UNIVERSITY OF SOUTHAMPTON

Flexible Service Provisioning in Multi-Agent Systems

by

Sebastian Stein

A thesis submitted in partial fulfillment for the
degree of Doctor of Philosophy

in the
Faculty of Engineering, Science and Mathematics
School of Electronics and Computer Science
Intelligence, Agents, Multimedia Group

May 2008

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING, SCIENCE AND MATHEMATICS
SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

by Sebastian Stein

Service-oriented computing is an increasingly popular approach for providing applications, computational resources and business services over highly distributed and open systems (such as the Web, computational Grids and peer-to-peer systems). In this approach, service providers advertise their offerings by means of standardised computer-readable descriptions, which can then be used by software applications to discover and consume appropriate services without human intervention. However, despite active research in service infrastructures, and in service discovery and composition mechanisms, little work has recognised that services are offered by inherently autonomous and self-interested entities. This autonomy implies that providers may choose not to honour every service request, demand remuneration for their efforts, and, in general, exhibit uncertain behaviour. This uncertainty is especially problematic for the service consumers when services are part of complex workflows, as is common in many application domains, such as bioinformatics, large-scale data analysis and processing, and commercial supply-chain management.

In order to address this uncertainty, we propose a novel algorithm for provisioning services for complex workflows (i.e., for selecting suitable services for the constituent tasks of a workflow). This algorithm uses probabilistic performance information about providers to reason about service uncertainty and its impact on the overall workflow. Furthermore, our approach actively mitigates this uncertainty by employing two key techniques. First, it proactively provisions redundant services for particularly critical or failure-prone tasks (thus increasing the probability of success). Second, it recovers dynamically from service failures by re-provisioning services at run-time (without necessarily receiving explicit failure messages). Unlike existing work in this area, our algorithm employs principled decision-theoretic techniques to determine which services to provision, whether to introduce redundant services and when to re-provision failed services. In doing so, it explicitly balances the cost of provisioning with the expected value of the workflow.

To show how our algorithm applies to a range of common service-oriented systems, we consider a variety of different scenarios in this thesis. More specifically, we first examine environments where the consumer lacks specific knowledge to differentiate between distinct service providers,

as is common in highly dynamic and open systems. Despite this lack of detailed knowledge, we demonstrate how the consumer can use redundancy and dynamic re-provisioning to influence the outcome of a workflow and to deal with uncertainty. Then, we look into systems where the consumer has more specific knowledge about highly heterogeneous providers. While existing work has concentrated on selecting the single best provider for each workflow task, we show that a consumer can often improve its performance by provisioning multiple providers with different qualities for a single task. Finally, we discuss how our algorithm can be adapted for systems where consumers and providers reach explicit service contracts in advance. In this context, we are the first to propose a gradual provisioning approach, whereby the consumer negotiates contracts for some tasks in advance, but leaves the negotiation of others to a later time. This approach allows the consumer to better react to uncertain service outcomes and to avoid paying reservation fees that are later lost when services fail.

Throughout this thesis, we compare our approach empirically to current provisioning algorithms. In doing so, we demonstrate that our approach typically achieves a significantly higher utility for the service consumer than approaches that do not reason about uncertainty, that rely on fixed levels of redundancy or service time-outs, and approaches that select single services to achieve the optimal balance of various performance characteristics. Furthermore, we show that these results hold over a large range of environments and workflow types and that our algorithm copes well even in highly uncertain environments where most services fail. As our approach relies on fast heuristics to solve a problem that is known to be intractable, it scales well to larger workflows with hundreds of tasks and thousands of providers. Finally, where it is tractable to compute an optimal solution, we show empirically that our algorithm achieves a high utility that is within 87% or more of the optimal.

Contents

List of Figures

List of Tables

List of Algorithms

Declaration of Authorship

I, Sebastian Stein, declare that the thesis entitled *Flexible Service Provisioning in Multi-Agent Systems* and the work presented in this thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published in a number of conference and journal papers (see Section 1.5 for a list).

Signed:

Date:

Acknowledgements

When I first met my supervisors, Nick Jennings and Terry Payne, they asked me whether I had any idea what pursuing a PhD degree would entail. At the time, I had a vision of spending months immersed in books, reading scientific papers and studying equations, to finally emerge with a groundbreaking idea that would change the field of computer science forever.

Clearly, I had no clue of what lay ahead of me. Working on a PhD thesis is rarely a straightforward process, nor does it lead to a single solution to all problems in its field. Rather, it involves many long hours of hard work, incremental improvements and numerous dead ends (which are all offset by the occasional breakthrough).

Therefore, I am all the more grateful for Nick and Terry's invaluable support and guidance throughout this process. Initially, they left me plenty of freedom to explore a wide subject area, but also pulled my feet onto the ground when it became necessary to set realistic and manageable research goals. Whenever I reached a dead end, they offered their advice and nudged me in the right direction. At the same time, they looked critically at any half-baked ideas, questioned every unusual pattern in my results and thereby helped me on my path towards becoming a researcher. Finally, Nick and Terry invested a tremendous amount of their time, both during our weekly meetings and while correcting my papers (often over weekends and holidays). In short, this thesis would not have been possible without their support and encouragement.

I would also like to thank all the members of the IAM group who have offered their feedback on my work and helped improve it. While there are too many to list here, I am particularly grateful for the comments and suggestions of Enrico Gerding, who has discussed with me many aspects of my work, and David Yuen, who has patiently helped me with even the most trivial mathematics problems. Furthermore, the weekly research seminars on agent-based computing, organised first by Raj Dash and then Alessandro Farinelli, have also been an important influence on my work, both by introducing me to pertinent research issues and by allowing me to present my own research to a supportive audience.

Additionally, I am grateful for feedback from the wider research community, and in particular for the constructive comments I have received from the reviewers of the ACM TOIT journal, and from the reviewers and participants at the ECAI-06, AAMAS-07, AAAI-07 and AAMAS-08 conferences, and the EUMAS-06 and SOCASE-07 workshops. I have also benefited greatly

by attending the SSSW-05 and EASSS-05 summer schools and the AAMAS-07 and AAAI-07 doctoral mentoring programmes.

This thesis has been supported financially by the EPSRC and by BAE Systems, and I would like to express my sincere gratitude to both organisations. As a representative of the latter, Andy Wright in particular, has shown interest in my research and he gave me interesting pointers for my early work.

I am also thankful for the support of my friends. They provided welcome diversions when I needed to get away from work, shared my passion for board games, joined me in exploring Southampton and its surroundings on weekends and introduced me to new cultures and countries that I would not have experienced otherwise.

Last but not least, I have received enormous support from my family. My parents, Axel and Brigitte, and my sister, Philine, have always given me love and encouragement. During my PhD, they provided me with a second home in Germany, called me frequently and came to Britain for regular visits (patiently putting up with my increasingly busy schedule). My thanks also go to my girlfriend, Shanna, who has shared with me many happy moments, given me strength and faced with me a particularly challenging time in the last months of my studies.

Nomenclature

General

$\mathbb{N}$	Natural numbers with zero ($\mathbb{N} = \{0, 1, 2, \ldots\}$)
$\mathbb{R}$	Real numbers
$\mathbb{R}^+$	Positive, non-zero real numbers
$\mathbb{R}_0^+$	Positive real numbers with zero
$\mathbb{Z}$	Integer numbers ($\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$)
$\mathbb{Z}^+$	Positive, non-zero integer numbers ($\mathbb{Z}^+ = \{1, 2, \ldots\}$)
$\mathcal{P}(A)$	Powerset of A ($\mathcal{P}(A) = \{a \mid a \subseteq A\}$)
$\mathcal{U}_c(a, b)$	Continuous uniform distribution over the interval $[a, b]$ (including a and b)
$\mathcal{U}_d(a, b)$	Discrete uniform distribution over the integers in the interval $[a, b]$ (including a and b)

Chapter 3

$D : (\mathcal{S} \times \mathbb{R}) \to \mathbb{R}$	Cumulative density function of the duration of a service
$E : \mathcal{P}(T \times T)$	Task precedence constraints of a workflow
$\mathcal{S} = \left\{s_1, s_2, \ldots, s_{\lvert\mathcal{S}\rvert}\right\}$	Set of service instances
$S_i = \mu(\tau(t_i))$	Set of services that can be invoked for a given task t_i
$T = \left\{t_1, t_2, \ldots, t_{\lvert T\rvert}\right\}$	Tasks of a workflow
$\mathcal{T} = \left\{T_1, T_2, \ldots, T_{\lvert\mathcal{T}\rvert}\right\}$	Set of all service types
T_i	Service type
$W = (T, E, \tau, u)$	Workflow
Λ	Consumer's high-level objective
$c : \mathcal{S} \to \mathbb{R}$	Maps service instances to their costs
$d : (\mathcal{S} \times \mathbb{R}) \to \mathbb{R}$	Probability density function of the duration of a service
$f : \mathcal{S} \to [0, 1]$	Maps service instances to their failure probabilities
s_i	Service instance
$\hat{t} : \mathbb{N}$	Current time step

t_i	Task
$t_{\text{max}} : \mathbb{R}_0^+$	Deadline of a workflow (for receiving u_{max})
$t_{\text{zero}} : \mathbb{R}_0^+$	Earliest time step for which $u(t_{\text{zero}}) = 0$
$u : \mathbb{R} \to \mathbb{R}$	Workflow utility function
$u_{\text{max}} : \mathbb{R}_0^+$	Maximum utility of a workflow
$\delta : \mathbb{R}^+$	Cumulative penalty for late completion of a workflow
$\mu : \mathcal{T} \to \mathcal{P}(\mathcal{S})$	Maps service types to sets of matching services
$\tau : T \to \mathcal{T}$	Maps tasks to associated service types

Chapter 4

C_i	Random variable denoting the duration of a single task
$\hat{D}_i : \mathbb{R} \to [0, 1]$	Non-conditional probability that a single invocation for task t_i is successful by a given time
$D_{\text{late}} : [0, 1]$	Estimated probability that workflow will complete after t_{max} and by t_{last}
$D_{\text{max}} : [0, 1]$	Estimated probability that workflow will complete by t_{max}
$\mathcal{E} : \mathbb{N}$	Number of direct, non-transitive edges in E
$\tilde{c} : \mathbb{R}$	Estimated expected cost of a provisioned workflow
$\bar{c}(\vec{n}, \vec{w}) : \mathbb{R}$	Expected cost of following $\vec{n}$ and $\vec{w}$
$c_i : \mathbb{R}$	Cost of a service instance from S_i
$\bar{c}_i : \mathbb{R}$	Expected cost of a provisioned task t_i
$d_i : \mathbb{R} \to \mathbb{R}$	Probability density function of a service instance from S_i
$\bar{d}_k : \mathbb{R}$	Expected completion time of a task if it is successfully completed during the kth invocation
$d_W : \mathbb{R} \to \mathbb{R}$	Estimated probability density function of workflow duration
$f_i : [0, 1]$	Failure probability of a service instance from S_i
$\hat{f}_i : [0, 1]$	Failure probability of a single invocation for task t_i ($\hat{f}_i = 1 - \hat{p}_i$)
$\vec{n} : \mathbb{N}^{\lvert T \rvert}$	Vector containing number of parallel services for each task
$n_i : \mathbb{N}$	Number of parallel services provisioned for task t_i
$p : [0, 1]$	Success probability of workflow
$p_i : [0, 1]$	Success probability of a provisioned task t_i
$\hat{p}_i : [0, 1]$	Success probability of a single invocation for task t_i
$\tilde{u} : \mathbb{R}$	Estimated expected utility of a provisioned workflow
$\bar{u}(\vec{n}, \vec{w}) : \mathbb{R}$	Expected utility when following $\vec{n}$ and $\vec{w}$

$\bar{u}_t(\vec{n}, \vec{w}) : \mathbb{R}$	Expected reward when following $\vec{n}$ and $\vec{w}$
$\tilde{r} : \mathbb{R}$	Estimated expected reward of a provisioned workflow
$\tilde{r}_s : \mathbb{R}$	Estimated reward of workflow completion (conditional on success)
$r_i : [0, 1]$	Probability that task t_i is reached during workflow execution
$t_{\text{last}} : \mathbb{R}$	Smallest real parameter x, where $u(x) = 0$
$\bar{t}_i : \mathbb{R}$	Expected completion time of a provisioned task t_i (conditional on success)
$\bar{t}_{\text{late}}$	Estimated expected completion time given that workflow is completed between t_{max} and t_{last}
$v_i = \lvert S_i \rvert$	Number of service instance available for a given task
$v_W : \mathbb{R}$	Estimated variance of workflow duration
$w_i : \mathbb{N}$	Waiting time until re-provisioning task t_i
$\vec{w} : \mathbb{N}^{\lvert T \rvert}$	Vector containing waiting times for each task
$\lambda_W : \mathbb{R}$	Estimated expected workflow duration
$\mu_i : \mathbb{R}$	Expected completion time of a single invocation for task t_i (conditional on success)
$\bar{\sigma}_i^2 : \mathbb{R}$	Completion time variance of a provisioned task t_i (conditional on success)

Chapter 5

$A = T \to (\mathcal{S} \to \mathbb{N})$	Set of all provisioning allocations
$\hat{D} : (\mathcal{S} \times \mathbb{N}) \to [0, 1]$	Probability that a given service has completed successfully by a certain time (not conditional on overall success)
$E_i : (A \times \mathbb{N}) \to [0, 1]$	Probability that task t_i has been completed successfully after a given amount of time and a given provisioning allocation
$P = \{P_1, P_2, \ldots, P_{\lvert P \rvert}\}$	Set of populations (partition of $\mathcal{S}$)
P_i	A service population
$Q_i : \mathcal{S} \to [0, 1]$	As q_i, but each value is normalised to the interval $[0, 1]$
$\hat{Q}_i : \mathcal{S} \to [0, 1]$	As $\hat{q}_i$, but each value is normalised to the interval $[0, 1]$
$\Phi : [0, 1]$	Average failure probability in a simulated service-oriented system
$b_{k,i} : \{0, 1, \ldots, t_{\text{zero}}\}$	Number of time steps to wait before provisioning the first $n_{k,i}$ services for t_i
$\bar{c}(\alpha) : \mathbb{R}$	Expected cost of following allocation α

