compensated. This might be very expensive. Another approach is to invoke redundant services sequentially such that an additional service is invoked only when the first service has not responded after a certain timeout [17]. This approach has lower execution cost, as less services are invoked and compensated. The response time of this approach, however, is high as service response times have a long tail and thus the required timeout needs to be long.

In this paper, we explore the search space between these two extremes, i.e., parallel and sequential service invocation. We take the response time characteristics of available services into account and invoke several services with a certain delay (i.e., time-offset). Compared to the parallel approach, our strategy reduces the cost of executing an activity because the services invoked earlier have sufficient time to respond before more services are invoked. Compared to the sequential approach, our strategy decreases the response time of an activity because the time-offset used to schedule services (to execute the activity) is much less than the typical timeout. In general, our strategy allows the flexibility to balance between response time requirements and execution cost. Given a workflow with a certain response time requirement (deadline), the proposed strategy can schedule the services for different activities such that the overall cost of executing the workflow is minimized.

In particular, our contributions are as follows. We solve the problem of scheduling services to guarantee probabilistic workflow deadlines (response time requirements) at minimal cost. To achieve this, we developed methods to divide the workflow-level deadline into activity-level deadlines and schedule services for each activity. To that end, we present two simulated annealing based approaches. A proactive approach, that computes service schedules in advance, and a dynamic approach, that schedules services during runtime and takes actual execution time of the different activities into account. We evaluate these strategies and demonstrate the validity of our approach for real service response time distributions measured by Zhang et. al. [15]. In the evaluations we show that our algorithms find 99.61 % optimal solutions while saving up to 85 % of the cost, compared to scheduling services in parallel. Moreover, compared to calculating the optimal solution our approach is about 100 000 times faster.

II. System Model

We assume that a workflow consists of a set of activities $A = \{a_1, \dots, a_n\}$ and ordering relations that specify the causal order of the activities. The approaches presented in this paper are not specific to any workflow specification language.

A workflow is executed by an Execution Engine (EE) that executes the activities of a workflow sequentially, conforming to the ordering relations specified in the workflow model. The *Enterprise Service Bus* (ESB) maintains a register of all services, available in the network. When the EE executes an activity of the workflow, it requests the ESB to invoke a suitable service implementing the required functionality (cf. Fig. 1). The ESB then sends a message to the respective service including execution instructions and computational parameters. After successfully processing the request, the service sends its response back to the ESB.

All services $s_i \in S$ that implement the functionality required by a specific activity $a_i \in A$, can be used interchangeably but have different response time characteristics. The response time of these services is influenced by the

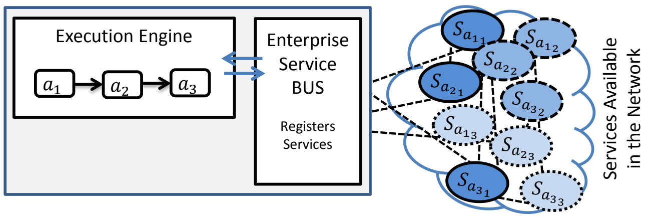

Fig. 1. The system architecture.

computing power and load of service provider [9], but mainly by the network conditions [14], [10]. We model the response time characteristic of a service as a probability distribution function $r_{s_i}(t)$ (cf. Fig. 2) that could be provided by the service provider itself, or monitored by a dedicated component in the ESB. Fig. 2 shows two services (s_1 and s_2), invoked at times τ_{s_1}, and τ_{s_2} and their respective probability distributions $r_{s_1}(t)$ and $r_{s_2}(t)$. For each service s_i, the cumulative distribution function $R_{s_i}(t) = \int r_{s_i}(t)dt$ is calculated and represents the probability that a service s_i responds within a certain time, as a function of t.

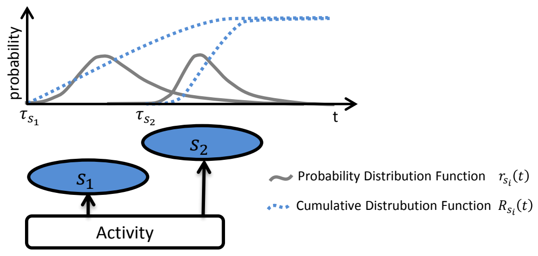
probability
τ_{s_1}
τ_{s_2}
t
s_2
s_1
Activity
Probability Distribution Function $r_{s_i}(t)$
Cumulative Distrubution Function $R_{s_i}(t)$

Fig. 2. Representations of service response time characteristics.

We differentiate between two fundamentally different types of activities. *Idempotent* activities, that can be executed arbitrarily often (e.g., reading sensor data), and *non-idempotent* activities that can be executed at most once (e.g., booking a flight). If, for example, the non-idempotent activity "Book a flight to Dublin" is fulfilled by two services, the system has booked two tickets. In consequence one of the two services needs to be compensated, which might be expensive. In [18] we present a concept how this dynamic service compensation can be implemented. We thus model the total cost of a service s_i as a sum of invocation costs ($c_{s_i}^{inv}$) and service compensation costs ($c_{s_i}^{comp}$). These costs can be communication and computation overhead, energy, money, etc.

