Provisioning Heterogeneous and Unreliable Providers for Service Workflows

Sebastian Stein and **Nicholas R. Jennings** and **Terry R. Payne**
School of Electronics and Computer Science
University of Southampton
Southampton, SO17 1BJ, UK
{ss04r,nrj,trp}@ecs.soton.ac.uk

Abstract

Service-oriented technologies enable software agents to dynamically discover and provision remote services for their workflows. Current work has typically assumed these services to be reliable and deterministic, but this is unrealistic in open systems, such as the Web, where they are offered by autonomous agents and are, therefore, inherently unreliable. To address this potential unreliability (in particular, uncertain service durations and failures), we consider the provisioning of abstract workflows, where many heterogeneous providers offer services at differing levels of quality. More specifically, we show that service provisioning is NP-hard, and then devise two heuristic strategies that use service redundancy in a flexible manner to address uncertainty and failure. In empirical experiments, we show that these heuristic strategies can achieve significant improvements over standard approaches in a wide range of environments.

Introduction

Web services and other service-oriented technologies are emerging as popular tools for building and deploying large-scale distributed systems, where autonomous agents provide remote resources and functionality to each other (Huhns & Singh 2005). By using semantically enriched service descriptions, agents in such systems are able to discover and invoke services without human intervention. Often, these services are invoked as part of workflows, which describe the types of services and ordering constraints required to meet a high-level goal.

Now, a key feature of large distributed systems is that the behaviour of service providers is usually uncertain. On one hand, this is due to the dynamic and open nature of such systems, where network failures, competition for resources and software bugs are common. Furthermore, providers are often self-interested agents (Jennings 2001), and may therefore decide to postpone or even ignore service requests if it is best for them to do so. Given this, such unreliability must be addressed when executing large workflows of interdependent tasks, where service failures and delays can easily jeopardise the overall outcome.

To date, existing work on service-oriented computing has largely overlooked the potential unreliability of service providers, and instead assumed truthful and deterministic service descriptions when executing workflows. Some work has considered reliability as a constraint on service selection or as part of a quality-of-service optimisation problem (Zeng *et al.* 2003). However, such an approach requires a human user to manually choose appropriate constraints and weights that balance the various quality-of-service parameters (which are usually assumed independent of each other). This shortcoming has partially been addressed by work that uses utility theory to choose among unreliable service providers (Collins *et al.* 2001). However, these approaches tend to rely only on a single service provider for each task in a workflow (which makes them vulnerable to particularly unreliable providers) and do not consider uncertain service durations in a satisfactory manner.

To address this, we proposed a strategy that uses stochastic performance information about service providers to flexibly provision multiple providers for particularly failure-prone tasks in an abstract workflow (Stein, Jennings, & Payne 2006). In that work, we showed that this redundancy allows the consumer to deal with uncertain service behaviour. However, we did not consider potential performance differences between providers that offer the same service type. This limits the strategy to systems where providers are homogeneous or where specific information about individual providers is not available.

In this paper, we substantially extend our previous work by addressing service heterogeneity. More specifically, we advance the state of the art in service provisioning in the following two ways. First, we formalise the service provisioning problem with heterogeneous providers and prove that it is NP-hard, thus providing a strong motivation for the use of heuristic strategies in this field. Second, we describe two novel strategies for solving this problem. In particular, these flexibly provision multiple heterogeneous providers for each task in the workflow so as to balance the expected reward of completing the workflow with its costs.

The remainder of this paper is structured as follows. Next, we formalise the service provisioning problem and investigate its complexity. Then, we describe our heuristic strategies for provisioning, followed by a detailed discussion of our experimental results. Finally, we conclude.

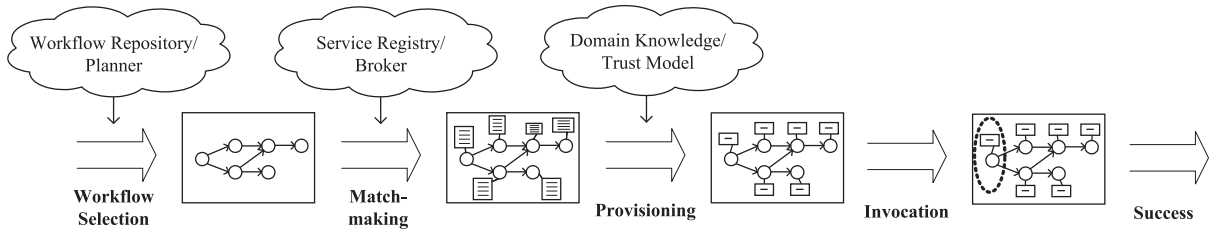

Figure 1: Lifecycle of a workflow.