$n_{k,i} : \{0, 1, \ldots, \lvert P_k \rvert\}$	Number of services from population P_k that are provisioned in parallel for task t_i
$q_i : \mathcal{S} \to \mathbb{R}$	Maps services to their ith quality parameter
$\hat{q}_i : \mathcal{S} \to \mathbb{R}$	Maps vectors of services to the ith quality parameter, aggregated over a workflow
$s^* : \mathcal{S}$	Service selected by a *local* strategy for a given task
$\bar{u}_t(\alpha) : \mathbb{R}$	Expected reward of following allocation α
$\check{w}_i : [0, 1]$	Weight factor used by QoS-based strategies
$w_{k,i} : \{1, 2, \ldots, t_{\text{zero}}\}$	Number of time steps to wait before re-provisioning $n_{k,i}$ more services from population P_k for task t_i
$\alpha : A$	Detailed provisioning allocation that maps each task to its provisioned services and their relative invocation times
$\beta : ((\mathbb{Z} \times \mathbb{Z}) \to \mathbb{Z})^3$	Fast provisioning allocation containing $n_{k,i}$, $w_{k,i}$ and $b_{k,i}$ for each task and population
$\rho^* : \mathcal{S}^{\lvert T \rvert}$	Vector of services selected by a *global* strategy

Chapter 6

$\mathbb{C}$	Set of all offers
$\mathbb{O} = \{\texttt{succ.}, \texttt{unavail.}, \texttt{fail.}\}$	Set of strategy outcomes
$C_\varphi \subseteq \mathbb{C}$	Response of the market to φ
$C_{\hat{t}} \subseteq \mathbb{C}$	All offers received during time step $\hat{t}$
$E' : \mathcal{P}(T \times T)$	Temporary workflow edges
$\mathcal{E}_i : \mathbb{R} \to \mathbb{R}$	Cumulative probability that the predecessors of t_i have finished
$P_d : \mathbb{C} \to [0, 1]$	Offer defection probabilities
$P_f : \mathbb{C} \to [0, 1]$	Offer failure probabilities
$B_i : \mathcal{P}(\mathbb{N})$	Indices of tasks that precede t_i
$P_s : \mathbb{C} \to [0, 1]$	Offer success probabilities
$T_\alpha \subseteq T$	Completed tasks
$T_\beta \subseteq T$	Uncompleted tasks with offers
$T_\gamma \subseteq T$	Provisioned tasks without offers
$\Psi = (T_\alpha, T_\beta, T_\gamma, d_\beta, d_\gamma, E')$	A provisioning strategy
Ω	Set of all high-level strategies
$a : C_{\hat{t}} \to T$	Acknowledgement, maps offers to workflow tasks
$\check{c} : \Omega \to \mathbb{R}$	Strategy costs

$c_e : \mathbb{C} \to \mathbb{R}$	Offer execution costs
$\bar{c}_e : \mathbb{R}$	The expected execution cost of a provisioning decision
$\check{c}_e : \Omega \to \mathbb{R}$	Strategy execution costs
$c_{ei} : \mathbb{R}$	Expected execution cost of t_i
$c_{\text{inv}} : \mathbb{C} \to \mathbb{R}$	Offer expected invocation costs
$c_r : \mathbb{C} \to \mathbb{R}$	Offer reservation costs
$\bar{c}_r : \mathbb{R}$	The expected reservation cost of a provisioning decision
$\check{c}_r : \Omega \to \mathbb{R}$	Strategy reservation costs
$c_{ri} : \mathbb{R}$	Expected reservation cost of t_i
$\bar{c}_t : \mathbb{R}$	The expected cost of a provisioning decision
$d : \mathbb{C} \to \mathbb{Z}^+$	Offer durations
$\check{d} : (\Omega \times \mathbb{O}) \to \mathbb{R}$	Strategy outcome times
$\check{d}^2 : (\Omega \times \mathbb{O}) \to \mathbb{R}$	Strategy squared times
d_β	Maps tasks to provisioned offers and high-level contingency plans
$d_f : \mathbb{R}$	Expected time at which failure becomes known
d_γ	Maps tasks to high-level strategies and maximum late probabilities
$d_{i,\text{end}} : \mathbb{R}$	Predicted completion time of t_i
$d_{i,\text{pre}} : \mathbb{R}$	Largest predicted completion time of any predecessor of t_i
$d_l : \mathbb{R}$	Expected time at which late outcome becomes known
$d_s : \mathbb{R}$	Expected time at which successful outcome becomes known
$l : \mathcal{T} \to \mathcal{P}(\Omega)$	Strategy library (maps service types to sets of strategies)
$o_{\text{after}} : \mathbb{Z} \to \mathcal{P}(\mathbb{C})$	Maps a time slots to all offers in a provisioning decision that start no earlier than it
$o_{\text{pre}} : (\mathbb{Z} \times \mathbb{Z}) \to \mathcal{P}(\mathbb{C})$	Maps a time interval to the offers in a provisioning decision that are completely enclosed by it
$n : \Omega \to \mathbb{Z}^+$	Strategy maximum offer numbers
$o : \mathbb{C}$	An offer
$\check{p} : (\Omega \times \mathbb{O}) \to [0, 1]$	Strategy outcome probabilities
$p_f : [0, 1]$	Probability that some offers in a provisioning decision are invoked, but all fail
$p_{\text{inv}} : (\mathbb{C} \times \mathbb{Z}) \to \mathbb{R}$	Offer invocation probability, given a task starting time
$p_l : [0, 1]$	Probability that the predecessors of a provisioned task complete late
$\hat{p}_l : [0, 1]$	Actual late probability
$p_{ml} : [0, 1)$	Maximum late probability of a task
$p_s : [0, 1]$	Probability that a provisioning decision completes successfully

$\hat{p}_s : \mathbb{Z} \to \mathbb{R}$	Probabilities of starting at the time given by $\hat{s}$
$s : \mathbb{C} \to \mathcal{T}$	Offer service types
$\hat{s} : \mathbb{Z} \to \mathbb{Z}$	Ascending sequence of unique starting times in a provisioning decision
$s_f : \Omega$	Provisioning strategy if s_p failed
$s_l : \Omega$	Provisioning strategy if task is late
$s_p : \Omega$	Main provisioning strategy
$s_u : \Omega$	Provisioning strategy if s_p did not result in provisioned offers
$t : \mathbb{C} \to \mathbb{N}$	Offer starting times
$t_a : \Omega \to \mathbb{N}$	Strategy advance times steps
$t_{\text{end}} : \mathcal{P}(\mathbb{C}) \to \mathbb{R}$	Maps set of offers to offer with largest end time
$\bar{t}_{s,i} : \mathbb{R}$	Expected squared duration of t_i
$t_w : \Omega \to \mathbb{Z}^+$	Strategy time intervals
$\tilde{v} : (\Omega \times \mathbb{O}) \to \mathbb{R}$	Strategy time variances
$v_{i,\text{end}} : \mathbb{R}$	Predicted completion time variance of t_i
$v_{i,\text{pre}} : \mathbb{R}$	Variance associated with $d_{i,\text{pre}}$
$v_l : \mathbb{R}$	Variance of time at which late outcome becomes known
$v_f : \mathbb{R}$	Variance of time at which failure becomes known
$v_s : \mathbb{R}$	Variance of time at which successful outcome becomes known
$\hat{w}_i : \mathbb{R}$	Waiting time when task is provisioned in advance
$\delta_f : \mathbb{C} \to \mathbb{R}$	Offer failure penalties
$\gamma_i \subseteq \mathbb{C}$	Provisioning decision for task t_i
$\epsilon \in \mathbb{O}$	An outcome of a high-level strategy
$\vartheta : \Omega \to$ $\{\texttt{cost}, \texttt{unrel.}, \texttt{end.}, \texttt{balanced}\}$	Strategy selection techniques
$\varphi : \mathcal{T} \times \mathbb{N}$	Call for proposals

To Axel, Brigitte, Philine and Shanna
for their love, support and encouragement.

Chapter 1

Introduction

The digital computer has been one of the most important inventions of the 20th century. Since its inception in the 1940s, it has had a profound impact on the development of contemporary society, supporting the large-scale automation of business activities, driving scientific progress and controlling the tools and appliances that we rely on in our daily lives. During its history, the computer has evolved quickly, from being an isolated and independent calculating device, to one that is now often part of complex distributed networks that span the globe.

This ubiquitous connectivity is revolutionising the usage of the computer, allowing users unprecedented access to a vast range of resources and services — including information repositories, remote computing facilities and even traditional business services that can be procured on the World Wide Web. In this context, there is a growing interest in building software applications that automatically discover and engage these resources, e.g., to execute complex business processes that rely on services by external suppliers, or scientific workflows that use data processing services hosted on remote mainframes.

However, building such software applications is posing new challenges to the research community, as traditional software engineering approaches are often inadequate in addressing the heterogeneity, dynamism and openness inherent in large distributed systems. Important issues that are already being addressed include appropriate methods for the automated discovery of distributed resources using computer-readable description languages, composition techniques that automatically combine several services into larger workflows, and standardised frameworks that allow heterogeneous applications to communicate and exchange data.

Furthermore, an important challenge is the need to deal with fundamentally *autonomous* software components. As applications begin to outsource functionality across organisational boundaries, they also start to rely on different stakeholders that have their own goals and agendas. Contrary to traditional software models, distributed components no longer obey every instruction, nor act in a deterministic manner. Instead, they follow their own decision-making procedures, which are opaque to the consumers and primarily represent the interests of their owners. As

such, these components may fail without warning or respond later than anticipated, thus posing significant risks to consumers that rely on them for important, perhaps business-critical, services.

This critical, but so far largely unexplored, feature of large-scale distributed systems is the principal focus of this thesis. In particular, we investigate how a software application can select, provision and monitor the services of external providers in a flexible manner to reduce service failures, meet workflow constraints and react to problems.

We begin in this chapter by outlining the background to our work and by setting a research agenda. More specifically, in Section 1.1, we discuss current trends in distributed computing. Then, in Section 1.2, we introduce service-oriented computing, a popular approach for building applications in distributed systems. Following this, Section 1.3 outlines the field of multi-agent systems, which we believe is central to achieving flexible service-oriented systems. From the preceding discussion, we then motivate our research and outline the requirements of our work in Section 1.4. This is followed by an overview of our research contributions (Section 1.5) and an outline of the remainder of this thesis (Section 1.6).

1.1 Current Trends in Distributed Computing

The proliferation of large-scale computer networks, such as the Internet, has made it possible for computer systems across the world to communicate and exchange data (Wittie (1991)). Both in offices and at home, this new medium has enabled human users to communicate instantly via electronic messages (Morris and Ogan (1996)) and to access an unprecedented amount of information via the World Wide Web (Berners-Lee et al. (1994)).

Given this widespread connectivity, there is growing interest in building computer applications that interact automatically with each other over networks, in order to share resources and data without human intervention. On one hand, such automation promises increased efficiency and economic rewards as applications are able to procure goods and services instantly, according to their current requirements and the market environment. On the other hand, automation simply becomes a necessity as systems are increasingly complex, with potentially thousands of heterogeneous and constantly changing components. Already, some successful applications exist, where data and functionality are distributed over large distances and across organisational boundaries (Timmers (1999); Anderson (2004)).

Such large-scale distributed systems can offer advantages to a wide range of users. In industry, companies are now interested in automating their business processes: for example to make supply-chains more agile and interoperable across different organisations (Johnson and Whang (2002)), to automate trading between businesses (He et al. (2003)) or to sell processing time and specialised services, such as video rendering (Byde (2006)). In fact, in 2005, 7.6 % of all UK businesses with 10 or more employees already used software to automatically interact

with and order from suppliers, and for businesses with 1000 or more employees, that figure was 42.1 % (Avery et al. (2007)). In a similar vein, researchers in academia expect to benefit from sharing experimental results and expensive hardware (Butler (1999)), an idea epitomised by the Grid (Foster et al. (2001)). Even home users already commonly participate in large-scale peer-to-peer networks to share idle processing time (Anderson et al. (2002)) or data (Matei et al. (2002)).

However, applications in large distributed systems are fundamentally different from traditional, monolithic software. Rather than being self-contained, deterministic programs, these applications access software components that are written and maintained by external organisations. Furthermore, those components reside on remote machines, may not be well documented, contain bugs and are subject to change at any time.

In particular, this means that the interacting software components in these distributed systems are typically highly *heterogeneous*. That is, they are implemented by different programmers, written in a variety of languages and execute on many distinct platforms. Hence, they often display different performance characteristics, such as reliability, response time and cost. Furthermore, there is considerable *uncertainty* in the behaviour of components, as these are usually opaque and outside the consumer's direct control. For example, the computer systems of a service provider may break down without warning due to a local power failure or high demand by many concurrent consumers. Finally, many large distributed systems are *open* in nature, as is the case with the Internet, where new entities are free to join at any time. This means that the level of demand for software components can change as more users join, but also that new and better offerings may become available over time. Similarly, such openness often implies that entities may also leave at any time, which requires software to adapt quickly and make alternative provisions for critical components.

Clearly, the above characteristics of distributed systems demand a flexible software engineering approach, that is able to discover and engage heterogeneous and previously unseen components at run-time. One prominent engineering paradigm with this aim is service-oriented computing, which we discuss in the following section, and which forms the primary application area of this thesis.