III. Problem Formulation

Given a workflow W that consists of $|A|$ activities, our objective is to minimize the overall execution cost of the workflow and ensure that its *final deadline* d_{final} is met with a given *probability* $\geq p$. To this end, two interconnected problems need to be solved:

(1) *Activity-level scheduling problem*: We possibly need to invoke multiple, redundant services in order to execute an activity $a \in A$ within a certain time d_a at minimal cost. The choice of service invocation times offers many possibilities, from sequential execution, partially overlapping execution, to fully parallel execution, as shown in Fig 3a-c

respectively. These possibilities offer different advantages and disadvantages. While parallel scheduling has the advantage to guarantee a short response time, it can yield high service compensation costs when unnecessary service invocations have to be compensated. Sequential service invocation avoids compensation cost, but the services might not respond in time so that the deadline is missed. Our approach is to balance between this extremes and schedule a number of services with a certain temporal offset between each other. To that end, we define a schedule $\tau = \{\tau_{s_1}, \dots, \tau_{s_n}\}$ as a set of service invocation times τ_{s_i} (start times) for each service s_i out of all services S that are available to fulfill an activity a.

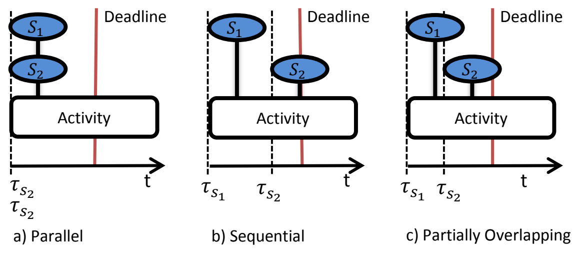

Fig. 3. The two extremes and the compromise.

(2) *Workflow-level scheduling problem:* In addition, for each activity $a \in A$ of a workflow W, we need to find sub-deadlines d_a, that allow W meeting d_{final} with $probability \geq p$. As shown in Fig. 4, the choice of the sub-deadlines determines the choice of services, including their number and therefore influences the costs for executing a certain activity. In Fig. 4a, for example, all deadlines are equally distributed and the services are scheduled partially overlapping for each activity. Assume, that activity B is non-idempotent and that each additional service requires compensation. In this case, it might be cheaper to give more time to the execution of the non-idempotent activity (B) and thus avoid compensation, even if more services need to be scheduled for the idempotent activities A and B (cf. Fig 4b).

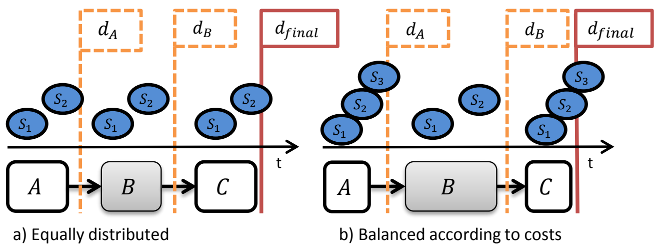

Fig. 4. Shifting of workflow-level deadlines.

Formally, given the services S and a schedule τ, the probability for an activity $a \in A$ to finish within time t, is defined as:

$$\omega_a(t, S, \tau) = 1 - \prod_{s_i \in S} (1 - \int r_{s_i}(t - \tau_{s_i})\, dt) \tag{1}$$

The expected execution costs is defined as:

$$c_a(\tau) = \sum_{s_i \in S} c^{inv}_{s_i} \cdot p^{inv}_{s_i}(\tau) + c^{comp}_{s_i} \cdot p^{comp}_{s_i}(\tau) \tag{2}$$

In Eq. 2, $c^{inv}_{s_i}$ and $p^{inv}_{s_i}(\tau)$ denote the invocation cost and invocation probability of each service s_i dependent on the schedule τ. Further, $c^{comp}_{s_i}$ and $p^{comp}_{s_i}(\tau)$ denote the compensation cost and probability dependent on the schedule τ. The invocation probability of a service $(p^{inv}_{s_i}(\tau))$ is given by Eq. 3 where a service s_i is only invoked if no other service has responded at time τ_{s_i}

$$p^{inv}_{s_i}(\tau) = 1 - \omega_a(\tau_{s_i}, S, \tau) \tag{3}$$

Likewise, the probability that a service needs to be compensated (cf. Eq. 4) is defined as a product of the probability that a service s_i is started at time τ_{s_i} and the probability that no service has answered before s_i.

$$p^{comp}_{s_i}(\tau) = p^{inv}_{s_i}(\tau) \cdot \int_{\tau_{s_i}}^{\infty} r(t) \cdot \omega_a(t, S \setminus s_i, \tau)\, dt \tag{4}$$

We aim to solve the following optimization problem:

$$\begin{aligned} \underset{\tau}{minimize} \quad & \sum_{a \in W} c_a(\tau) && (5)\\ s.t. \quad & \prod_{a \in W} \omega_a(d_a, S, \tau) \geq p && (6)\\ and \quad & \sum_{a \in W} d_a \leq d_{final} && (7) \end{aligned}$$

IV. Scheduling

In this section, we first discuss the complexity of the problem we are going to solve and then describe the *activity-level scheduling* and *workflow-level scheduling* approaches in detail.