Service Provisioning Problem

In our work, we assume that a service consumer has an abstract workflow containing task descriptions and ordering constraints required to meet some high-level goal. In practice, this may originate from a repository of pre-defined plans or may be synthesised at run-time using planning and composition techniques. We also assume that the consumer has identified suitable service providers for each task in the workflow, using an appropriate matchmaking process (e.g., by contacting a service registry or broker). In this context, we focus on *provisioning* these providers, i.e., deciding which to invoke for each of the tasks (the overall lifecycle of a workflow is shown in Figure 1). In the following, we first formalise our system model, then define the provisioning problem and show that it is NP-hard.

Formal Model of a Service-Oriented System

A *workflow*, as selected during the first stage of Figure 1, is a directed acyclic graph $W = (T, E)$, where $T = \{t_1, t_2, t_3, \ldots, t_{|T|}\}$ is a set of *tasks* and $E : T \leftrightarrow T$ is a set of *precedence constraints* (a strict partial order on T). Furthermore, $u(t) \in \mathbb{R}$ denotes the *reward* of completing the workflow at time step $t \in \mathbb{Z}_0^+$. This is expressed by a maximum reward, $u_{\max}$, a deadline, d, and a cumulative penalty for late completion, δ:

$$u(t) = \begin{cases} u_{\max} & \text{if } t \leq d \\ u_{\max} - \delta(t-d) & \text{if } t > d \wedge t < d + \frac{u_{\max}}{\delta} \\ 0 & \text{if } t \geq d + \frac{u_{\max}}{\delta} \end{cases} \quad (1)$$

The *service providers* in a service-oriented system are given by $S = \{s_1, s_2, s_3, \ldots, s_{|S|}\}$. To associate workflow tasks with appropriate providers, there is a *matching function* m that represents the result of the matchmaking stage in Figure 1. This maps each task $t_i \in T$ to a set of service providers $m(t_i) \in \wp(S)$ whose members can accomplish the task (they provide the same *type* of service). We also assume that some information is known about each provider s_i (in practice, this might be learnt through previous interactions or obtained through a trust model (Teacy *et al.* 2006)):

- $f(s_i) \in [0, 1]$ is the *failure probability* of provider s_i (i.e., the probability that the provider will default or fail to provide a satisfactory result),
- $D(s_i, t) \in [0, 1]$ is the *(cumulative) duration distribution* of provider s_i (i.e., the probability that the provider will take $t \in \mathbb{Z}^+$ time steps or less to complete an invocation),
- $c(s_i) \in \mathbb{R}$ is the *invocation cost* of provider s_i (this may be a financial remuneration for the provided service or a communication cost).

Furthermore, we introduce the notion of *service populations*, to group sets of services whose behaviour is assumed identical. Hence, there is a set partition of S, $P = \{P_1, P_2, P_3, \ldots, P_{|P|}\}$, whose members are disjoint subsets of S with $\bigcup_i P_i = S$. Any two members s_x and s_y of a given population P_i always have the same failure probability, duration distribution and cost, and each task that is mapped to s_x by m is also mapped to s_y.

This notion is introduced for convenience, because we believe that it is a common feature of distributed systems, where a number of agents may use the same service implementation, might adhere to certain quality standards, or where the consumer's knowledge about service providers is limited (e.g., in the absence of more accurate information, a service consumer may simply classify $P_1 = \{s_1, s_2, s_3\}$ as cheap, unreliable providers, and $P_2 = \{s_4, s_5\}$ as reliable, but expensive providers).

During the invocation stage, a workflow W is completed by invoking appropriate providers for each task, according to the precedence constraints E. When invoked, the consumer first pays a cost, $c(s_i)$, then the service provider completes the task successfully with probability $1 - f(s_i)$ in a random amount of time distributed according to $D(s_i, t)$. As the service is executed by an autonomous agent, we assume that the outcome of an invocation is not known to the consumer until this time has passed (if successful), and that a failure is not communicated at all. Furthermore, a single provider can be invoked for several distinct tasks in the workflow, but not repeatedly for a given task. When multiple providers are invoked for a task, it is completed when the first provider returns a successful response. Finally, to evaluate the performance of a consumer, we define the *net profit* of a workflow execution as the difference of the reward given by $u(t)$ (or 0 if unsuccessful) and the total cost incurred.