1.2 Service-Oriented Computing

Service-oriented computing has been suggested as an appropriate paradigm for systems where many heterogeneous software components interact. In this approach, resources, software functions and other behaviours are offered by their providers as computer *services* (Huhns and Singh (2005)). These services encapsulate key functionalities that consumers can procure in order to fulfil their own aims and objectives.

An important feature of service-oriented computing is the dynamic selection process between services and their consumers. Rather than being explicitly specified a priori by a programmer, services that achieve a given task are discovered by an application at run-time. For this reason, providers use public registries to publish the necessary interfaces and descriptions of their services, which are then interpreted and engaged automatically by the service consuming application. Such dynamic binding offers some resilience against the dynamism of a distributed system as an application does not need to rely on the availability of a single provider of a particular service. Rather, it can choose the most appropriate service provider that is available when needed.

In most realistic application scenarios, including those mentioned in the preceding section, service consumers will need to execute complex *workflows*, composed of many services (Deelman et al. (2003b)). Here, each service contributes some atomic unit of functionality to the overall goal of the consumer. Often, the output of one service may be passed directly to the next service, creating interdependencies between the constituent services of a workflow. Such service compositions may also contain conditional branches, loops or other patterns that are commonly found in generic workflow languages (van der Aalst et al. (2003)).

So far, service-oriented computing has largely focussed on basic protocols and standardised data formats that enable applications to discover and communicate with each other. As such, it is a vital enabling technology for distributed systems, but there has been little work on exploring the inherent uncertainty of services in these systems and the fundamentally autonomous nature of service providing agents. However, when relying on external providers for vital services, appropriate mechanisms and strategies are needed to deal with the associated risk and uncertainty. In the following section, we introduce the field of multi-agent systems, because we believe that its methods and models are critical to understanding and addressing these challenges.

1.3 Multi-Agent Systems

When many heterogeneous and independent entities (i.e., the service consumers and providers) interact in an open system, it is vital to recognise that these often represent distinct stakeholders with different, perhaps conflicting, aims. For example, these entities could include several service consuming applications, executed by different research laboratories, that all require the same highly specialised service for their workflows. Likewise, several companies might sell the same type of service and compete with each other for customers.

A fitting metaphor for such entities is the notion of *agents*. These are self-interested entities that act autonomously in order to achieve their own goals (Jennings (2000); Wooldridge (2002)). Researchers in the field of multi-agent systems have developed powerful models of how such agents interact, and how computational agents and distributed systems can be engineered to display desirable properties despite their fundamentally self-interested nature (Weiß (1999); Wooldridge and Ciancarini (2000); Dash et al. (2003)).

A central concept of agent-based research is the rationality of individual agents. In order to meet their goals and objectives (or those of their owners), agents normally seek to maximise their private utility, a measure of their personal welfare. Acting in such a way allows the agent to make appropriate decisions that balance the risks and potential benefits of its actions. It also implies that a rational agent would not generally offer services for free when there is an associated cost to itself, and that it may act to the detriment of other agents when this increases its own utility. As an example, the provider of a scientific supercomputer may offer processing time to other agents on a Grid, but withdraw these without warning when the computer is needed by members of its own department.

As a result, the behaviour of an autonomous agent is inherently uncertain for external observers, including the consumers of its services. Such uncertainty could be manifested by the failure of a provider to deliver its service (for example, because it can offer the service to a better customer, because the service is no longer profitable or simply because it suffered a system crash). Even when a service is delivered, there will still be uncertainty about when it is completed and about the quality of the result, as the provider may try to minimise costs to itself, serve several customers at the same time and possibly rely on third parties for parts of its service.

Furthermore, it is important to realise that when self-interested agents interact, they do so generally on a mutually beneficial basis, i.e., agents only interact when this increases their own utility. Hence, it is usually necessary to place these agents into an appropriate economic context, where they exchange services for other resources. To this end, expressive mechanisms, such as negotiation protocols or auctions, have been developed to allow agents to reach mutually beneficial agreements about the provision of services, usually in exchange for financial remuneration (Sandholm (1999); Jennings et al. (2001)). These mechanisms might include advance provisioning and negotiation over various parameters of a service, including its cost, deadline and quality parameters.

Viewing service providers and consumers as self-interested agents that interact through market mechanisms is highly appropriate for the type of large distributed systems we consider here. As these agents belong to distinct companies or organisations, they would normally have a considerable interest in making rational decisions that maximise their own utility and do not lead to situations that are detrimental to themselves. This is highlighted especially by the current interest by companies in automating their business processes and offering specialised services to paying customers, in order to gain some economic benefit (as discussed in Section 1.1).

Despite this, the field of service-oriented computing has often failed to view service consumers and providers as fully autonomous and self-interested agents. Rather, they have been treated as loosely coupled, but mostly cooperative entities that honour service requests without question. This is unrealistic, because such an approach neglects the inherent uncertainty of autonomous agents and fails to acknowledge the need for providers and consumers to reach mutually beneficial agreements over the provision of services.

Due to these considerations, designing computational agents that rely on external services for their workflows remains a critical challenge for service-oriented computing. Because of the unreliability and potential cost of procuring services, such agents must make rational decisions at run-time according to the interests of their owners. This means that an agent needs to react to failures in a timely manner to meet its deadlines and that it should minimise costs, but spend extra resources when appropriate, for example to ensure the success of a particularly critical part of a workflow. To this end, in the following section we outline the research requirements that we aim to address in order to build such agents.

1.4 Research Requirements

In this thesis, we are interested in *designing principled tools and methods for building a computational agent that is capable of executing complex workflows in highly dynamic and uncertain service-oriented environments*. In particular, we aim to use appropriate techniques and methodologies from the field of multi-agent systems to extend the currently prevalent perception of service-oriented computing. In carrying out this work, we devise techniques applicable to realistic applications that will be common in large distributed systems and that will depend heavily on remote services. These applications might include scientific workflows executed on a Grid infrastructure, business workflows that acquire goods and services from external providers, or workflows in peer-to-peer systems, where the consumer relies on a fast-changing population of providers (for example in an ad hoc network of wireless devices (Corson et al. (1999))). As such, our work focusses on efficient and scalable techniques that provide fast results in a dynamic setting. This means that we are interested in building boundedly rational agents that achieve good, "satisficing" results, where it would be impractical to achieve optimality (Simon (1957)).

To frame the thesis, we begin by outlining a set of requirements that detail the types of problems and system features that we expect our methods to deal with. We present these as *model* requirements, which pertain to general features of the systems that we investigate (drawn from the discussion above); *workflow* requirements, which describe the types of problems that we expect to cover; and *agent* requirements, which outline properties of the techniques that we will develop.

1.4.1 Model Requirements

This section contains requirements for an appropriate model of a distributed service-oriented system. These requirements deal mostly with the inherent autonomy of service providers that we intend to address.

M.1. **Uncertain Service Behaviour**

As discussed in Section 1.3, service outcomes are generally uncertain. At the very least, the model must assume uncertainty along the following dimensions:

a. *Service Success*
   Service success cannot be guaranteed. Even when a provider has agreed to offer a service or has already started execution, there is still a possibility that the provider fails to honour the service request.

b. *Service Duration*
   Services do not normally take a fixed amount of time. Rather, this time varies due to uncertainty in the task itself, the network traffic and the current workload of the provider.

M.2. **Remuneration for Service Provision**
The model must not assume that services are provided free of charge. Usually, some form of financial remuneration should be given to providers for their services. In particular, the model should explore the impact of the following common pricing models:

a. *Fixed Pricing*
   Providers charge for services based on a fixed, public price that is known by all consumers.

b. *Flexible Pricing*
   Providers produce individual quotes for each service request, which may change between requests and may be valid for short time periods only.

M.3. **Service Interaction Models**
While most current frameworks consider on demand invocation as the main mechanism for engaging services, more expressive interaction models have been suggested and should be considered by the model. These become especially relevant when consumers procure expensive, complex services that have to be provisioned ahead of time. Hence, an appropriate model should explore the following mechanisms:

a. *On Demand Invocation*
   Services are only procured when they are needed. This offers both consumers and providers high flexibility, but could prove to be too unreliable when a consumer needs some assurance that a given service is available at a certain time.

b. *Advance Provisioning*
   Agents might negotiate in advance over the provision of a service. Such an approach would provide the consumer with some assurance that a provider intends to fulfil a service at a negotiated future time (although not necessarily a guarantee).

M.4. **Provider Heterogeneity**
As services are usually offered by distinct agents with varying resources and different interests, their characteristics can vary considerably. Hence, the same type of service might be offered by several agents at a different price, response time and with a different level of reliability.

M.5. **Dynamism**
Over time, the characteristics of service providers and the system as a whole are likely to change due to varying levels of demand or availability and the changing interests of individual agents.

1.4.2 Workflow Requirements

In this section, we identify the requirements for the types of workflows that service consuming agents may face in the environments that we consider.

W.1. **Workflow Expressivity**
To cover common task compositions, a workflow model must include some basic workflow patterns, including:

a. *Parallel Task Ordering*
Workflows may contain services that can be executed in parallel, for example when they are completely independent.

b. *Sequential Task Ordering*
Workflows may also contain dependencies between services, for example where the output of one service provides the input for another.

W.2. **Use of Appropriate Reward Model**
An appropriate reward model should be present to express the value of a workflow. This should take into consideration not just whether a workflow has been successfully executed or not, but also the timeliness of this event.

1.4.3 Agent Requirements

This section contains an overview of the requirements that a successful service consuming agent must meet in a distributed environment.

A.1. **Principled Decision Framework**
The techniques developed in this thesis must allow an agent to make autonomous decisions with little or no human intervention. To this end, they should be based on a principled framework (such as probability and decision theory), in order to yield a generic and widely applicable model that deals effectively with a range of scenarios. In the context of such a framework, we will strive to maximise some objective performance criteria, but not necessarily obtain optimality where this would be impractical.

A.2. **Failure Handling**
The agent must be able to handle service failures in an appropriate manner. This should include:

a. *Reactive Failure Handling*
When failures occur, it is vital to respond accordingly and minimise the disruption to a workflow.

b. *Proactive Failure Avoidance*
When an agent is faced with tight deadlines or when it has to rely on expensive and time-consuming services, it must deal with failures proactively by taking appropriate actions to reduce their probability of occurrence. This might include advance provisioning, using more reliable providers or including redundancy in workflows.

A.3. **Scalability**
The strategies that an agent employs must be scalable to large systems. As the agent might potentially interact with huge numbers of providers, any strategies must work well in systems of varying sizes. This similarly applies to workflows consisting of many tasks. More specifically, we expect our strategies to handle systems with thousands of providers and workflows consisting of hundreds of interdependent tasks[1].

A.4. **Adaptivity**
Our techniques must be able to adapt to new events as they occur, even if they were not initially planned for. Such adaptation might include selecting faster, more expensive services when a workflow begins to fall behind schedule, or, conversely, picking cheaper services when the agent does better than expected.

1.5 Research Contributions

Given the above requirements and our aim of developing suitable methods for building a service consuming agent, we have identified the process of *provisioning* services as a key area to investigate. Provisioning, i.e., the selection of particular service instances for specific tasks of a workflow, has received comparatively little attention in the research literature so far, but we believe that it is vital for controlling and mitigating nondeterministic service performance. Provisioning providers in an appropriate manner will allow a service consumer to differentiate between providers that offer a service at differing levels of quality or reliability and to invest extra resources in tasks that are particularly critical. Furthermore, re-provisioning providers on-the-fly enables the service consumer to respond quickly to failures and recover partially complete workflows without starting from scratch.

To this end, we develop a novel model for a distributed system and for simple workflows, which can be used in a wide range of application areas — from Grid to Web services and peer-to-peer systems. In the context of this model, we describe a number of provisioning strategies, and, in doing so, advance the state of the art in service provisioning as follows:

[1] We believe these numbers represent a challenging scenario given the typical numbers of providers and tasks in complex workflows (Li et al. (2004); Stevens et al. (2004); Zeng et al. (2004)).

1. We present the first algorithm that provisions multiple services redundantly for particularly critical workflow tasks in an automatic and principled manner, based on service performance information and the predicted benefit of doing so. Introducing such redundancy allows the consumer agent to decrease the probability of workflow failures, and we show empirically that our algorithm outperforms existing approaches that do not consider uncertainty or use redundancy in a static manner.

2. We explicitly consider the problem of *crash* failures in service workflows, where services appear to work on an assigned task, but in fact never return a result. While existing work assumes either timely error messages or manually fixed time-out values, we present, for the first time, a method for flexibly determining how long to wait for services before re-assigning a task to a different provider.

3. In developing the above two contributions, we show how our techniques can be applied in environments where different amounts of performance information is available about service providers. When this is extremely limited, the agent can use task-specific information to make fast decisions within a restricted decision space. In this case, we are the first to describe how the consumer can address uncertainty proactively without specific information to distinguish between providers. On the other hand, when the consumer has detailed knowledge about each provider, it can harness this to not only select the most appropriate one, but also to rely on multiple heterogeneous providers for a single task where this is beneficial.

4. When services are provisioned using advance agreements, we show that the service consumer can benefit significantly by gradually provisioning its workflow during execution, rather than provisioning all tasks at once (as is done by existing work). To enable such behaviour, we present a novel strategy that predicts the benefit of advance provisioning and balances this with the risk of losing agreed services due to task conflicts.

5. We discuss a highly adaptive provisioning strategy that continuously incorporates new information about service performance into its decision-making procedure at run-time, and changes its behaviour accordingly. In this context, it is the first strategy that uses such information not only to react to outright failures, but also to make additional provisions when the workflow begins to fall behind schedule, to reduce its investments when services perform better than expected, and to realise when to completely abandon an infeasible workflow.