A. Complexity

In Sec. II we explained that the cost $c_a(\tau)$ of executing an activity $a \in A$, depends on the services S and their schedule τ. Depended on this, we defined the start probability of each service at time t, $1 - \omega_a(t, S, \tau)$, such that a service is only started if all previous services have not responded so far. Thus, to determine a start point τ_{s_i} of a service s_i the start points of all previous services have to be determined, i.e., all possible combinations have to be evaluated. Formally, let $T = \{t_1, \dots, t_n\}$ represent all possible start times. To calculate the optimal schedule, all possible schedules $\binom{|T|}{|\tau|}$ (combinations of possible start times T and start points of available services τ_{s_i}) must have to be evaluated. The complexity of evaluating all these combinations is in the class of $\mathcal{O}(|T|!)$. Thus the problem is NP-hard. In addition, as time is not discrete $(|T| = \infty)$, there exists an infinite number of possible schedules. Further, to optimize on the workflow-level, we need to find optimal activity-level deadlines d_a. Thus, the above described evaluations need to be done for all possible activity-level deadlines d_a.

To overcome this and to compute the solution in a reasonable time, we chose a simulated annealing based approach. We start with an initial schedule τ and activity deadlines d_a and refine them using different kinds of neighborhood functions. We will discuss this in detail in Sec. IV-B and Sec. IV-C.

B. Activity-level Scheduling

To solve the activity-level scheduling problem, we use an iterative, simulated-annealing-based approach. We start with schedule τ^{curr} (current solution), where all start points τ_{s_i} are in the beginning (cf. Fig. 5a). This ensures that we already have a worst-case solution for that the deadline is fulfilled. However, the cost might not be minimal. In every iteration, a neighborhood function is used to generate a new schedule τ^{new} (new solution) by modifying the current solution τ^{curr}. The new schedule is accepted (becomes the current solution) when it satisfies the activity deadline d_a and has lower cost ($c_{new} = \sum_{a \in W} c_a(\tau^{new})$) than the current solution ($c_c = \sum_{a \in W} c_a(\tau^{curr})$). However, to overcome local minima, we also accept solutions with higher cost with a certain probability δ given by Eq. 8.

$$\delta = e^{-\frac{c_c - c_{new}}{T_c}} \tag{8}$$

In order to calculate this probability, we define a current temperature value T_c that is initialized with a maximum temperature T_{max}. Every iteration this value is decreased, and with it the acceptance probability of worse solutions. When the temperature reaches a specified lower bound the algorithm stops and the current solution (schedule) becomes the final solution.

This process may move the start points of unneeded services after the deadline, so that they are not invoked. In the case of Fig. 5b, two out of all four available services have been selected. Further, the selected services have been scheduled in a way that the combined response probability reaches its maximum exactly at the deadline and not earlier.

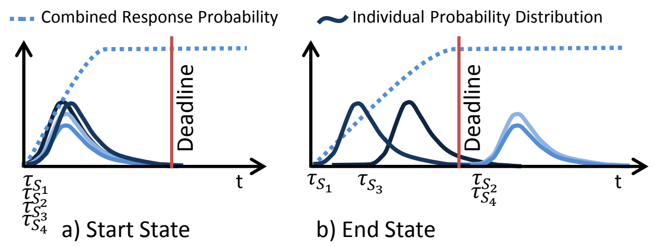

Fig. 5. Simulated Annealing approach for one activity.

The hardest part of designing a simulated annealing algorithm is to define the right neighborhood function, because it is responsible for converting the initial solution into the final solution. To achieve this, the function must cover the whole search space and quickly converge to the global optimum.

In general, two factors influence the quality of the activity-level neighborhood function: The ability to *1) select the right amount of services* and *2)* to decide how to *re-position* the start times of the selected services.

1) The Right Number of Services: To determine the *right amount of services* to re-position we propose two different approaches:

Temperature Dependend Selection selects a number of services (ϑ_T), out of all available services S, based on the current temperature T_C of the system. The idea is that more services are selected and changed in the beginning, (when the temperature is high) to cover the search space. As the temperature decreases, the number of selected services is decreased, so that only small refinements of the solution are possible (cf. Eq. 9).

$$\vartheta_T = \frac{T_c}{T_{max}} \cdot |S| \tag{9}$$

Service Depended Selection selects a number of services (ϑ_S) depended on the total number of services $|S|$ available for the respective activity. The intuition behind this is that constantly a fixed fraction of the services can be changed. The speed and stability of the convergence of the neighborhood function is dependent on a design parameter θ. If θ is chosen too small not enough services are selected and the search space is not covered, for too large values the solution does not converge because the new schedule is a completely different schedule and not a neighbor.

$$\vartheta_S = \frac{|S|}{\theta} \tag{10}$$

2) How to re-position: To determine the new start position of a selected service, we developed three different strategies that will be described below and evaluated in Sec VI.