Problem Definition and Complexity

During the provisioning stage, a consumer decides which service providers to invoke for the tasks of its workflow. To formalise this problem, we let $\alpha \in (T \rightarrow \wp(S \times \mathbb{Z}_0^+))$ be a *provisioning allocation* that maps workflow tasks to sets of service providers and associated invocation times. These times are relative to the time a task first becomes executable and indicate in what order different service providers

should be invoked for the task (given that it has not already been completed). For example, the provisioning allocation $\alpha(t_1) = \{(s_1, 0), (s_2, 100)\}$ indicates that up to two service providers will be invoked for task t_1: s_1 and s_2. The first, s_1, will be invoked as soon as t_1 becomes available, and s_2 will be invoked if s_0 was still unsuccessful after 100 time units. With this, we formulate service provisioning as follows:

Definition 1. *(*PROVISIONING*): Given a workflow W, utility function u, matching function m and quality functions f, D and c, find a provisioning allocation α^* that maximises the expected net profit of a consumer following it.*

Theorem 1. PROVISIONING *is NP-hard.*

We prove this by giving a polynomial time transformation from an instance of the NP-complete KNAPSACK problem to a PROVISIONING instance.

Definition 2. *(*KNAPSACK*): Given a finite set of items $I = \{1, 2, 3, \ldots, N\}$, a weight $w(i) \in \mathbb{Z}^+$ and a value $v(i) \in \mathbb{Z}^+$ for each item $i \in I$, an overall capacity $C \in \mathbb{Z}^+$ and a target value $V \in \mathbb{Z}^+$, decide whether there is a subset $I' \subseteq I$, so that $\sum_{i \in I'} w(i) \leq C$ and $\sum_{i \in I'} v(i) \geq V$.*

Proof. Let $v_{\max}$ be the highest value of any item in I. Then, for every item $i \in I$, create a service provider s_i with $f(s_i) = 0$, $c(s_i) = v_{\max} - v(i) + 1$ and define $D(s_i, t)$ so that the service duration is always exactly $w(i) + 1$ time units (i.e., $D(s_i, t) = 0$ if $t < w(i) + 1$ and $D(s_i, t) = 1$ if $t \geq w(i) + 1$). Also, create a service provider s_0 with $f(s_0) = 0$, $c(s_0) = v_{\max} + 1$ and define $D(s_0, t)$, so that the service duration is always exactly 1. Create workflow $W = (T, E)$ with $T = \{t_1, t_2, \ldots t_N\}$ and let E be any total order on T. Now define a matching function m, so that $m(t_i) = \{s_i, s_0\}$ for all t_i. Finally, create utility function u with deadline $d = N + C$, maximum utility $u_{\max} = N(v_{\max} + 1) - V + \frac{1}{2}$ and penalty $\delta = u_{\max}$. This transformation is performed in $O(N)$. It is easy to see that a non-empty solution to this new PROVISIONING instance exists if and only if the answer to the original KNAPSACK instance is "yes". □

Heuristic Provisioning

In order to deal with the inherent difficulty of service provisioning, we outline two heuristic strategies in this section. The first (*full flexible*) performs a local search for a good provisioning allocation using a novel heuristic utility estimation function. This strategy considers a large solution space that covers all feasible choices for α. However, because it may potentially take a long time to converge to a good solution, our second strategy (*fast flexible*) considers provisioning at the higher level of service populations (rather than individual providers), thus reducing the space of candidate solutions and allowing the search to converge faster.

Full Flexible Provisioning

Formulated as an optimisation problem, the aim of our flexible provisioning strategy is to find an optimal allocation α^*:

$$\alpha^* = \arg\max_{\alpha}(\bar{r}(\alpha) - \bar{c}(\alpha)) \quad (2)$$

Algorithm 1 Local search to find a good allocation α^*.

```
1:  α ← CREATEINITIAL(P)                          ▷ Initial solution
2:  ũ ← ESTIMATEUTILITY(α, P)
3:  failed ← 0
4:  repeat                                        ▷ Main loop
5:      failed ← failed + 1
6:      T' ← T
7:      while failed > 0 ∧ |T'| > 0 do
8:          t_i ∈ T'                              ▷ Random choice
9:          T' ← T' \ t_i
10:         N_{α,t_i} ← GENERATENEIGHBOURS(α, t_i, P)
11:         for all α' ∈ N_{α,t_i} do
12:             ũ' ← ESTIMATEUTILITY(α', P)
13:             if ũ' > ũ then
14:                 (α, ũ) ← (α', ũ')
15:                 failed ← 0
16:             end if
17:         end for
18:     end while
19: until failed ≥ maxFailed
20: return α
```

where $\bar{r}(\alpha)$ is the *expected reward* of following α and $\bar{c}(\alpha)$ the *expected cost*. However, as we have shown, this problem is inherently difficult. For this reason, we chose local search as a commonly used heuristic technique to find good solutions for intractable problems (Michalewicz & Fogel 2004). The local search algorithm we use is given in Algorithm 1 and follows the common structure of such methods.