These contributions have led to a number of peer-reviewed publications:

- **Stein et al. (2006):** Stein, S., Jennings, N. R., Payne, T. R. 2006. Flexible provisioning of service workflows. In *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI-06)*. pp. 295–299. IOS Press.

This paper describes a provisioning algorithm that uses redundancy (Contribution 1) and flexible service time-outs (Contribution 2) to address uncertainty in environments where highly limited performance information is available (part of Contribution 3).

- **Stein et al. (2007a):** Stein, S., Jennings, N. R., Payne, T. R. 2007. Provisioning heterogeneous and unreliable providers for service workflows. In *Proceedings of the 6th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-07)*. pp. 523–525. ACM Press.

  This short paper presents a new algorithm for provisioning services when these are highly heterogeneous (Contribution 3).

- **Stein et al. (2007b):** Stein, S., Jennings, N. R., Payne, T. R. 2007. Provisioning heterogeneous and unreliable providers for service workflows. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI-07)*. pp. 1452–1458. AAAI Press.

  This is an extended version of Stein et al. (2007a), presenting a more detailed discussion of the algorithm, further empirical results and an improved algorithm that finds a solution in less time.

- **Stein et al. (2007c):** Stein, S., Payne, T. R., Jennings, N. R. 2007. An effective strategy for the flexible provisioning of service workflows. In *Proceedings of the International Workshop on Service-Oriented Computing: Agents, Semantics, and Engineering (SOCASE-07)*. LNCS 4504. pp. 16–30. Springer.

  This paper extends the work in Stein et al. (2006) by improving the techniques used to predict the expected utility of a provisioned workflow. It also shows empirically that our algorithm is robust in the presence of inaccurate information about service providers.

- **Stein et al. (2008a):** Stein, S., Payne, T. R., Jennings, N. R. 2008. Flexible provisioning of web service workflows. *ACM Transactions on Internet Technology*, 8(4). ACM Press. (in press).

  This article is a long journal version of the contributions described in Stein et al. (2006) and Stein et al. (2007c). In addition to these, it contains further empirical results, a comparison of our heuristic to the optimal strategy, a bioinformatics application example and a significantly extended discussion of the context and limitations of our work.

- **Stein et al. (2008b):** Stein, S., Payne, T. R., Jennings, N. R. 2008. Flexible service provisioning with advance agreements. In *Proceedings of the 7th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-08)*. ACM Press. (in press).

  This paper describes a strategy that gradually provisions services using advance agreements (Contribution 4) and continuously adapts its decisions as new information becomes available (Contribution 5).

The research results presented in the above publications are summarised and expanded upon by this thesis. To guide the reader through the remaining chapters, the following section contains a brief outline of the thesis structure.

1.6 Thesis Outline

The remainder of this thesis is structured as follows: In Chapter 2, we conduct a thorough survey of current research in the context of our requirements. We are particularly interested in the extent to which our requirements are already met by current work, what models and frameworks we can build upon and where further improvements are needed.

Following this, Chapter 3 establishes a formal background for our work. In particular, we provide a formal model of a service-oriented system, we describe the workflows an agent faces, and we outline an example application scenario for our work, based on a bioinformatics workflow.

Then, in Chapter 4, we develop a novel provisioning algorithm for environments where limited performance information is available about service providers. In the same chapter, we conduct a thorough empirical evaluation and show that the algorithm performs significantly better than existing approaches. This chapter collates the results published in Stein et al. (2006), Stein et al. (2007c) and Stein et al. (2008a).

This is then followed in Chapter 5 by an extended system model that includes detailed performance information about heterogeneous service providers (i.e., where many providers offer the same type of service at varying levels of quality). In that chapter, we extend our algorithm for such environments and show that it outperforms other approaches. The chapter includes the results published in Stein et al. (2007a) and Stein et al. (2007b).

Next, in Chapter 6, we consider service-oriented systems, where providers and consumers reach explicit advance agreements about the provision of services. Again, we outline a flexible provisioning algorithm to address uncertainty in these systems and demonstrate that it performs better than the current state of the art. This chapter contains the results published in Stein et al. (2008b).

We conclude in Chapter 7 with a summary of our research and an outlook on future work. This is finally followed by the appendices, which provide further background information on our work. In particular, Appendix A investigates how sensitive it is to inaccurate service performance information, Appendix B discusses its scalability, Appendix C shows that the problems we consider are NP-hard, Appendix D derives in more detail some of the equations presented in the main body of the thesis and Appendix E lists the acronyms we used throughout our work.

Chapter 2

Literature Review

In this chapter, we provide a comprehensive overview of current research in the area of service provisioning, and evaluate this with respect to our specific requirements outlined in Chapter 1. The literature review is divided into four main sections. The first three, Sections 2.1–2.3, contain a survey of the background technologies that our work builds upon. Specifically, we discuss in more detail the paradigms of service-oriented computing (Section 2.1) and multi-agent systems (Section 2.2), motivate their use in the context of this research and provide some example applications. In Section 2.3, we briefly cover reliability engineering as a source of techniques for addressing uncertainty. In the fourth part of the chapter, Section 2.4, we focus on our overall research aim of executing complex workflows in service-oriented environments. To this end, we investigate current approaches and algorithms for provisioning and executing services that are part of workflows. Finally, we conclude in Section 2.5 by summarising our findings and relating them back to our original requirements.

2.1 Service-Oriented Computing

As stated in Section 1.2, service-oriented computing is a methodology for offering and consuming resources and functionality over a distributed computer system. Historically, the sharing of resources between distributed computers has often been considered and thus it is not a new concept in itself. However, most early systems were built for a special purpose and so they usually employed ad hoc mechanisms in order to interoperate (Kahn (1972); Knight (1972); Kang et al. (1988); Neches (1993)). This meant that the systems were inflexible, relied on static links between components and used application-specific protocols and data models (Singh and Huhns (2005)). When taken together, these factors led not only to large adoption costs, but also required complicated and expensive maintenance when components were added to or removed from a system (Casati et al. (2001)).

The Service-Oriented Computing (SOC) approach addresses these shortcomings by allowing services to be discovered and invoked automatically at run-time rather than through manually

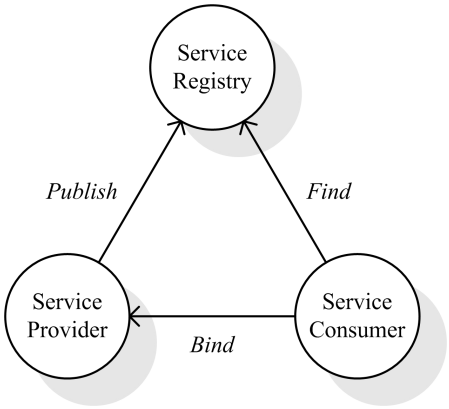


FIGURE 2.1: Basic model of the participants in a service-oriented system and their interactions (arrows originate from the usual initiators of the corresponding interactions).

specified and fixed application interfaces. To illustrate this and to further extend our brief introduction in Section 1.2, Figure 2.1 shows the basic interactions between service consumers and providers that are a central feature of most contemporary service-oriented systems (Agrawal et al. (2001); Papazoglou (2003); Huhns and Singh (2005)). Usually, a service provider will publish descriptions of its services on some registry, which is accessible to potential consumers. When a consumer requires a certain service, it will then search the registry, obtain the relevant information about providers offering the service, and start to communicate directly with a chosen provider.

The fundamental advantage of this process is that it is fully automated and requires no human intervention. This is achieved by using computer-readable descriptions of services, so that consumers can automatically match their requirements with service offerings and adapt to service-specific interfaces and protocols. In order to enable such automation, service-oriented frameworks normally rely on standardised data formats to describe services and their interaction protocols (e.g., WSDL and SOAP, which are discussed in Section 2.1.1).

In more detail, Huhns and Singh (2005) give several reasons why such an approach is appropriate for building large open systems consisting of many interacting computational agents. These include:

- Services are suitable abstractions of the functions that agents provide to each other (not least due to the analogous use of the word in the real world). Specifically, they are at a higher level than components in traditional software modelling approaches (e.g., objects or procedures), they enforce loose coupling and, hence, simplify the implementation of complex applications.
- Shared data formats, protocols and computer-readable service interfaces allow heterogeneous software components to communicate and interoperate, even if they were implemented in different programming languages, reside on various platforms and were never specifically designed to exchange data with each other.

- By publishing service descriptions in public registries, providers are not only able to advertise their services to a wide audience, consumers, in turn, benefit from a large choice of potential services to meet their needs. This allows them to provision appropriate services dynamically, depending on non-functional selection criteria, including cost, reliability, quality or trustworthiness.

These reasons, coupled with the more general trends outlined in Section 1.1, have led to a surge of interest in service technologies. Several major companies offer development platforms and tools for building service-oriented applications (such as Sun's Jini[1], Microsoft's .Net platform[2] and IBM's Websphere[3]). At the same time, a number of standardisation efforts have emerged to enable the interoperation of services, including CORBA (Yang and Duddy (1996)), Web service standards (Kreger (2003)) and the Grid service architecture (Talia (2002)).

Because of these trends and the reasons given above, we adopt SOC as a conceptual model for our own work. In the following sections, we introduce several representative service-oriented frameworks to give an indication of the current state of the art and to provide target applications for our research.

2.1.1 Web Services

Web services have become a popular technology for enabling the interactions shown in Figure 2.1 by providing common protocols and data formats for service consumers and providers to communicate over the Internet. In more detail, two core technologies govern the use of Web services: the Web Service Description Language (WSDL) and the Simple Object Access Protocol (SOAP). The former is a language for describing how to invoke a Web service, the operations it provides, and, in particular, the data types that the service expects and returns (Christensen et al. (2001)). Messages to and from service providers can be sent using SOAP, a protocol for exchanging XML documents over the Internet, usually using the Hypertext Transfer Protocol (HTTP) (Mitra (2003)). To complement these technologies, the Universal Description, Discovery and Integration (UDDI) specification defines a suitable service registry to allow providers to advertise their WSDL descriptions and other relevant information to potential consumers (Curbera et al. (2002)).

There are several reasons for the growing popularity of Web services. The ubiquity of TCP/IP and the potential for worldwide interconnectivity of applications certainly contribute to their success. Additionally, they can be built using free technologies and exchange data using the Extensible Markup Language (XML), which is widely supported on a range of platforms and programming languages (Bray et al. (2004)). Legacy applications can also easily be exposed as Web services, as usually only a small overhead is needed to create Web interfaces to existing

[1] `http://www.sun.com/software/jini/`
[2] `http://www.microsoft.com/net/`
[3] `http://www.ibm.com/websphere/`

applications (Coyle (2001)). Another advantage is the fact that specifications for describing and invoking Web services have been developed in a communal effort led by the W3C[4], resulting in increased standardisation and interoperability.

At this time, Web services are being widely adopted by a range of organisations. This can be witnessed by an increasing number of companies that offer their services over the Internet using the specifications described above. These include services that facilitate the automatic trading of goods (e.g., services offered by Amazon[5] and eBay[6]), information providing services (e.g., Web services by Google[7] and Yahoo[8]) and data processing services (e.g., EC2[9]).

In particular due to its spreading popularity, the Web services framework is a natural target domain for our research. Furthermore, it aligns well with the broad trends of distributed systems we identified in Section 1.1:

- Common protocols and platform-independent data formats allow **heterogeneous** services to communicate.
- Services are not directly under the control of their consumers and may therefore display **uncertainty** in their behaviour.
- Using the Internet as a basic infrastructure results in an inherently **open** system.

Moreover, there has been considerable work on addressing security issues within the Web services framework, which is a key concern in distributed systems (Naedele (2003)). Languages such as WS-Security and WS-Trust allow service providers and consumers to specify security mechanisms that they require for their interactions (Nadalin et al. (2006, 2007)). These mechanisms typically refer to well-established standards and protocols, e.g., Kerberos (Neuman and Ts'o (1994)) or X.509 (Cooper et al. (2008)), and address issues like the authentication of interaction partners, message encryption, message integrity preservation and access policies. As such, they are essential in providing a basic level of robustness to malicious attacks or eavesdropping by third parties. However, they do not directly address the uncertainty and autonomy of service providers, which may still defect or behave in an erratic manner (despite adhering to specified security policies).

Given the popularity of Web services, their suitability for open distributed systems and the existence of a robust infrastructure that already addresses basic security issues, we develop an abstract model of a service-oriented system in Chapter 3 that is based broadly on the current Web services framework. Although not explicit in the specifications, using Web services as a basis allows us to construct a model that includes uncertain service behaviours (Requirement

[4] `http://www.w3.org/2002/ws/`
[5] `http://aws.amazon.com/`
[6] `http://developer.ebay.com/`
[7] `http://code.google.com/`
[8] `http://developer.yahoo.com/`
[9] `http://aws.amazon.com/ec2/`

M.1), heterogeneity (M.4) and dynamism (M.5). Already, the Web services framework meets our requirement for covering on demand service invocation (M.3.a) by providing a mechanism that depends largely on just-in-time invocation of services.

However, Web services do not readily provide mechanisms that satisfy all of our requirements. In particular, current Web services are either provided free of charge or use ad hoc payment mechanisms, often external to the service framework (i.e., consumers arrange payment manually and through separate systems, as is the case for most companies mentioned above). Such a process is labour-intensive and incompatible with the vision of service-oriented systems where services are discovered and selected dynamically depending on the consumer's needs.

Additionally, Web services typically use simple on demand invocation (Curbera et al. (2002)), much like remote procedure calls (Nelson (1981)). This widely used metaphor is inappropriate for services that are offered by autonomous agents, because it implies that Web services are, like software procedures, dependable and predictable components. As argued in Section 1.3, this is an unrealistic assumption. Furthermore, on demand invocation precludes the possibility of provisioning services in advance — one of our requirements (M.3.b).