The *Random Shifting* strategy sets the start point of the selected service to a randomly selected position. This position can either be between the starting time of the activity and the deadline, or at a fixed position after the deadline to disable the service. In general, choosing a random neighbor is the default approach of simulated annealing based algorithms. On the one hand, this has the advantage that the results cover arbitrary search-spaces, on the other hand the solutions converge slowly.

Fixed Shifting shifts the start times of a selected service by a fixed value. Instead of just setting a new position, the intention behind shifting is to gradually change an already good schedule by slightly modifying the current start points. A selected service is shifted to the left or to the right by this fixed shift value. This approach can however only cover the search-space with a certain resolution. As the algorithm needs to work with different deadlines, this fixed shifting value also needs to be dependent on the actual deadline. It needs to be big enough so that a service start point can be moved within the whole search space and also small enough to make fine adjustments to the start points.

Temperature Shifting shifts the start time of a service by a temperature dependent value. The initial shift distance is set to the deadline and is then deceased proportionally to the current temperature (T_c). The intention behind this is, that in the beginning (high T_c) greater adjustments of the schedule are possible as the services can move further and the whole search space can be explored. As the system cools down, (T_c gets smaller) more fine grained adjustments of the start points of the services are possible. This alleviates the fixed resolution problem of the *Fixed Shifting approach*.

It is worth noting that for each activity $a \in A$ a large number of services might be vailable that can be used to fulfill the activity a. We assume that we have already selected a reasonable set of services $|S| \ll \infty$ for every activity, from which our algorithm can choose. For this paper we assume that there exists a metric that can be used to select the “best” available services. This could for example be the n services with the lowest average response time.

C. *Workflow-level Scheduling*

In order to minimize the overall execution costs of the workflow, we have to minimize the total costs of all activities and obey the final deadline d_{final} of the workflow. In consequence, we need to find suitable activity deadlines d_a such that the combined costs of all activities are minimal.

In the following, we present an algorithm that determines the activity-level deadlines similar to the activity-level scheduling algorithm described above. We start with an initial solution, where the time until the final deadline is equally distributed between all activities. The start points of all services S of each activity $a \in A$ are initialized with the activity-level scheduling algorithm presented in Sec. IV-B.

The neighborhood function randomly selects an activity deadline d_a and uses a *Fixed Shifting* based neighborhood function to define a new deadline. This means, the neighborhood function randomly picks an activity deadline and shifts it by a fixed value to the left or right.

When the deadline of an activity has been changed, the schedules of the activities that precede and succeed the changed deadline need to be recomputed. Consider Fig. 6. If deadline d_{a_2} is delayed, activity a_2 has more time to execute and activity a_3 less. Thus we have to recompute the schedules for those activities. To do this, we use the activity-level scheduling approach presented in Sec. IV-B.

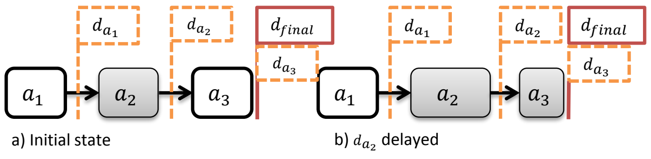

Fig. 6. One activity-level deadline is changed.

V. Scheduling in the Presence of Dynamics

The workflow-level scheduling strategy presented in Sec IV has been designed to schedule the complete workflow in advance, so that each activity deadline d_a is fulfilled with a certain probability (e.g., 99.9 %). Thus, when such a schedule is executed, an activity will most likely be fulfilled before d_a (take less time). Furthermore, especially for long running workflows, the response time behavior and availability of the different services S might change during execution. To take dynamics into account, we present a dynamic scheduling approach that updates the schedules during run time and thus further decreases the execution costs of the workflow.

Before executing the workflow, we have to roughly identify a deadline for each activity. Identification of deadlines is an offline process and is performed in three steps. As a first step, the final deadline of the workflow is divided equally between all activities. The second step is then, to calculate the expected execution cost of each activity by scheduling services for each activity, using the activity-level scheduling approach presented in Sec. IV-B. In the last step activity-level deadlines d_a are adjusted according to the expected execution cost identified in step 2, to give more time to expensive activities.