In more detail, the algorithm takes as input a PROVISIONING instance $\mathcal{P}$ and starts with a random allocation α (line 1), which is then iteratively improved, based on an estimated utility value (lines 4 – 19). During each iteration, the algorithm picks a random task t_i from the workflow (line 8), and considers each of a set of neighbours of α, which are obtained by randomly applying small changes to the provisioned service providers for task t_i. During this process, the algorithm keeps track of the best neighbour so far, which is then used as the new allocation α for the following iteration (line 14). If no better neighbour is found for task t_i, the algorithm continues to consider all other tasks in a random order. It terminates when the main search loop is executed `maxFailed`[1] times without discovering a better solution, at which point the current α is returned (line 20).

Algorithm 1 depends on three functions: CREATEINITIAL, GENERATENEIGHBOURS and ESTIMATEUTILITY. Respectively, these create an initial solution, generate neighbour allocations of a given α and estimate its utility. We describe the first two briefly below, followed by a more detailed discussion of ESTIMATEUTILITY.

Initial Provisioning Allocation Creation: Initially, we provision a random[2] non-empty subset of the matching

[1] This accounts for the fact that we select random neighbours and may miss potentially better solutions (we set this to 10).

[2] Unless stated otherwise, a random choice is picked uniformly at random from the set of all candidates.

providers for each task t_i, and assign a random time to each provider from the interval $\{0, 1, 2, \ \ldots, t_{\max} - 1\}$, where $t_{\max}$ is the first time step at which the consumer receives no more reward from the workflow (i.e., $u(t_{\max} - 1) > u(t_{\max}) = 0$). Finally, for each task t_i, we find the lowest invocation time of any provisioned provider for t_i, and deduct it from all invocation times for t_i. This ensures that there are no unnecessary delays before the first invocation.

Neighbour Generation: We generate neighbours of a particular provisioning allocation α by considering only a given task t_i (see Algorithm 1, line 10). To this end, we first pick a random population (that has at least one provider provisioned) and then a random provisioned member s_x of this population. Finally, the following transformations are applied separately to α, where possible, in order to generate the set of neighbours ($\mathcal{N}_{\alpha,t_i}$):

- Provider s_x is removed from α.
- Provider s_x is replaced by another suitable provider of a different population (using the same time).
- A random unprovisioned provider s_y of the same population as s_x is provisioned for the same time as s_x.
- The time for s_x is changed (increased and decreased in integer and random steps, yielding four new neighbours).
- A random unprovisioned provider s_z from any suitable population is provisioned at a random time.

We again alter the provisioning times of all new neighbours to ensure that there are no unnecessary delays.

Heuristic Utility Estimation: The objective of our utility estimation function is to estimate the expected utility of an allocation, as shown in Equation 2. To solve this exactly is intractable, as the expected reward $\bar{r}(\alpha)$ requires the distribution of the overall workflow completion time — a problem that is known to be $\#P$-complete (Hagstrom 1988). Hence, our heuristic function uses the overall success probability of the workflow (p), a probability density function for the estimated completion time if successful ($d_W(x)$) and an estimated cost ($\tilde{c}$) to estimate the utility ($\tilde{u}$) as follows (omitting parameters for brevity):

$$\tilde{u} = p \int_0^\infty d_W(x) u(x) \,\mathrm{d}x \ - \tilde{c} \tag{3}$$

These parameters are calculated in two steps. First, we determine a number of *local* parameters for each task t_i, and then, these are combined to give p, $d_W(x)$ and $\tilde{c}$. For the local calculations, we let $\hat{D}(s_x, t) = (1 - f(s_x)) \cdot D(s_x, t)$ be the probability that a service provider s_x has completed its service successfully within no more than t time steps (not conditional on overall success). Furthermore, we let $S_i(\alpha, t)$ be the set of provisioned service providers and associated times that are invoked at most t time steps after task t_i was started. Then, $T_i(\alpha, t) = 1 - \prod_{(x,y)\in S_i(\alpha,t)}(1 - \hat{D}(x, t - y))$ is the probability that task t_i was completed successfully within no more than t time steps. With this, we calculate the following four local parameters for each task t_i:

- **Success Probability (p_i):** This is the probability that t_i will be completed successfully no later than $t_{\max}$ time steps after the task was started:

$$p_i = T_i(\alpha, t_{\max}) \tag{4}$$

- **Mean Completion Time (λ_i):** This is the mean time until the task is completed, conditional on the task being successfully completed within $t_{\max}$ time steps[3]:

$$\lambda_i = \frac{1}{p_i} \sum_{t=1}^{t_{\max}} t \cdot (T_i(\alpha, t) - T_i(\alpha, t-1)) \tag{5}$$