In the following sections, we continue to outline some of the most prominent emerging applications of SOC. Specifically, we describe *Grid computing* to highlight an important target domain of our work and show how service remuneration is beginning to be addressed (Section 2.1.2), we briefly mention *peer-to-peer* systems as a particularly dynamic and open environment (Section 2.1.3), and, finally, we give an overview of how the Semantic Web is envisaged to influence SOC (Section 2.1.4).

2.1.2 Grid Computing

Grid computing is an approach for sharing heterogeneous computational resources (services, data or simply spare processing cycles) across distributed systems using common protocols and data formats (Foster and Kesselman (1999); De Roure et al. (2003); Bernholdt et al. (2005)). For this reason, it is closely related to the field of service-oriented computing, which shares similar goals. However, Grid computing is a more specialised area that targets the domain of high-performance applications and inter-organisational collaborations. More specifically, Grid systems aim to allow companies and research institutes to pool their resources dynamically in order to collaborate on large projects or offer specialised services to each other.

This vision is outlined in detail by Foster et al. (2001), who introduce the notion of a *Virtual Organisation (VO)*. Here, a VO is a set of individuals and/or institutions that have agreed to collaborate and share resources, according to a set of well-defined access policies. These VOs are formed when the need for collaboration arises, for example as one company requires a high-performance image processing service or as several service providers combine their offerings to produce a new product. This VO formation process might involve agreements that are made

by humans outside the Grid system (Verma et al. (2002)), or it might be carried out automatically by software agents that search for and make agreements with appropriate service providers (Norman et al. (2004)).

These concepts of high-performance distributed computing and the formation of VOs among heterogeneous resource providers are central to Grid computing, and this is what defines the field, rather than a particular implementation. However, several systems and tools have emerged (Baker et al. (2002)). One prominent example is the Open Grid Services Architecture (OGSA) (Foster et al. (2002)), which is based on the Web services framework outlined in the previous section. In this model, Grid resources are offered as Web services, which have been extended to facilitate the formation of VOs. In particular, this is achieved by defining protocols for managing the lifecycle of services (so that service consumers can claim resources and release them as needed) and notification mechanisms that keep the consumers informed about the status of its services. The Globus toolkit (Foster (2005)) provides additional tools to implement Grid systems and includes a set of Grid services for common tasks (such as scheduling, monitoring and discovery services), a messaging infrastructure, security mechanisms and service containers that facilitate the deployment of new services. Many current Grid implementations use these technologies to manage increasingly large systems, from the Open Science Grid (Pordes et al. (2007)), the D-Grid (Gentzsch (2006)) to the National Grid Service (Geddes (2006)).

As such, Grid computing has so far concentrated on building the necessary infrastructure to allow large numbers of users access to shared resources. This has resulted in systems that are scalable and secure, but that are also tightly regulated by human administrators and that assume essentially cooperative participants. Foster et al. (2004) argue that this leads to considerable inflexibility, especially as Grid systems become increasingly heterogeneous and open. To address this, they propose a synergy of the robust Grid infrastructure with the more flexible decision-making, coordination and negotiation procedures of multi-agent systems. In a similar spirit as the overall aim of this thesis, such procedures would allow software applications to take decisions autonomously in open Grid systems, where service providers are not generally cooperative and where there is some competition between users.

Against this background, there has already been some interest by the research community in dealing with competition amongst service consumers. In particular, some have proposed the use of appropriate economic models to allocate Grid resources to consumers (Buyya et al. (2001)). While no standards or widely-used market mechanisms currently exist in the Grid domain, emerging work is addressing the need to account for the financial remuneration of service use. For example, the Grid resource broker Nimrod-G (Buyya et al. (2002)) accepts tasks from Grid users and then allocates them to distributed resources that charge for their services. The current implementation uses a simple pricing model that employs either fixed prices or demand-based functions to charge consumers, but the authors discuss at length a variety of other mechanisms, such as auctions or bilateral negotiation to determine prices. Unfortunately, the Nimrod-G broker is not directly applicable to our work, because it relies on a centralised mechanism that expects truthful service descriptions, does not take into account unreliable providers and offers

little scope for customised reward functions to guide its decisions (it minimises either cost or execution time). Nevertheless, the pricing models discussed by the authors offer a first step towards satisfying our requirement for addressing service remuneration (Requirement M.2), and we will use a similar fixed pricing scheme in our own model in Chapter 3.

While the emphasis of Grid computing has been on building large systems for high-performance computing with well-defined access policies, security and authentication mechanisms and clear hierarchies and boundaries of VOs, in the following section we examine an approach to distributed computing that does not rely on such a well-defined infrastructure.

2.1.3 Peer-to-Peer Computing

Another emerging paradigm for designing distributed systems is Peer-to-Peer (P2P) computing (Oram (2001); Schoder and Fischbach (2003); Chawathe et al. (2003)). In a similar vein to SOC and Grid computing, its goal is to enable the sharing of data or resources between large numbers of heterogeneous agents (Foster and Iamnitchi (2003)). However, the characterising feature of P2P systems is their self-organising and highly dynamic nature. Although initial systems such as Napster used central servers to track participants and their shared resources (Saroiu et al. (2003)), current P2P systems are often entirely decentralised. Rather than relying on a fixed infrastructure or central servers, these systems form ad hoc networks that connect each participating agent to a set of *peers* (Zhao et al. (2004)). Each peer is, in turn, connected to others, forming an *overlay* network that is independent from the underlying transport network (such as the Internet). In these networks, various techniques are employed to allow agents to discover shared resources. They include *flooding*, where queries for resources are sent to all neighbours and then propagated through the network (Saroiu et al. (2003)), or *distributed hash tables*, which structure the network such that queries can be efficiently routed to relevant participants (Ratnasamy et al. (2001); Stoica et al. (2001)).

While initial applications for P2P systems were restricted to file sharing between home computer users (Matei et al. (2002)), the area has spawned several efforts aimed at exploiting processing cycles of idle desktop computers (Richards (2002); Anderson et al. (2002)). Furthermore, some work has been carried out to establish generic service frameworks on top of P2P systems (Gong (2001); Anderson (2004); Stäber and Müller (2007)), but none has so far been widely adopted and most applications in this field remain specialised to one particular purpose (such as file sharing).

As P2P systems have seen wide deployment with large numbers of participants (rather than the small-scale Grid prototypes in use today), they most clearly show some of the trends we predicted in Section 1.3. In particular, they highlight the self-interested nature of participants, the need to deal with failures and the requirement of employing appropriate mechanisms to incentivise providers to offer their services (Golle et al. (2001)). For these reasons, P2P systems first offer some simple techniques that may help satisfy our Requirement A.2. Application

failures, for example when an agent leaves the network before completing its service, are usually addressed either by trying to contact a different provider or by requesting the same service redundantly from several providers (Milojicic et al. (2002); Friese et al. (2005)). However, these approaches are specific to the application, do not take into account the potential costs of service failures (or unnecessary redundancy), nor do they use domain knowledge to react appropriately to particularly failure-prone tasks or untrusted service providers. Nevertheless, these methods form the basis of a more flexible service provisioning strategy that we develop in Chapter 4.

Having discussed several important target domains for our work, we now examine a recent field that addresses some critical shortcomings of current SOC approaches by drawing on research in the Semantic Web.

2.1.4 Semantic Web Services

The current state of the art in SOC, as exemplified by the Web services framework in Section 2.1.1, addresses some fundamental issues concerning the interoperability of heterogeneous software components by standardising shared protocols and data formats. However, the facilities for *dynamically selecting* services are severely restricted. As discussed, WSDL descriptions published on UDDI registries provide the main mechanism for discovering appropriate services. Unfortunately, these descriptions contain information only about the data types that the service accepts and returns and are hence purely syntactic in nature. Yet, to achieve dynamic selection, a computational agent needs to understand the *meaning* of a service in relation to its own goals, i.e., whether a given service is actually sufficient for the task at hand. This is particularly important as services are offered across organisational boundaries, are written and maintained by different programmers and hence follow different usage conventions.

To address these shortcomings and to enable the dynamic discovery of services, the area of Semantic Web services is concerned with employing logical formalisms for describing services (McIlraith et al. (2001); Burstein et al. (2005)). These formalisms are rooted in technologies emerging as part of the Semantic Web, an effort to present knowledge in a computer-readable format across distributed information sources (Berners-Lee et al. (2001)). At the core of these fields are *ontologies*, formal specifications of conceptualisations (Gruber (1993)) (explicit descriptions of abstract concepts and their logical relationships), and the objects that populate them (concrete instantiations of abstract concepts).

It is envisaged that providers will be able to describe their services in a more expressive way than has hitherto been possible by using such ontologies. This is due to several advantages that these formalisms offer over the purely syntactical WSDL description. First, ontologies allow service providers to use common service vocabularies, which describe unambiguously the characteristics that all services share, including how to specify the inputs and outputs, interaction protocols and the effects of services (Martin et al. (2004a)). Second, as Semantic Web technologies are specifically designed to allow the distributed representation of knowledge, service descriptions

can refer to concepts and objects that are defined elsewhere (Hendler (2001)). This means that services can use the vocabulary of other domains, for example to describe the types of data they accept, how they interact with real objects or even how they relate to other service instances. Third, the use of logical formalisms allows consumers to *reason* about services. Hence, the consumers can pose complex queries about the types of services they need, including specific constraints, and then use reasoning algorithms to find matching instances (Paolucci et al. (2002)). Even when the service providers and consumers use different ontologies or present incomplete information, as is likely in such heterogeneous scenarios, appropriate knowledge and mapping rules from distributed sources may help the consumer reason across different vocabularies and find appropriate matches (Noy and Musen (2002)).

To date, several approaches have been proposed for describing Web services using ontologies. OWL-S is a service ontology (Martin et al. (2004b)), expressed in the Web Ontology Language (OWL), a popular description language developed in the context of the Semantic Web (McGuinness and van Harmelen (2004)). As such, it builds on parallel efforts to describe knowledge on the Semantic Web, and OWL-S, as well as its predecessor, DAML-S, have been widely used in research on Semantic Web services (Narayanan and McIlraith (2002); Gibbins et al. (2003); Sirin et al. (2003); Wu et al. (2003)).

Another approach, the Web Service Modeling Ontology (WSMO) (Roman et al. (2005)), also provides an ontology for describing services. However, while the overall aims of these efforts are similar, WSMO differs from OWL-S both in the formalism employed and its overall scope. WSMO uses its own family of logical modelling languages, most of which are based on Frame Logic (Kifer et al. (1995)). Furthermore, it includes an associated execution environment, responsible for discovering, aligning and invoking services. While this extends the functionality of the approach, it also removes the autonomy of the service consumer to choose its preferred services and control the reasoning process.

Finally, a generic language for specifying Semantic Annotations for WSDL and XML Schema (SAWSDL) has recently been published as a W3C recommendation (Kopecký et al. (2007)). This language allows WSDL descriptions to refer to semantic concepts from ontologies, for example to formally define Web service operations or their input and output parameters. As such, it builds closely on an established technology and can therefore easily be used to extend existing service descriptions. However, unlike OWL-S and WSMO, it does not suggest a generic service ontology and it is agnostic towards any specific formalism for representing semantic concepts.

In summary, semantic descriptions of services are an important step towards enabling the dynamic discovery and selection of services. While the research outlined here does not satisfy any of our requirements directly, we mention it because it is developing a vital enabling technology for our work, and we envisage our techniques to extend and complement methods in this area. It should also be noted that significant work on the dynamic composition of services

has emerged from the field of Semantic Web services. As this topic is related to our goal of executing workflows of multiple services, we will return to it separately in Section 2.4.2.

As we have seen so far, most work in the area of service-oriented computing has concentrated on describing the functional properties of services and on enabling heterogeneous applications to communicate over a common network infrastructure. However, there is an increasing interest in describing the non-functional characteristics of services, in order to enable consumers to provision suitable services and to reach explicit contracts with providers. We will outline this research in the following section.

2.1.5 Quality-of-Service

A growing body of research is beginning to address Quality-of-Service (QoS) issues in service-oriented computing (Menasce (2002); Ran (2003)). This research acknowledges that relying only on functional service descriptions is insufficient for the widespread adoption of service-oriented computing in large distributed systems, and so considers how to describe, monitor and publish the non-functional properties of services. In particular, many of the service technologies described earlier have been extended to include properties such as the cost, reliability, response time, availability and privacy or security guarantees of a service. For example, D'Ambrogio (2006) describes an appropriate extensions for WSDL, Wang et al. (2006) show how a QoS ontology can be added to WSMO and Zhou et al. (2004) develop a similar ontology for DAML-S. These formalisms allow consumers to reason about the non-functional aspects of services and formulate more expressive service requests than in frameworks that only consider functional properties.

There have been several proposals on how this QoS information can be aggregated and made available to consumers. Ran (2003) describes an extended UDDI registry that allows providers to publish their non-functional service characteristics and suggests the use of a neutral third party to certify that these are in fact truthful. In the Web Services Agent Framework (WSAF), Maximilien and Singh (2004) use service proxies that automatically select appropriate service implementations based on a consumer's service request with QoS constraints. These proxies then monitor the performance of the selected service, collect feedback from the consumer and use this information to build up more accurate performance profiles for future service selection.

While the above technologies are used to describe the general QoS characteristics of services, the Web Service Level Agreement (WSLA) language allows a service consumer and provider to express specific terms for their interactions in the form of a machine-readable contract (Ludwig et al. (2003)). Besides covering standard performance terms (cost, reliability, response time and so on), such a Service Level Agreement (SLA) specifies how the performance metrics are calculated and monitored at run-time (possibly enlisting the support of a third party) and how the parties should respond in case the terms are violated (e.g., by notifying the consumer or even by paying a financial compensation).