In more detail, Eq. 11 is used to calculate the activity deadlines d_a. The time to the final deadline (d_{final}) of the workflow is divided into two parts. The first part is equally divided between all activities. The second part is divided proportionally to each activities share of the total expected execution costs $\sum_{a \in W} c_a(\tau)$. The ratio of both parts is determined by the value of λ. In Sec. VI-C we will explain how this λ value was determined.

$$d_a = d_{final} \cdot \lambda \cdot \frac{1}{|A|} + d_{final} \cdot (1-\lambda) \cdot \frac{c_a(\tau)}{\sum_{a \in W} c_a(\tau)} \quad (11)$$

Once the activity deadlines are determined, the execution of workflow starts. We define a scheduling window as a number of $n \ll |A|$ activities that are scheduled right before execution, using the workflow-level scheduling strategy presented in Sec. IV. Initially, the first scheduling window is computed. When the execution approaches the end of a scheduling window, the next window is calculated, taking the actual execution time of the previous (already executed) activities into account. This process continues, until all activities have been executed (cf. Fig. 7).

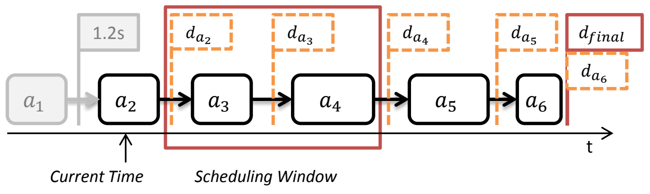

Fig. 7. The dynamic approach.

VI. Evaluations

In this section, we evaluate the scheduling strategies for *activity-level scheduling* and *workflow-level scheduling* presented in Sec IV, w.r.t. deadlines, execution cost, and computation time, in comparison to the optimal solution. Furthermore, we compare the workflow-level scheduling strategy with the dynamic workflow-level scheduling strategy regarding real execution cost.

Our evaluations are based on 30 287 600 response time measurements taken by 142 users querying 4 532 real webservices [13]. From these measurements, we generated response time characteristics $(r_{s_i}(t))$ for all 4 532 Services. For our measurements we used a cluster consing of 24 Intel®Xeon®, 3.00GHz CPUs, with a total of 377.8GB of RAM.

We implemented a simulation framework in Java that was used to develop and test our *activity-level scheduling* strategies, based on the replayed behavior of real services [13]. To evaluate the *workflow-level scheduling* strategies we extended our framework to generate workflows, consisting of up to 150 activities. For each activity, up to 20 services have been selected for scheduling.

A. *Activity-level Scheduling*

In this section, we evaluate the activity-level scheduling approach presented in Sec. IV-B. To evaluate different neighborhood functions and show the benefits of our approach, we compare them with the parallel strategy that uses as few services as needed, started in parallel, to meet the deadline.

On the activity-level, we measure the quality of a solution in terms of the expected number of services. As defined in Sec III, services are only invoked if all previous services have not respondent so far. Thus, the number of services invoked is probabilistic. The expected number of services is therefore an estimate of needed services to successfully fulfill an activity until a certain deadline.

We also estimate how "optimal" our solution actually is. To that end, we compare the results of our strategy to a brute force strategy, that calculates and evaluates "all possible" schedules. As the number of schedules is infinite (since time is continuous Sec. III) this can only be done approximately. In order to do that we use an iterative algorithm that divides the search-space into k start positions and evaluates all possible combinations of start positions and services $\binom{k}{|S_a|}$. With every iteration, k is increased, until no significant increase in accuracy is detected.

In our system, we model response time requirements in the form of deadlines that need to be met (cf. Sec II). Therefore we evaluate our approaches for various deadlines and compare the resulting expected number of services.

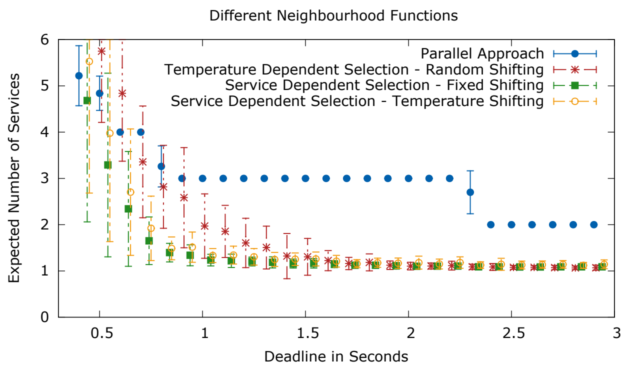

Fig. 8. Comparing different neighborhood functions for different deadlines.

In Sec. IV-B we highlighted that a neighborhood function needs to *select* and *re-position* a number of service start points (τ_{s_i}) and presented methods to do this. We evaluated all combinations of service selection and start point re-positioning strategies. In Fig. 8, we present the performance of the most promising combinations and compare them to the parallel provisioning approach. All presented neighborhood functions are named after the combination of strategies they use, e.g. *Temperature Depended Selection - Random Shifting* uses a temperature depended service selection and a random service selection strategy.

For the evaluation shown in Fig. 8, we randomly selected 16 services out of all 4 532 available services. We evaluated every neighborhood function for 26 deadlines in the range between 0.3 and 3.0. We repeated this experiment 60 times and plotted the median and standard deviation.