- **Variance (v_i):** This is the variance of the completion time, conditional on successful completion as above:

$$v_i = -\lambda_i^2 + \frac{1}{p_i} \sum_{t=1}^{t_{\max}} t^2 \cdot (T_i(\alpha, t) - T_i(\alpha, t-1)) \tag{6}$$

- **Expected Cost (c_i):** The total cost the consumer is expected to spend on this task:

$$c_i = \sum_{(x,y)\in\alpha(t_i)} (1 - T_i(\alpha, y)) \cdot c(x) \tag{7}$$

Next, these parameters are aggregated to obtain the overall estimated utility of the allocation. This process is similar to the one described in (Stein, Jennings, & Payne 2006) with the notable difference that task duration variance is taken into account to give a more accurate estimate of the duration distribution. To this end, we first calculate the overall success probability of the workflow (p) as the product of all individual success probabilities ($p = \prod_{\{i \mid t_i \in T\}} p_i$).

The estimated cost for the workflow is the sum of the expected cost of all tasks, each multiplied by the probability that it is reached (i.e., all its predecessors are successful):

$$\tilde{c} = \sum_{\{i \mid t_i \in T\}} r_i c_i \tag{8}$$

$$r_i = \begin{cases} 1 & \text{if } \forall t_j \cdot ((t_j \mapsto t_i) \notin E) \\ \prod_{\{j \mid (t_j \mapsto t_i) \in E\}} p_j & \text{otherwise} \end{cases}$$

Finally, to estimate the duration distribution of the workflow, we employ a technique commonly used in operations research (Malcolm *et al.* 1959), and evaluate only the critical path of the workflow (i.e., the longest path when considering mean task durations). To obtain an estimated distribution for the duration of this path, we use the central limit theorem and approximate it with a normal distribution that has a mean λ_W equal to the sum of all mean task durations along the path and a variance v_W equal to the sum of the respective task variances. The corresponding probability density function for this duration is then $d_W(x) = (v_W 2\pi)^{-1/2} \, \mathrm{e}^{-(x-\lambda_W)^2/(2v_W)}$.

To solve the integral in Equation 3, we let $D_W(x)$ be the cumulative probability function[4] of $d_W(x)$, we let $D_d =$

[3] In practice, we approximate this by evaluating successively more finely grained time intervals to reach an estimate in $\lambda_i \pm 0.1$.

[4] $D_W(x) = \int_{-\infty}^{x} d_W(y) \,\mathrm{d}y$ is a common function that can be approximated numerically. In our work, we use the SSJ library (`http://www.iro.umontreal.ca/~simardr/ssj`).

$D_W(d)$ be the probability that the workflow will finish no later than the deadline d and $D_l = D_W(t_0) - D_W(d)$ the probability that the workflow will finish after the deadline but no later than time $t_0 = \frac{u_{\max}}{\delta} + d$ (both conditional on overall success), and we note the general form of $u(t)$ (Equation 1) to write:

$$\tilde{u} = p \cdot (D_d \cdot u_{\max} + D_l \cdot u(\lambda_W + (e^{\frac{-(d-\lambda_W)^2}{2v_W}} - e^{\frac{-(t_0-\lambda_W)^2}{2v_W}}) \frac{\sqrt{v_W}}{D_l \cdot \sqrt{2\pi}})) - \tilde{c} \quad (9)$$

This concludes our discussion of the *full flexible* strategy. In the following, we describe a second strategy, *fast flexible*, that includes some modifications to reduce the search space and convergence time of our provisioning approach.

Fast Flexible Strategy

A potential drawback of the above strategy is the fact that it explores a large state-space, which may take a long time to converge to a good solution (there could be thousands of service providers). To address this, it is possible to simplify the search space. Hence, rather than considering service providers individually, we associate three integer values with each possible service population P_k for a given task t_i:

- $n_{k,i} \in \{0, 1, 2, \ldots, |P_k|\}$: the number of providers to invoke in parallel (0 means none are invoked),
- $w_{k,i} \in \{1, 2, 3, \ldots, t_{\max}\}$: the number of time steps before invoking more providers from the same population,
- $b_{k,i} \in \{0, 1, 2, \ldots, t_{\max}\}$: the number of time steps before the first set of service providers is invoked.