Generally, there has been some interest in enabling explicit advance agreements between consumers and providers for the provision of services. Web Services Agreement (WS-Agreement) also provides a language for specifying SLAs, but additionally considers the overall lifecycle of an agreement, including its initial negotiation, possible re-negotiation and expiry (Andrieux et al. (2007)). However, both WS-Agreement and WSLA only define the necessary languages and protocols to reach an agreement, without describing how computer applications might make automatic decisions about these at run-time (in Section 2.2, we will discuss in more detail how technologies from the field of multi-agent systems may help automate these decisions.).

In the context of Grid computing, Czajkowski et al. (2005) argue that advance provisioning will become more important over the coming years and gradually replace the current practice of on-demand provisioning, where resources are made available only when they are actually needed by the consumer. The authors believe that this will lead to higher reliability and quality of services, and allow consumers a higher degree of control and flexibility when choosing their services. This view is supported by an empirical study conducted by Singh et al. (2007). Here, the authors propose a strategy that provisions Grid resources in advance for a workflow, given a set of offers from all service providers (which are assumed to be reliable). They show that their strategy begins to outperform an approach based on on-demand provisioning as workflows become increasingly parallel and there is an increasing load on the system. Specifically, their strategy completes workflows in a shorter and more predictable amount of time at a similar cost.

In summary, the current work on QoS in service-oriented computing is a promising development that will help consumers address the uncertainty and dynamism in large distributed systems. In particular, consumers can use the performance information available through QoS ontologies and repositories to select more reliable services that are appropriate for their workflows, and we will discuss a number of current approaches for this in Section 2.4.3. Additionally, explicit service contracts and advance provisioning further reduce the uncertainty in service-oriented systems, as consumers can negotiate over the time-scales of services and any penalties that should be imposed in case of failure. However, using such a contract model does not in itself provide a reliable system — providers may still fail or defect maliciously, possibly leaving the system without paying the agreed penalties.

The work mentioned here is vital for addressing our model requirements. First, QoS ontologies and related work on monitoring service behaviour over time allows us to express non-functional service parameters of hetergeneous providers (Requirements M.1, M.2.a, M.4 and M.5). Second, work on advance provisioning and SLAs (along with the negotiation techniques that we discuss in Section 2.2.2) is an important enabling technology for reaching advance agreements and to provide flexible pricing mechanisms (Requirements M.2.b and M.3.b).

Now, having discussed service-oriented systems, applications and related technologies, we turn towards the field of multi-agent systems. This research area has addressed some issues that are central to our research, but that have so far been largely overlooked by work on SOC. These include the need to make rational decisions in uncertain environments, to model service providers

as automous agents that do not always behave as they are told, and to place interactions of self-interested agents in the context of appropriate remuneration mechanisms.

2.2 Multi-Agent Systems

In Section 1.3, we introduced the notion of autonomous agents and outlined their importance in the context of large-scale distributed systems. Specifically, we used them as a metaphor for service providers and argued that such providers are self-interested and therefore inherently unreliable. While autonomous agents provide the general motivation for our work, in this section, we turn our attention towards particular research topics and techniques that were borne out of research into multi-agent systems.

To this end, in Section 2.2.1, we first look at general techniques for building computational agents that make decisions on behalf of their human users. Then, in Section 2.2.2, we examine the economic mechanisms and protocols that have been developed to help self-interested agents reach mutually beneficial agreements. In Section 2.2.3, we discuss current approaches for modelling the trustworthiness of agents and exchanging this using reputation mechanisms. Finally, in Section 2.2.4, we highlight some existing work that already employs agent-based techniques to build distributed service-based systems.

2.2.1 Building Decision-Making Agents

The autonomous self-interested agent is a metaphor suitable not only for characterising service providers, but also for helping build a service-consuming agent that makes decisions in uncertain, dynamic environments (recall our overall objective given in Section 1.4). Several approaches exist for building agents in general (Müller (1996); Wooldridge (2002); Russell and Norvig (2003)), including *reactive* approaches that display emergent behaviour by applying simple rules (Brooks (1986)) or logic-based *reasoning* agents that manipulate symbolic knowledge in order to produce plans that fulfil their goals (Georgeff et al. (1999)). While each of these techniques has seen some successful applications (Luck et al. (2006); Belecheanu et al. (2006)), neither applies directly to our work, because we need to take into account uncertainty (Requirement M.1) and an economic setting where agents are self-interested and expect some financial remuneration (M.2).

Instead, we turn towards a field that has been widely applied in settings which include decisions with uncertain consequences: *decision theory* (Raiffa (1968)). This field has recently emerged as an important source of techniques for building computational agents, because of its solid mathematical foundation for making the “best” decision under uncertainty (Parsons and Wooldridge (2002)).

At the core of this work is the concept of a private *utility function* $u : S \to \mathbb{R}$ that maps a given situation s from the set of all situations S to a real number $u(s)$ — the decision-maker's value, or utility, of being in the situation s. Such a utility function represents the preferences of the decision-making agent, so that the agent prefers being in situation s_1 to being in s_2 if and only if $u(s_1) > u(s_2)$. The existence of this function arises from several basic assumptions regarding the agent's preferences (Von Neumann and Morgenstern (1944)). These assumptions (such as orderability and transitivity of preferences) furthermore allow us to calculate the *expected* utility of a gamble that involves several probabilistic outcomes. This is simply done by taking the sum of the utility values of all potential outcomes, multiplied by their respective probabilities. Hence, we can write this *expected* utility as follows:

$$u([p_1, s_1; \ldots; p_n, s_n]) = \sum_i p_i u(s_i) \tag{2.1}$$

where $[p_1, s_1; \ldots; p_n, s_n]$ denotes a gamble that results in situation s_i with probability p_i (with $\sum_i p_i = 1$).

Now, given this utility function to express preferences between situations and gambles with uncertain outcomes, decision theory includes the decision maker's actions in this model. This is done by treating each decision as a gamble with different outcomes and associated probabilities. Hence, a decision d_j is treated as a gamble $[p_{1j}, s_{1j}; \ldots; p_{nj}, s_{nj}]$, where p_{ij} is the probability that the decision will lead to situation s_{ij}. We can then express the expected utility of a decision d_j as follows:

$$u(d_j) = \sum_i p_{ij} u(s_i) \tag{2.2}$$

Attaching such utility values to decisions offers an obvious tool for choosing one member of a set of several possible decisions D (a *decision problem*): the *principle of maximum expected utility* (Lindley (1971)). According to this, a decision-maker should always choose the decision d^* that maximises the expected utility $u(d^*)$:

$$d^* = \underset{d_j \in D}{\operatorname{argmax}}\, u(d_j) \tag{2.3}$$

This leads us to the definition of a *rational* agent:

Definition 1 (Rational Agent)**.** When faced with a decision problem D, a rational agent always chooses a decision $d^* \in D$ that maximises its expected utility $u(d^*)$.

In practice, rationality can be difficult to achieve — it may be impossible to enumerate all possible decisions in a given situation or there may simply not be sufficient time for the required

deliberation. Simon (1957) describes this problem (in the context of a human decision-maker) as the principle of *bounded rationality*:

> The capacity of the human mind for formulating and solving complex problems is very small compared with the size of the problems whose solution is required for objectively rational behavior in the real world — or even for a reasonable approximation to such objective rationality. (Simon, 1957, page 198)

Furthermore, he argues that, as a consequence of this bounded rationality, humans construct highly simplified models of the world when making decisions. They then make decisions that are rational with respect to this model, but not even approximately optimal had they taken into account all options and information available to them. Essentially, these models provide decisions that are "good enough", or *satisficing*, for the purposes of the decision-maker without maximising the overall achievable outcome.

Such a view of decision-making processes is similarly applicable when building computational agents. Many seemingly simple decision problems are known to be intractable and so illustrate the difficulty of finding an optimal solution within limited time (Brassard and Bratley (1996)). Yet, in many applications, computer agents need to arrive at solutions within a reasonable amount of time, using only the memory and information that is currently available.

For example, an agent controlling a spacecraft may detect a deviation from its course and must decide whether to activate its thrusters and at what power (Muscettola et al. (1998)). Now, the agent's decision making process is not only bounded by limited memory, processing speed and inaccurate sensor information, it is effectively situated in a dynamic environment. While deliberating, the spacecraft continues to move, perhaps deviating further from its course. Similarly, the agent could take more sensor readings to improve its information, but again lose valuable time (and deplete its energy).

In our work, we face similar challenges as outlined by the example above. The systems we consider are dynamic (Requirement M.5), workflows may have strict time constraints (W.2) and our methods must be scalable (A.3). More explicitly, Requirement A.1 states that our agent must make good decisions within the bounds of its limited computational capacity and knowledge.

In order to tackle such difficult problems, Simon and Newell (1958) proposed the use of *heuristic* methods. They modelled these on human problem-solving processes and predicted that such methods would enable computers to deal with problems previously deemed intractable (such as chess-playing). In the current literature, heuristics are usually algorithms based on simplified models of complex domains (Russell and Norvig (2003)). While heuristics can help derive optimal solutions without the need for an exhaustive search (Pearl (1984)), they often solve simpler, tractable problems or iteratively improve a candidate solution until a satisfactory answer is found without necessarily guaranteeing this to be the overall maximum (Golden et al. (1980); Kirkpatrick et al. (1983); Michalewicz and Fogel (2004)). Such heuristics resemble very closely

the bounded rationality that Simon observed in human decision-making, and we will use similar techniques in this thesis to tackle difficult decision problems.

Now, heuristics are often successful in reducing the complexity of problem-solving algorithms. This might be achieved, for example, by suggesting a polynomial-time approximation to a problem for which all known exact algorithms have an exponential running-time (Golden et al. (1980)). Furthermore, there has been considerable effort in the area of *anytime* algorithms (Dean and Boddy (1988)). These can be interrupted after an arbitrary amount of time t to provide a solution whose quality is a monotonically increasing function of t. Such heuristics are particularly well suited for the problem we consider (and in particular our Requirement A.3 for scalability), as they can be applied in complex environments where time is a critical resource. Finally, there has also been substantial work on determining decision-theoretically how much time t to allocate to the solving of a problem (Horvitz (1988); Boddy and Dean (1994)). Such work is a promising approach for building boundedly rational agents that take the best possible decisions within a computationally limited framework.

The techniques discussed in this section will help us build a flexible decision-making agent capable of operating in the dynamic and uncertain environments we consider. In particular, we outlined some general methodologies employed to enable agents to make rational decisions, and thus satisfy our Requirement A.1. In the following section, we examine how current agent-based research may help us address Requirements M.2 and M.3.

2.2.2 Cooperation through Negotiation

As discussed in Section 2.1.1, current approaches in Web services typically assume that providers offer their services unquestioningly and free of charge through remote procedure calls. Although we described some work in Section 2.1.5 that is beginning to address the formation of advance agreements, it has so far mostly looked at the syntactic definition of contracts and not how they can be formed automatically by service consumers and providers. This means that such contracts are typically drawn up manually by human administrators. This is not tenable in highly dynamic environments, where service providers and consumers are envisaged to discover and engage each other automatically, and where they are self-interested agents who seek to benefit from their interactions (as argued in Section 1.3).

In order to reach mutually beneficial agreements, *automated negotiation* has been suggested as a powerful technique in multi-agent systems (Rosenschein and Zlotkin (1994); Jennings et al. (2001)). This is essentially a distributed search through the space of potential agreements among several autonomous agents, involving the interchange of relevant information and ultimately aiming to find an agreement that is acceptable to all participants.

In more detail, Figure 2.2 shows an example negotiation between a provider and a consumer that demands service X to be performed immediately. Because the provider is unable to fulfil this request, it responds by making a counter-offer for the same service, but with a delay of ten

minutes. Also, because it expects remuneration for its effort, it includes a new issue in its offer — the price for the service. Now, the prospective consumer concedes by accepting some delay and a payment for the service, but it does not agree with the amounts proposed by the provider, and so it makes another counter-offer. This is then agreed to by the provider and becomes binding for both agents.

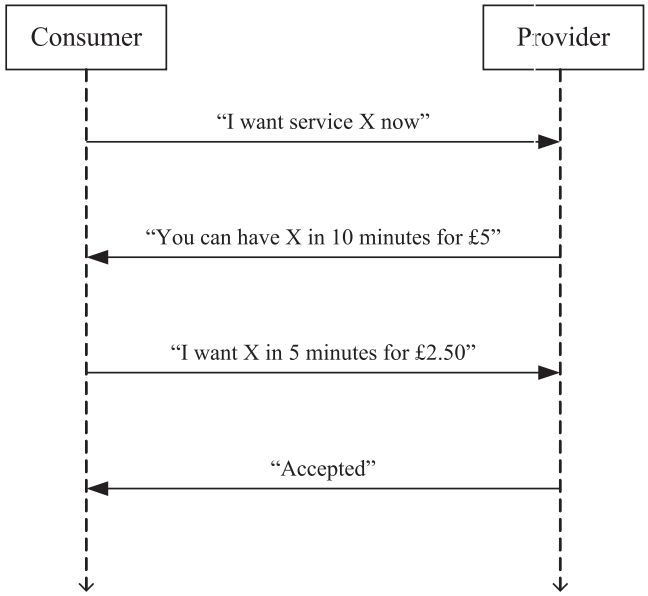


FIGURE 2.2: Example of two agents negotiating over the provision of service X.

As illustrated by this example, there are several key advantages that make negotiation a suitable model for the interactions of self-interested agents in a service-oriented context. First, it makes explicit the need to find mutually beneficial agreements and includes the possibility that service providers are unwilling to cooperate. Second, it offers considerable flexibility by allowing agents to explore the space of possible agreements. Hence, new issues can be introduced that one agent did not consider, the agents can seek compromises and they can coordinate if they have incompatible goals or other commitments.