Simulations show, that *Service Dependent Selection - Fixed Shifting* is outperforming all other strategies, especially for short deadlines. We can also see that the parallel approach shows a discrete step behavior. This is because all required services are started in parallel at the beginning. As expected, for tight deadlines more services are scheduled, but even for loose deadlines, at least two services are required. The reason for this is that response time characteristics have a long tail. Thus, one service is not enough to guarantee that a single service responds within the first three seconds.

Most interesting are the results for short deadlines, as here the strengths and weaknesses of the different strategies are exposed. Several experiments showed that neighborhood functions using *Service Dependent Selection* outperform random and temperature depended selections strategies, because they ensure a certain amount of change between the current state and the new neighbor. One example for such an approach is the "Temperature Dependent Selection - Random Shifting" approach depicted in Fig. 8.

In consequence, we also evaluated service dependent selection with fixed shifting and temperature dependent shifting of services. An initial expectation might be, that temperature depended shifting would be able to cover the search space more efficient as it starts with bigger steps. This however is not reflected in our experiments. As we can see in Fig. 8 *Temperature Shifting* is also outperformed by *Service Dependent Selection - Fixed Shifting* especially for short deadlines. The reason for this is that the changes made by the temperature based approach are too large at high temperatures and too small at low temperatures. The shifting range at medium temperatures equals then exactly the fixed shifting range of *fixed shifting* approaches.

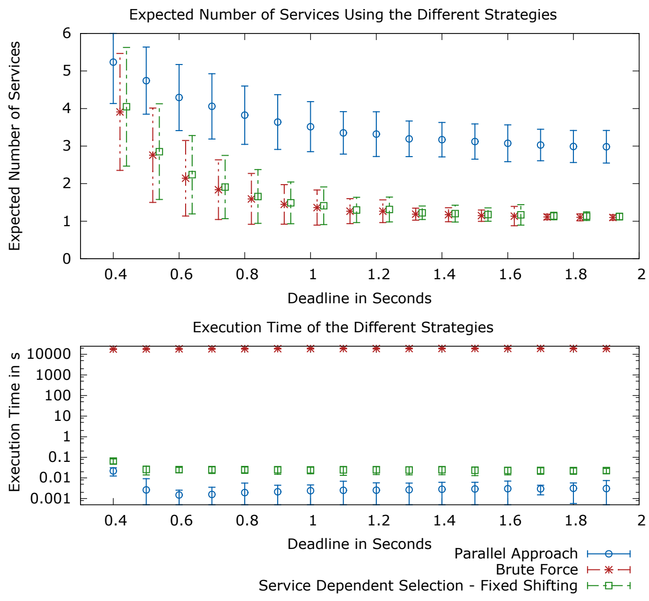

Fig. 9. Calculation time and expected number of services for 8 Services and various deadlines.

Fig. 9 compares the runtime of our best approach (Service Dependent Selection - Fixed Shifting) with the parallel provisioning approach and the brute force approach that generates the near-optimal solution. It shows that the solutions generated by our strategy are on average 99.61 % as good as the solutions generated by brute force. Furthermore, they only take a fraction of the time to compute. It also shows that our approach is almost as fast as just selecting the minimum number of services that need to be invoked in parallel, while producing significantly better results.

B. *Workflow-level Scheduling*

Fig. 10 compares the workflow-level scheduling strategy described in Sec. IV-C with the parallel provisioning approach. For this evaluation we computed 150 workflows for each deadline. Each workflow consisted of 20 activities. For each activity, the algorithms had to choose from 18 different services. To model compute intense activities, we introduced a random length factor between 1 and 4 that scaled the response time of the activities. In this experiment, the execution cost of a service was set to 1 and the compensation cost for non-idempotent activities to 100. We repeated this experiment 3 times and used workflows where 0 %, 50 %, and 100 % of the activities have been idempotent. We averaged the results and printed the median and standard deviation. As expected, execution cost increased with an increasing number of non-idempotent activities. We can also see that the expected execution costs of our approach is $3-4$ times lower compared to the parallel provisioning approach.

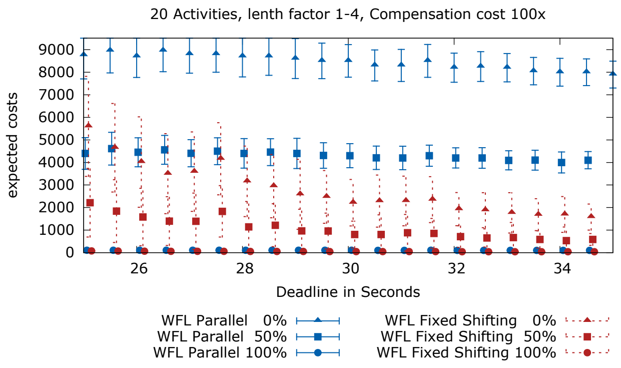

Fig. 10. Comparing the naive parallel with the proactive scheduling approach

C. *Scheduling in the Presence of Dynamics*

In this section, we compare the workflow-level scheduling strategy presented in Sec. IV-C to the dynamic scheduling strategy (Sec. V) and the parallel strategy. Furthermore, we show how we determined the value for λ.