For example, $n_{k,i} = 7$, $w_{k,i} = 15$, $b_{k,i} = 5$ means that the consumer should provision 7 service providers of population P_k, to be invoked 5 time steps after the relevant task was started, and that the consumer will invoke another 7 providers after 15 more time steps, and so on, until $t_{\max}$ or until no more providers are available. The special case $n_{k,i} = 0$ denotes the case where no providers of population P_k are provisioned for task t_i.

The hill-climbing procedure for this search space is similar to the *full flexible* stratgey. Again, we choose a random initial provisioning allocation, normalise $b_{k,i}$ for all populations of a given task, so that there is at least one P_k with $b_{k,i} = 0$, and then proceed to generate random neighbours by changing $n_{k,i}$, $w_{k,i}$ and $b_{k,i}$ of one randomly chosen population (again in both unit and randomly-sized steps).

To further reduce the running time of the algorithm, we also exploit its *anytime* property and stop its main loop (lines 4–19 in Algorithm 1) after g iterations. Because this may result in terminating the algorithm before it is was able to reach a good solution, we perform h random restarts and use the best solution[5]. Due to its aim of reducing the time of finding a good provisioning allocation, we refer to this strategy as the *fast flexible* strategy. In the following, we evaluate both our strategies and compare their respective merits.

[5]We use $g = 100$ and $h = 5$, because these values lead to good results in a variety of environments.

Empirical Evaluation

Because of the inherent intractability of our original problem and the heuristic nature of our strategies, we decided to conduct an empirical evaluation of the work presented herein. Our main aim is to investigate whether our strategies can achieve significant improvements over the currently prevalent approaches that do not consider service uncertainty, and to compare our two strategies to each other. In the following, we first describe our experimental setup and then report our results.

Experimental Setup

To test the performance of our strategies, we simulate a service-oriented system with randomly generated service providing agents and workflows. More specifically, we first generate five random *service types* that broadly characterise the services found in our system. We attach an average failure probability to each type by sampling from a beta distribution with mean θ and variance 0.01 (denoted $B(\theta, 0.01)$), where θ represents the overall average failure probability in the system and is the main variable we vary throughout our experiments. We also attach an average cost, duration shape and scale[6] to each service type, which are drawn from the continuous uniform distributions $U_c(1, 10)$, $U_c(1, 10)$ and $U_c(1, 5)$, respectively.

Next, we generate a random number of populations for each service type (drawn from the discrete uniform distribution $U_d(3, 10)$), and populate each with a random number of providers (drawn from $U_d(1, 100)$). Then, to introduce further heterogeneity, we vary the performance characteristics between the populations of each type, based on the type-specific values determined above. To this end, we generate the failure probability of a population by sampling from $B(x, 0.005)$, where x is the type-specific average failure probability. Furthermore, we determine the cost of each provider as the product $2 \cdot y \cdot z$, where y is the type-specific average cost and z is sampled from $B(0.5, 0.05)$. This is repeated for the duration parameters (resampling z for each).

Finally, workflows always consist of 10 tasks, with precedence constraints that are generated by randomly populating an adjacency matrix until at least 25% of all possible edges are added, ignoring any that would create a cyclic graph. We define $u(t)$ by setting $d = 100$, $u_{\max} = 1000$ and $\delta = 50$. The matching function $m(t_i)$ is created by mapping each task to the providers of a randomly chosen service type. This process ensures that our strategies are tested across a large spectrum of randomly generated environments, with considerable heterogeneity across service types and within the populations of a given type. Overall, this setup was chosen to represent a plausible workflow scenario, but we have experimented with other environments and observed the same trends as discussed in our results.

For each experimental run, we record the overall net profit our strategies achieve by provisioning and then executing a single workflow. To obtain statistical significance, we repeat all experiments 1000 times with new randomly generated

[6]These refer to the shape and scale of a discretised Gamma distribution, which is commonly used for service and queueing times.

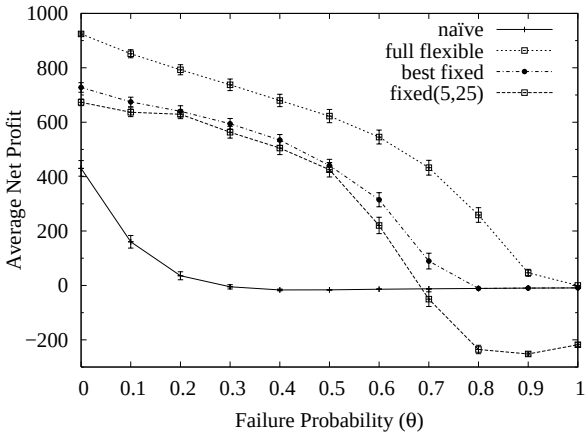

Figure 2: Performance of the *full flexible* strategy.