One of the first negotiation protocols[10] for computational agents was the *contract net* (Smith (1980)). Figure 2.3 shows an example negotiation using this protocol. First, a service-consuming agent announces its requirements for a specific task to a set of potential providers. If they are available and willing to carry out the task, the providers then bid for the task by replying to the consumer with their individual characteristics (e.g., the expected quality of their services and associated costs). Finally, the consumer chooses the most promising bidder and awards the task to it.

This protocol has been popular due to its simplicity, distributed nature and its close resemblance to non-automated contract procurement (especially in the public sector, where such a procedure is often mandatory). For these reasons, it has been successfully adopted in a variety of domains,

[10] A negotiation *protocol* is the mechanism that governs how agents can exchange information, make proposals and reach an agreement.

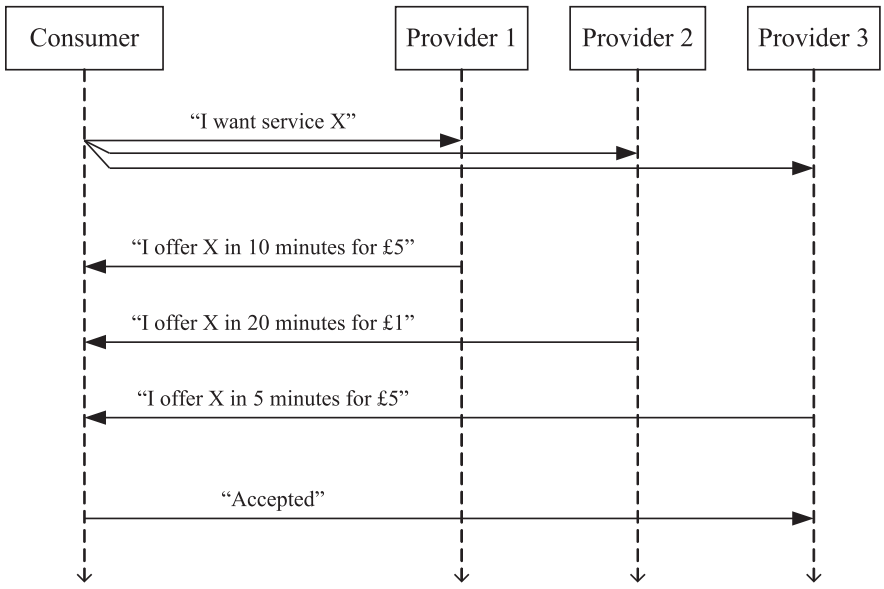


FIGURE 2.3: Example negotiation using the contract net to provision service X.

from transportation (Kuhn et al. (1993)), to the management of manufacturing systems (Maturana and Norrie (1997)) and robotics (Botelho and Alami (1999)). Furthermore, in the context of Web services, Paurobally and Jennings (2005) use the contract net protocol to exemplify how agents can negotiate about WS-Agreement contracts.

Nevertheless, the contract net protocol has a number of disadvantages. It was originally suggested in the context of a system where agents are not self-interested, but rather fully cooperative. Hence, service providers are assumed to report truthfully on their capabilities, there are no formal clearing rules that determine the winner of a contract and the service consumer is not required to accept any bids (nor to notify those bidders that it rejects) and it may even cancel accepted bids at any time before they are completed. For these reasons, the contract net protocol may discourage self-interested service providers from participating. Similarly, there is no obvious strategy for submitting bids — instead, bidders are forced to speculate about the bids of other agents and any potentially better offers they might receive from other consumers in the future (Sandholm (1999)).

Now, in order to develop more suitable negotiation mechanisms for self-interested agents, there has recently been considerable interest in applying game theoretic principles to multi-agent interactions (Sandholm (1999)). In particular, *mechanism design* (Dash et al. (2003)) is concerned with designing protocols that can be proved to show certain desirable properties, such as stability (rational agents will act in a predictable manner), individual rationality (agents are better off by joining the protocol) and Pareto efficiency (it is not possible to improve the outcome for one particular agent without decreasing the utility of another).

In this context, several protocols for bilateral ("one-to-one") negotiation have been proposed. These include simple mechanisms such as the monotonic concession protocol (Rosenschein and Zlotkin (1994)) or the *alternating offers* mechanism suggested by Rubinstein (1982) (where

alternating offers are exchanged as shown in Figure 2.2). Some of these have already been successfully applied in service-oriented systems. For example, Faratin et al. (1998) modify the alternating offers mechanism to allow agents to negotiate over the terms on which a service is provided.

Another prominent negotiation mechanism that involves many participating parties are auctions. In an auction, a seller usually offers some item or service for sale, which is then bid for by the buyers. Associated with such an auction are strict rules about how bids should be placed, when the auction finishes and how it should be cleared (who receives the item and at what price). As for other types of negotiation, these rules can be engineered to guarantee certain game theoretic properties, and auction theory provides a vast array of auction types for different requirements (Krishna (2002)). In the context of service-oriented computing, Vulkan and Jennings (2000) describe how reverse auctions can be used by consumers to solicit bids from many competing service providers.

A problem with the negotiation schemes presented so far is that agreements are always binding. That is, once a seller agrees to provide a service, it must do this exactly as promised. However, in realistic scenarios, resource availability changes, services take uncertain amounts of time and a service provider may not have sufficient time to evaluate all contingencies before agreeing to provide a service. For this reason, binding agreements may discourage providers from participating in the mechanism or result in extremely pessimistic strategies, where a provider only agrees to provide a service if it is certain of its success.

To address this problem, Sandholm and Lesser (1995a) describe the *leveled commitment contracting protocol*. Here, the agents include explicit decommitment penalties in their negotiations. Rather than acting as deterrents for defection (as is common in the legal domain), these penalties allow each agent to drop its commitment to the contract by paying a fixed amount of money. To demonstrate the value of this approach, the authors prove that it allows agents to reach agreements in scenarios where this would otherwise not have been possible. Furthermore, they show that both agents can benefit (derive a higher expected utility) from agreeing to a leveled commitment contract rather than a fully binding one. In other work, Sandholm and Lesser (1995b) modify the contract net protocol to include leveled commitments. In this modified protocol, consumers and buyers are committed to any offers they make (including the initial announcement), but may decommit by paying the appropriate penalty.

We believe that this contracting protocol is a realistic and practical negotiation mechanism for the uncertain distributed systems that we consider. Partly for this reason, we adopt it in Chapter 6 to model the automatic negotiation of advance agreements in a service-oriented system. Furthermore, it is simpler than some of the other protocols we have discussed, and so adopting it allows us to concentrate on building a more generic decision-making agent that can be extended (in future work) to more specialised market mechanisms.

To summarise this section, carefully engineered negotiation protocols can display a number of desirable properties that can entice self-interested agents to participate and come to mutually

beneficial agreements. However, it should be noted that they often make restricting assumptions. These might include particular types of cost and utility functions that all participants share or it might be the need for agents to be perfectly rational. In particular, mechanism design often relies on agents that never deviate from a given protocol. For example, when an agent fails to provide its service in the leveled commitment protocol, it is assumed to notify its consumer and pay the appropriate penalty. This is a reasonable assumption for contracts between human traders, which are enforceable through legal measures. However, as we already argued in Section 2.1.5, such legal enforcement may be difficult or too costly to pursue in large-scale open systems, where agents continuously join and leave, where identities may be hidden and where services are offered across national boundaries. Therefore, it is conceivable that malicious agents may enter contracts that they cannot honour or that later turn out to be infeasible. Because we envisage this to be a critical problem in large distributed systems, we made the treatment of such uncertain provider behaviour a central requirement of our research (M.1 and A.2).

Against this background, a large body of work in the area of trust and reputation has looked at how to model such uncertain and possibly malicious behaviour. We discuss this in more detail in the following section.

2.2.3 Trust & Reputation

Current online marketplaces identify unreliable traders by using rating systems, where the human users leave feedback for each other (e.g., eBay[11] or Amazon Marketplace[12]). Such systems allow users to trust each other even if they have not previously interacted, and to avoid persistently malicious participants. Against this background, recent work in multi-agent systems has been concerned with building similar mechanisms to assist autonomous agents in making decisions about their potential trading partners (Ramchurn et al. (2004)).

At the simplest level, trust in these models is based purely on past interactions with other agents. When an agent honours a contract, this will be remembered and has a positive effect on any future decisions to interact with that agent, while defection produces the opposite, negative effect. However, relying solely on such experience is of limited use, because it is impossible to judge the behaviour of potential partners with whom no previous interactions have taken place. Hence, a second level of trust, usually referred to as reputation, is placed on a system, whereby agents exchange their experiences to form public opinions of others in the system (Sabater and Sierra (2002)). This approach is non-trivial and remains an open research challenge, because agents can lie and collude to influence the reputation of other agents. They must have an incentive to share their experience, and care must be taken not to prejudice against new entrants to the system, but at the same time discourage agents with a bad reputation history to leave and re-enter the system. These issues are typically overlooked by the work on QoS (Section 2.1.5), and so

[11] `http://www.ebay.co.uk/`
[12] `http://www.amazon.co.uk/marketplace/`

it is vital that we consider current approaches for modelling trust and reputation in multi-agent systems.

Now, there are many such approaches and models that differ in their representation and aggregation mechanisms (Ramchurn et al. (2004); Teacy (2006); Jøsang et al. (2007)). Some build their own frameworks to describe degrees of trust using discrete or continuous values (Abdul-Rahman and Hailes (1997); Sabater and Sierra (2002)), but these often lack the semantic grounding of a well-established formalism. This is addressed by other work that employs probability theory to represent trust. These approaches typically model trust as a probability distribution over a binary event, i.e., the probability that the agent performs the service that is required by the consumer (Ismail and Jøsang (2002); Wang and Vassileva (2003)).

Specifically, Teacy et al. (2006) outline a particularly interesting approach that uses principled probabilistic methods to combine direct observations with reputation reports from possibly inaccurate sources. In particular, their work uses statistical techniques to establish the confidence of an agent in its trust values towards other agents based on the number of previous interactions and then improves these, if necessary, by including the opinions of other agents. In so doing, it filters out opinions that seem improbable, given the agent's own experience and so their mechanism achieves some robustness against untruthful or noisy opinions. In further work, Teacy (2006) shows how this model can be extended to represent continuous outcomes, such as the duration of a service invocation.

In summary, the above work on modelling trust is vital as an enabling technology for our own work. It offers feasible solutions for aggregating opinions about service providers who are possibly unknown to the consumer and for instantiating QoS ontologies without relying on a neutral and centralised observer. For these reasons, information provided by a trust model may help us describe some of the uncertainty that providers display (Requirement M.1) and distinguish between heterogeneous providers that offer the same type of service (Requirement M.4). Furthermore, research on modelling trust probabilistically provides us with a formal mechanism for describing uncertainty and fits naturally with the work on decision theory outlined in Section 2.2.1. Hence, it will help us build a principled decision making framework under uncertainty (Requirement A.1).

We now conclude our summary of multi-agent systems by looking at two systems that apply agent-based techniques to service-oriented scenarios. Both of these research projects aim at providing a basic infrastructure over which agents can negotiate about the provision of services. We present these here, because they offer what we believe to be a more realistic model of how services will be provisioned in distributed systems (rather than the remote procedure calls predominantly used by Web services).

2.2.4 Example Agent-Based Applications

The first system we highlight is the Multi Agent Negotiation Testbed (MAGNET) (Collins et al. (2002)). The aim of this work is to provide a central marketplace, over which service consumers and providers can interact, and which enforces negotiation protocols and monitors the performance of agents. As such, it offers a centralised trust mechanism, but introduces a single point of failure and raises the question of whether the mechanism itself can be trusted.

In MAGNET, the market allows for many negotiations to run concurrently between many heterogeneous agents. While the architecture is intended to eventually support a number of different negotiation protocols, the authors describe only one mechanism modelled on a combinatorial auction. Here, an agent initiates a reverse auction by submitting a request for quotes to the market. This request includes a set of tasks that the agent needs to outsource, appropriate precedence constraints (e.g., task t_1 has to complete before starting t_2) and time restrictions (e.g., task t_1 must be started and completed in a given time interval). Interested service providers then submit sealed bids on combinations of tasks. Finally, the consumer determines the winners by minimising the overall cost while satisfying the precedence constraints between tasks. As such, the system demonstrates how a complex service-oriented system can be built using an established market mechanism.

However, the system also suffers from a number of weaknesses. Apart from otherwise being completely centralised, the responsibility of determining the auction winners is shifted to the service consumer. This essentially means that the consumer is not bound to the auction protocol and can reject any of the offered bids — even if they are in the optimal (least-cost) set of bids. Furthermore, even if the winner determination problem was solved by the neutral market, this process is notoriously difficult (Sandholm (2002)) and may lead to scalability problems. Nevertheless, some interesting work has emerged from this framework regarding uncertain service providers. We will return to this during our discussion of current provision techniques in Section 2.4.3.3.

The second multi-agent system we consider is the Advanced Decision Environment for Process Tasks (ADEPT) (Jennings et al. (1996)). This constitutes a less centralised approach for negotiating over the provision of services. The overall aim of this framework is to provide an infrastructure for handling complex organisational workflows. Recognising that such workflows are usually distributed across several companies, and even separate departments within one organisation, each of which has their own goals and agendas, the authors suggest the use of autonomous agents as a natural design metaphor. Not only does such a metaphor encapsulate the distribution of responsibilities (each agent manages part of the workflow, possibly using the services of others), it also deals with conflicts of interest by forcing agents to provision services in advance through negotiation. Such a mechanism ensures that services are provided to those that need them most, it allows agents to coordinate, and it provides some resilience against failures, because services can be renegotiated at run-time.

Now, ADEPT is a good model for our own work for several reasons. First, it allows several negotiations as well as negotiation protocols to operate concurrently and in a distributed manner (as outlined, for example, by Faratin et al. (1998) and Vulkan and Jennings (2000)). As stated by our Requirements M.2 and M.3, we envisage a distributed system to offer exactly such a variety of negotiation mechanisms. It has also been applied to a real business scenario faced by British Telecom, and so has been shown to work in practice. Finally, negotiations are decentralised (as they would be in a large distributed system) and agents operate across (but also within) organisational boundaries, a concept that is becoming central to Grid computing in the form of virtual organisations (as discussed in Section 2.1.2).