As described in Sec. V, the dynamic scheduling approach determines the deadlines based on the expected cost of the different activities and the total available time. The key factor to determine the time for each activity is the balancing between fixed share of time and cost depended share of time (λ) each activity receives. Fig. 11 shows that the lowest costs can be achieved, if half of the total available time is equally distributed between all activities ($\lambda \approx 0.5$). The other half of the time should be assigned based on each activities share of the total cost of the workflow. For this evaluation, we tested 500 different workflows using 50 different values for λ in the range $[0.0, 1.0]$.

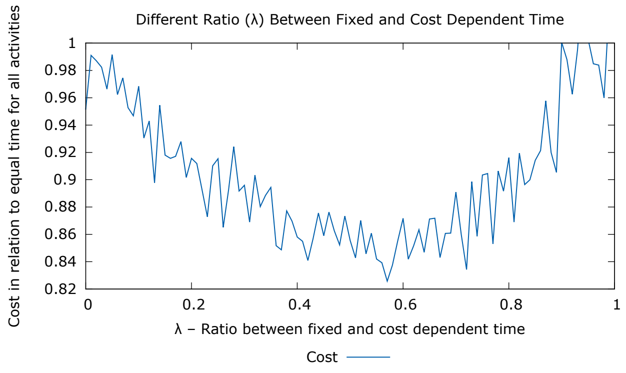

Fig. 11. The benefit of different values for λ.

In Fig. 12 we compare the real execution cost of the schedules produced by dynamic scheduling, workflow-level scheduling and parallel scheduling. To this end, we generated workflows consisting of 100 activities where again 0 %, 50 %, and 100 % of the activities have been idempotent. We set the execution cost of a service to 1 and the compensation cost for non-idempotent activities to 100. We performed 900 experiments for deadlines in the range of $[25.0, 35.0]$ seconds and plotted the mean values and standard deviation.

To simulate the execution of a workflow schedule, we determined an actual response time for each service from its response time distribution and evaluated it against the produced schedules. Fig. 12 shows that schedules produced by the dynamic scheduling strategy result in the lowest execution cost. Even executing workflow schedules where all activities are non-idempotent is about 4 times faster than executing workflows with 50 % non-idempotent activities scheduled by the workflow-level scheduling or the parallel approach. As workflow-level scheduling produces schedules that meet every deadline with a very high probability, most of the activities are fulfilled earlier than the deadline. The dynamic strategy can make use of this buffer and dynamically allocate more time for non-idempotent activities to reduce costs.

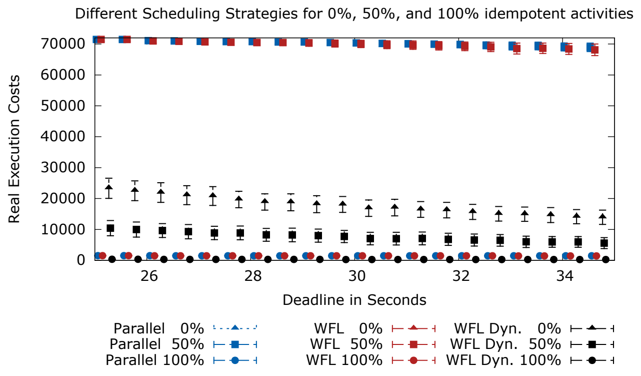

Fig. 12. Comparing the real execution cost of all strategies.

VII. Related Work

Since low latency and timely execution of workflows are important requirements, these problems have already been approached in different ways.

A common approach to increase the reliability of a workflow is to schedule several services in parallel to fulfill one activity [11]. The problem with these approaches is that they are likely to schedule too many services (over provisioning), as we presented in our evaluations. This can lead to high cost, especially if the activity is non-idempotent and unnecessary services require expensive compensation.

A similar approach is to select a small number of services, e.g. 4 and schedule them in parallel. If none of them responds within a certain time, a second set of services is started [19], [16], [17]. These approaches, however, can still lead to the invocation of too many services and high cost. To avoid over provisioning, different quality monitoring, forecasting, and dynamic binding approaches [20], [14] can be used. Such approaches can only help to select the right services and reduce the time between service selection and invocation. They can neither guarantee a certain responsiveness, nor provide mechanisms to ensure that an activity is fulfilled until a certain deadline. This is because there exist no backups that can take over in the case of failures. Furthermore they cannot efficiently mask problems that are due to network failures.

A different approach to ensure the availability and successful execution of workflows in distributed environments has been proposed in [21]. Instead of scheduling several services to fulfill the same activity, several structurally different copies of the same workflow are executed in parallel. The execution of the differently ordered workflows is then coordinated so that non-idempotent activities are not executed at the same time by different copies of the workflows. To employ this strategy, the workflow needs to be specified in a declarative workflow language. One problem with this approach is that for large workflows, the generation of the structurally different copies of the workflow can become extremely complex.