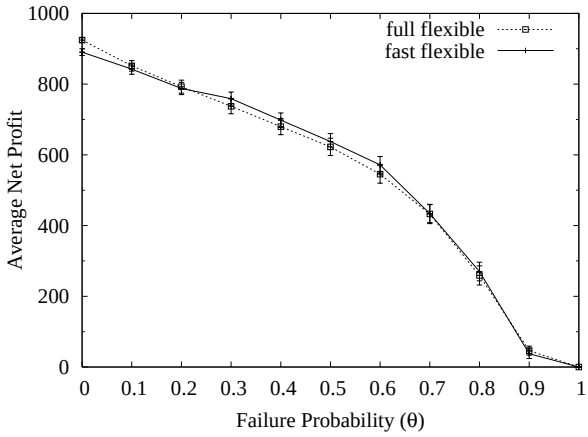

Figure 3: Performance comparison of *fast*/*full* strategies.

agents and workflows, and carry out appropriate statistical tests (we use two-sample t-tests and ANOVA at the 99% confidence level). To evaluate our strategies, we compare them to the following three benchmark strategies:

- **naïve:** The *naïve* strategy models current approaches that do not consider service uncertainty, and so it provisions only single providers for each task in the workflow (chosen randomly from all matching providers).
- **fixed(n,w):** This strategy provisions multiple providers for each task in the workflow, but does so in a fixed manner without explicitly considering the service parameters. Specifically, the *fixed(n,w)* strategy provisions sets of n random service providers in parallel, every w time-steps after a task becomes available.
- **best fixed:** This strategy represents an upper bound for the quality achieved by any *fixed(n,w)* strategy. We obtain this by exhaustively testing all feasible parameters for n and w in a given environment (for each θ value) and then retaining the result of the best performing strategy.

Full Flexible Profit Results

In our first set of experiments, we compared the *full flexible* strategy to our various benchmarks strategies. The results of this are shown in Figure 2 (all results are given with 95% confidence intervals). Here, the *naïve* strategy performs badly, clearly demonstrating the need for more flexible provisioning approaches. In fact, even when providers never fail (i.e., $\theta = 0.0$), it only achieves an average net profit of 430.3 ± 28.8 due to the uncertain and heterogeneous service durations that frequently cause it to miss its deadline. As the failure probability rises, performance degrades further, and the strategy begins to make a net loss when $\theta = 0.3$. After this, it stays constantly low as it rarely finishes any workflows successfully.

Figure 2 also shows *fixed(5,25)* as one of the overall best performing strategies with constant redundancy and waiting times. This strategy performs well in most environments tested and achieves significantly better results than the *naïve* strategy up to a failure probability of $\theta = 0.6$. However, beyond that, it starts to make considerable losses because it now fails to complete workflows successfully, but still commits a high level of investment to each task (5 providers are invoked in parallel every 25 time steps). For comparison, we also show *best fixed*, which achieves slightly better results in some cases and avoids making large losses when θ is high. Despite these promising results for the benchmark strategies, it should be noted that both were chosen retrospectively from a large set of strategies (the potential choices for n and w), most of which performed significantly worse. In reality, finding these at run-time would be non-trivial and therefore necessitates more flexible mechanisms.

Finally, the figure shows the results of the *full flexible* strategy. This follows a similar trend as the *best fixed*, but retains a positive net profit even when providers have an average failure probability of 0.9. More importantly, it significantly outperforms all other strategies at all failure probabilities. Unlike the other strategies, it also avoids making an average net loss in any environment (rather, it starts to ignore infeasible workflows, e.g., when $\theta = 1$). Averaged over all values for θ, *full flexible* achieves an average net profit of 535.66 ± 8.29, while *best fixed* and *fixed(5,25)* achieve 362.49 ± 7.74 and 263.4 ± 9.1, respectively. The *naïve* strategy only achieves 48.33 ± 4.24.

Fast/Full Comparison Results

Given these results of the *full flexible* strategy, we were interested in how it compares to the *fast flexible* strategy. Because of the simplified search space, we expected *fast flexible* to reach a solution faster, but possibly perform worse overall (as less solutions are considered).

To investigate this, we recorded the time taken by each strategy to reach a provisioning allocation during the experiments outlined in the previous section (these were executed on 2.2 GHz AMD Opterons with 1.98 GB RAM). Measured over all θ, the average time of *full flexible* is 37.82 ± 0.51s, the average time of *fast flexible* is only 4.82 ± 0.04s, thus reducing the run-time by over 85%. Similarly, the respective standard deviations are 27.29 ± 0.36s (72% of the average) and 1.88 ± 0.02s (39% of the average), indicating that the

time of *fast flexible* is also significantly less variable.