Furthermore, we already presented [18] how the approach described in this paper could be combined with a workflow replication and dynamic service compensation strategy to ensure high availability in Collective Adaptive Systems.

VIII. Conclusion

Applications running in distributed and pervasive environments face many different kinds of failures. When workflows are used to model applications and business processes in such environments timely execution is a serious issue. In this paper, we have presented a new approach to guarantee probabilistic deadlines for workflows executed in such environments. Our approach takes the type of the executed activity into account and significantly decreases the expected execution costs compared to the state of the art. Additionally, even tight deadlines can be guaranteed with high probability. In particular, we have developed methods to divide the workflow-level deadline into activity-level deadlines and schedule several services staggered over time such that the expected execution cost are minimal. Furthermore, we have extended this concepts to cope with high dynamics. Future research in this area will be on further optimizing the proposed method to deal with branching within workflows.

Acknowledgment

The authors would like to thank the European Union's Seventh Framework Programme for partially funding this research through the ALLOW Ensembles project (project 600792).

References

[1] H. Wolf, K. Herrmann, and K. Rothermel, "Modeling dynamic context awareness for situated workflows," in *On the Move to Meaningful Internet Systems Workshops*. Springer, 2009.

[2] ——, "Flexcon – robust context handling in human-oriented pervasive flows," 2011.

[3] K. Herrmann, K. Rothermel, G. Kortuem, and N. Dulay, "Adaptable pervasive flows-an emerging technology for pervasive adaptation," in *IEEE Self-Adaptive and Self-Organizing Systems Workshops*, 2008.

[4] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proc. 1st workshop on Mobile Cloud Computing*. ACM, 2012.

[5] G. Wu, S. Talwar, K. Johnsson, N. Himayat, and K. D. Johnson, "M2m: From mobile to embedded internet," *Commun. Mag., IEEE*, 2011.

[6] F. AlShahwan, K. Moessner, and F. Carrez, "Providing light weight distributed web services from mobile hosts," in *Proc. IEEE Int. Conf. Web Services*, 2011.

[7] A. Vulimiri, O. Michel, P. Godfrey, and S. Shenker, "More is less: reducing latency via redundancy," in *ACM Workshop on Hot Topics in Networks*, 2012.

[8] S. Souders. Velocity and the bottom line. [Online]. Available: http://programming.oreilly.com/2009/07/velocity-making-your-site-fast.html

[9] Q. Wang, Y. Kanemasa, J. Li, D. Jayasinghe, M. Kawaba, and C. Pu, "Response time reliability in cloud environments: An empirical study of n-tier applications at high resource utilization," in *SRDS*, 2012.

[10] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly available transactions: Virtues and limitations." *PVLDB*, 2013.

[11] A. Vulimiri, P. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker, "Low latency via redundancy," *arXiv preprint arXiv:1306.3707*, 2013.

[12] Z. Zheng, Y. Zhang, and M. R. Lyu, "Investigating qos of real-world web services," *IEEE Transactions on Services Computing*, 2012.

[13] Z. Zheng and M. R. Lyu, "Collaborative reliability prediction for service-oriented systems," in *Proc. IEEE/ACM 32nd Int. Conf. Software Engineering*, 2010.

[14] A. Amin, A. Colman, and L. Grunske, "An approach to forecasting qos attributes of web services based on arima and garch models," in *Proc. IEEE Int.Conf. Web Services*, 2012.

[15] Y. Zhang, Z. Zheng, and M. R. Lyu, "Wspred: A time-aware personalized qos prediction framework for web services," in *Proc. IEEE Symposium on Software Reliability Engineering*, 2011.

[16] S. Stein, T. R. Payne, and N. R. Jennings, "Robust execution of service workflows using redundancy and advance reservations," *Services Computing, IEEE Transactions on*, 2011.

[17] H. Guo, J. Huai, H. Li, T. Deng, Y. Li, and Z. Du, "Angel: Optimal configuration for high available service composition," in *IEEE Web Services*, 2007.

[18] D. R. Schäfer, S. G. Sáez, T. Bach, V. Andrikopoulos, and M. A. Tariq, "Towards Ensuring High Availability in Collective Adaptive Systems," in *Proc. 1st Int. Workshop of Business Processes in Collective Adaptive Systems: BPCAS'14; Eindhoven, Netherlands*, 2014.

[19] S. Stein, E. Gerding, A. Rogers, K. Larson, and N. R. Jennings, "Flexible procurement of services with uncertain durations using redundancy." in *IJCAI*, 2009.

[20] B. Cavallo, M. Di Penta, and G. Canfora, "An empirical comparison of methods to support qos-aware service selection," in *Proc. of the 2nd Int. Workshop on Principles of Engineering Service-Oriented Systems*. ACM, 2010.

[21] D. R. Schäfer, T. Bach, M. A. Tariq, and K. Rothermel, "Increasing Availability of Workflows Executing in a Pervasive Environment," in *Proc. IEEE Int. Conf. on Services Computing, Anchorage, Alaska, USA*. IEEE, 2014.