To compare the performance of both strategies, we recorded their average net profit in the same environments as discussed in the preceding section. The resulting data is shown in Figure 3, and it indicates that they are highly similar. In fact, when averaging over all failure probabilities, the average net profits are 536.0 ± 8.23 (*full flexible*) and 539.0 ± 8.03 (*fast flexible*). Hence, their overall performance is not significantly different. This trend continues when comparing the results individually for all values for $\theta > 0.0$. The only exception is at $\theta = 0.0$, when the *full flexible* slightly outperforms the *fast flexible* strategy. This is because the former is able to provision single providers initially, but can provision multiple providers at a later time if the single service takes unusually long. However, the difference is minor — at $\theta = 0.0$, the *full flexible* strategy achieves an average net profit of 924.5 ± 6.8 while the *fast flexible* achieves 890.4 ± 9.4.

Conclusions and Future Work

In this paper, we have extended the state of the art in service provisioning by focussing on environments where many heterogeneous and unreliable providers offer the same service types at different levels of quality and for varying costs. In particular, we have described a novel strategy (the *full flexible* strategy), which provisions multiple heterogeneous providers for the tasks of a workflow, and we have given practical modifications that provide faster solutions (the *fast flexible* strategy) with typically equally good results.

As service heterogeneity is a central feature of most large distributed systems, our work is pertinent for agent developers in a diverse range of areas. In particular, our strategies are equally applicable to the workflows of scientists in Grid environments, to the business processes of companies in e-commerce, and to agents that manage and broker resources in service-oriented systems.

Furthermore, the system model described in this paper is based loosely on current service-oriented frameworks, which are becoming popular for building distributed systems (such as Web or Grid services). Despite this, we have given an abstract model that does not depend on a particular implementation and so can be applied in a variety of settings and frameworks. In addition, we have employed commonly used stochastic measures to describe services, which can, in practice, be inferred through existing machine learning techniques or trust frameworks, and naturally allow some uncertainty about the performance characteristics of services. Hence, our work is applicable in realistic deployment scenarios, where complete performance information about providers is not necessarily available[7].

In future work, we will consider the adaptive provisioning of services as more information becomes available during workflow execution (e.g., to provision more services as the agent starts to fall behind schedule). Furthermore, we are interested in the use of advance negotiation mechanisms, where services are not invoked on demand, but rather in the context of pre-negotiated service level agreements.

[7] In related work, we showed that our approach achieves good results even if the available information is inaccurate (Stein, Payne, & Jennings 2007).

Acknowledgements

This work was funded by the Engineering and Physical Sciences Research Council (EPSRC) and a BAE Systems studentship.

References

Collins, J.; Bilot, C.; Gini, M.; and Mobasher, B. 2001. Decision processes in agent-based automated contracting. *IEEE Internet Computing* 5(2):61–72.

Hagstrom, J. N. 1988. Computational complexity of PERT problems. *Networks* 18:139–147.

Huhns, M. N., and Singh, M. P. 2005. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing* 9(1):75–81.

Jennings, N. R. 2001. An agent-based approach for building complex software systems. *Communications of the ACM* 44(4):35–41.

Malcolm, D. G.; Roseboom, J. H.; Clark, C. E.; and Fazar, W. 1959. Application of a technique for research and development program evaluation. *Operations Research* 7(5):646–669.

Michalewicz, Z., and Fogel, D. B. 2004. *How to solve it: Modern Heuristics*. Springer, 2nd edition.

Stein, S.; Jennings, N. R.; and Payne, T. R. 2006. Flexible provisioning of service workflows. In *Proc. 17th European Conference on Artificial Intelligence (ECAI06), Riva del Garda, Italy*, 295–299. IOS Press.

Stein, S.; Payne, T. R.; and Jennings, N. R. 2007. An effective strategy for the flexible provisioning of service workflows. In *Proc. Workshop on Service-Oriented Computing: Agents, Semantics, and Engineering (SOCASE 2007), Honolulu, Hawai'i, USA*, volume 4504 of *Lecture Notes in Computer Science*, 16–30. Springer.

Teacy, W. T. L.; Patel, J.; Jennings, N. R.; and Luck, M. 2006. TRAVOS: Trust and reputation in the context of inaccurate information sources. *Journal of Autonomous Agents and Multi-Agent Systems* 12(2):183–198.

Zeng, L.; Benatallah, B.; Dumas, M.; Kalagnanam, J.; and Sheng, Q. Z. 2003. Quality driven web services composition. In *Proc. 12th International World Wide Web Conference (WWW'03), Budapest, Hungary*, 411–421. ACM Press.