However, as a general framework or design metaphor, ADEPT does not directly address the problem of unreliable service providers, which is central to our work. Although failures are considered and providers monitored and penalised appropriately, service consumers do not anticipate failures proactively or use any form of trust measure. Within an organisation, where agents will generally honour their contracts, this is appropriate, but in the systems we consider, a purely reactive approach to failures will likely be insufficient.

This concludes our discussion of the agent-based techniques that we build upon in our work. Before discussing concrete techniques for provisioning workflows in the literature, we briefly summarise some results from the field of reliability engineering. This line of research has examined the construction of reliable systems from failure-prone components, and as such has tackled a similar problem to ours.

2.3 Reliability Engineering

In Section 2.1.3, we have already briefly discussed the use of redundancy to increase the reliability of task execution in a peer-to-peer system. This idea of using multiple services (or physical components) to decrease the overall probability of failure in a system has a long history in the field of reliability engineering. Work in this area has typically been concerned with selecting an appropriate number of redundant components to build a system with maximum reliability, given a set of resource constraints. Usually, it is assumed that such systems consist of a series of connected stages and that a single failure in any stage results in the overall failure of the system. Hence, work in reliability engineering typically considers variations of the following optimisation problem (Tillman et al. (1977)):

$$\begin{array}{lll} \text{maximise} & R = f(n_1, n_2, \ldots, n_N) & \\ \text{subject to} & \sum_{j=1}^{N} c_{ij}(n_j) \leq \hat{c}_i & i \in \{1, 2, \ldots, r\} \\ & n_j \in \{0, 1, \ldots, m_j\} & \end{array} \tag{2.4}$$

where n_j denotes the number of redundant components introduced at stage j of the system (out of N stages), R is the overall reliability, given as a function f of all n_j, $c_{ij}(n_j)$ is the amount of a resource i (out of r resources) that is spent on using n_j redundant components at stage j, $\hat{c}_i$

is the overall amount of this resource available, and m_j is the maximum number of redundant components that can be introduced at stage j.

Most commonly, the reliability R is simply the product of the success probabilities of all stages, which again depend on the failure probability d of each component:

$$\begin{aligned} R &= f(n_1, n_2, \ldots, n_N) \\ &= \prod_{j=1}^{N} (1 - d^{n_j+1}) \end{aligned} \tag{2.5}$$

Generally these optimisation problems are difficult to solve optimally — in fact, the above formulation of the problem has been shown to be NP-hard by Chern (1992). They are usually solved by finding an equivalent integer linear programming formulation and using established techniques for these (Tillman and Liittschwager (1967); Mizukami (1968); Ghare and Taylor (1969)) or by employing fast heuristics (Gopal et al. (1978); Kuo (2000); Liang and Smith (2004)).

While this work on reliability engineering was originally applied in the manufacture of physical devices, the idea of using redundancy to deal with failures has also been adopted by software engineers in the form of n-version programming or similar approaches (Scott et al. (1987); Lyu and He (1993); Avižienis (1995)). Here, critical software functionality is implemented several times independently by a number of developers and then executed in parallel. If one version fails or provides incorrect results, a voting mechanism is used to obtain the correct results from the remaining versions. Huhns et al. (2003) describe how several autonomous software agents can use similar mechanisms to cooperate in solving a common task and thus perform better and more reliably than they could if solving the problem in isolation.

Redundancy has also been applied directly to the problem of offering more reliable services in a distributed system. In particular, traditional Web servers often employ redundancy to seamlessly mask failed components (service failover) and to distribute requests to several replicated servers to balance the load on each one (Ingham et al. (1999); Aghdaie and Tamir (2003)). Similarly, the use of redundancy has been suggested to build fault-tolerant Web services (Keidl et al. (2003); Li et al. (2005); Merideth et al. (2005)). However, most of this work concentrates on the required infrastructure to build such robust systems. In work that is more closely related to the problem addressed in this thesis, Huang et al. (2006) suggest collecting several unreliable, but functionally equivalent services as part of a larger and more robust "service pool". When a consumer requests a service corresponding to the functionality offered by the pool, each of its member services is invoked sequentially in a certain order until one of them returns successfully. In this context, the authors present an algorithm for building such service pools, in order to meet some given minimum reliability while minimising the overall invocation time.

We believe that redundancy is a vital technique for addressing unreliability in distributed systems, and the widespread availability of many independent services makes this a feasible option.

However, none of the approaches discussed here is directly applicable when building a service-consuming agent. This is because most tackle the problem from the provider's perspective and are concerned with building a closed system that requires some initial, fixed investment in order to achieve a desired level of reliability, but whose components remain static during its lifetime. A service-consumer, on the other hand, is much more flexible as it may provision additional services at run-time only when required. Furthermore, most approaches concentrate on minimising the cost or maximising the reliability of a system given some constraints (as shown in Equations 2.4 and 2.5). However, it may not be obvious how such constraints should be chosen and which quality should be optimised, especially when the consumer seeks to balance the overall reliability with the associated cost (e.g., it may be happy to pay $100 for a workflow that is 90% likely to succeed, but would also pay $50 for one with a success probability of only 75%).

Nevertheless, we will use ideas from reliability engineering in our work and show how redundant provisioning of services can be used to proactively address service failures in a distributed system (Requirement A.2.b).

This concludes our discussion of the basic frameworks and technologies which our work builds upon. We have summarised several key infrastructures that are emerging in the context of service-oriented computing (including Web services, Grid computing, peer-to-peer systems and the Semantic Web), we outlined the key technologies that agent-based computing contributes to our research, and we briefly looked at work in the area of reliability engineering. In the final part of this chapter, we will now look at particular approaches that a single agent can employ to execute its workflows in the distributed systems we have discussed so far.

2.4 Executing Service Workflows

Throughout Sections 2.1–2.3, we have concentrated on the main enabling technologies that form the background of our work. However, as outlined in Chapter 1, our research is primarily concerned with *building a computational agent that is capable of executing complex workflows in highly dynamic and uncertain service-oriented environments*. To this end, we now look at current approaches for doing exactly that and evaluate their respective merits in relation to our requirements. Specifically, in Section 2.4.1, we begin by looking at current solutions for executing workflows in Grid and Web service environments, which have already been deployed successfully in distributed systems, but often require a substantial manual effort. Then, in Sections 2.4.2 and 2.4.3, we examine emerging research that aims to automate the execution of workflows.

Before proceeding, we briefly elaborate the concept of a workflow, which we introduced in Section 1.2. Essentially, *a workflow is a set of tasks and their interdependencies, which collectively achieve some business objective* (Hollingsworth (1995); Georgakopoulos et al. (1995); van der Aalst (1998)). Tasks usually represent atomic activities that contribute to the overall objective

and may generate and consume resources (including data). These tasks may have intricate interdependencies, which dictate how data is passed between them, and in which order they are executed. As an example, Figure 2.4 shows a workflow from a domain that we are particularly well acquainted with.

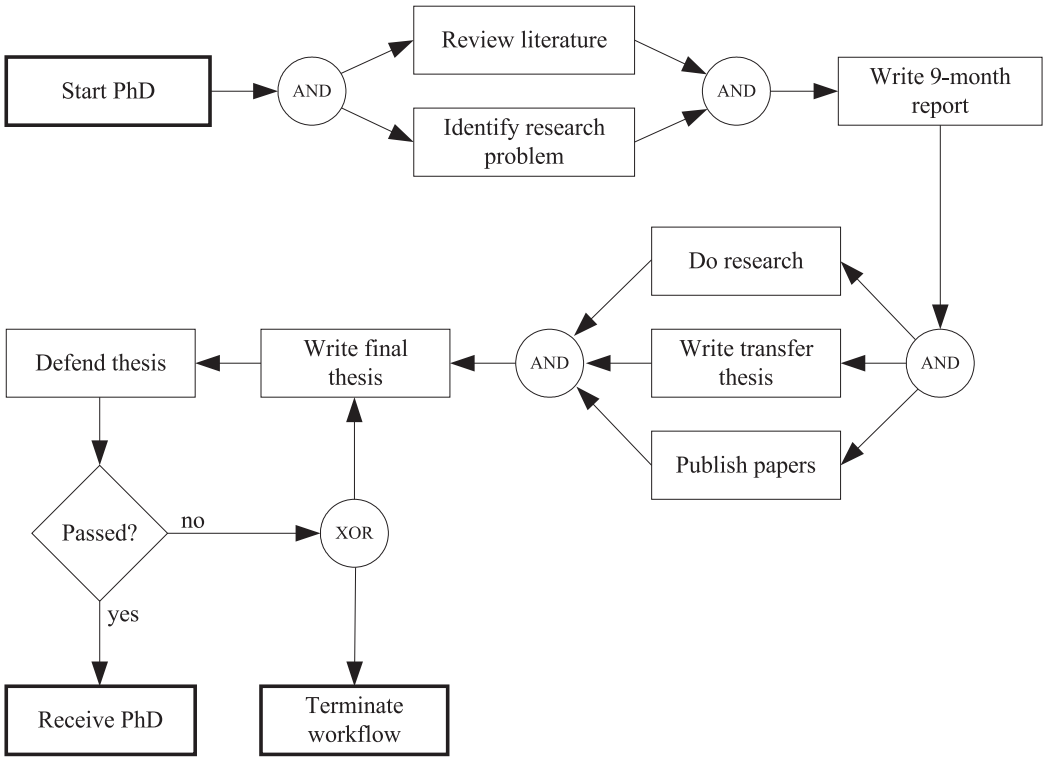


FIGURE 2.4: Example workflow summarising the tasks faced by a PhD student.

Workflows can be expressed in a variety of languages and formalisms, but these often offer similar constructs for expressing tasks and their dependencies. In this context, van der Aalst et al. (2003) describe twenty workflow patterns, which they believe cover most scenarios faced by automated workflow management software. Our example includes the most common of these in the form of task sequences (e.g., writing the thesis is followed by its defence), parallel tasks (e.g., the literature review is carried out in parallel with the problem definition) and alternative branches (e.g., the choice to re-write the thesis or to abandon the workflow).

In the context of distributed systems, workflows are a natural way to express how services can be engaged in order to achieve some goal. For example, for a scientific Grid application, a workflow may contain different data acquisition and manipulation services that perform complex calculations on behalf of the scientist (we discuss a detailed example in Section 3.5). In a business scenario, a workflow may encapsulate the process of satisfying a large order from a customer, which relies on services from the company's warehouse, billing department and possibly from external companies (e.g., for logistics, credit services and insurance). In practice, many current approaches use statically defined workflows that serve as a general template for specific objectives and are then instantiated at run-time.

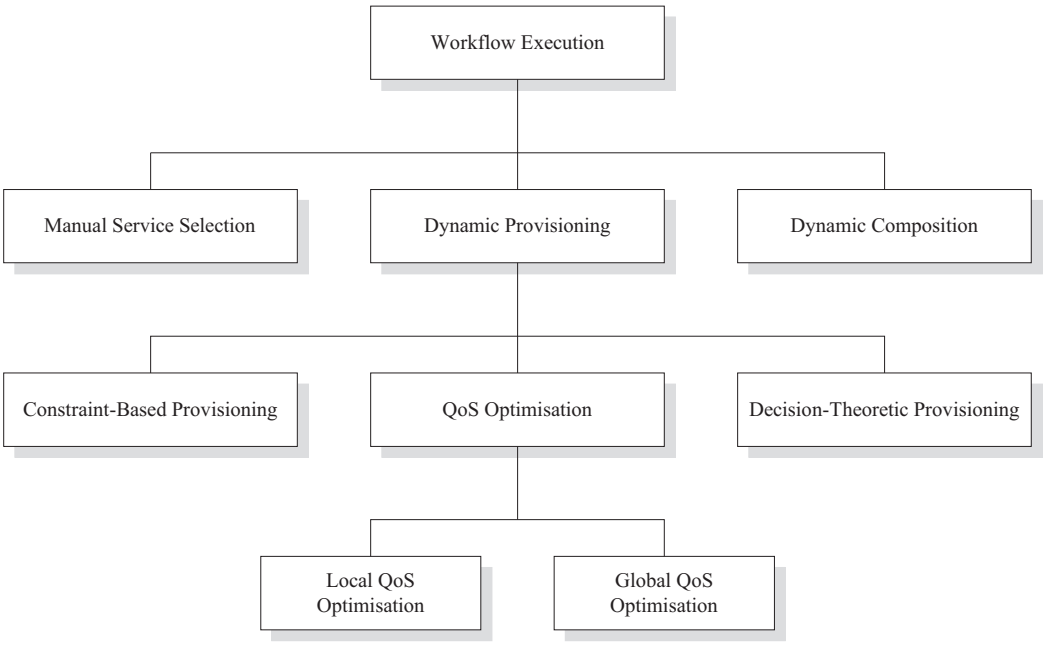


FIGURE 2.5: Current approaches for the execution of service workflows.

Against this background, one of the most prominent workflow description languages in the domain of Web services is the Web Services Business Process Execution Language (WS-BPEL) (Curbera et al. (2003); Weerawarana et al. (2005)). Here, a workflow consists of interactions with Web services and describes the control flow (including basic sequential, parallel and conditional execution) as well as the data flow between services. WS-BPEL builds directly on top of the Web service standards described in Section 2.1.1 and represents individual services using their WSDL interfaces. By using such interfaces, rather than references to concrete service instances, WS-BPEL provides some flexibility for the dynamic selection, or provisioning, of matching services at run-time. However, as we will see in Section 2.4.1, this is rarely exploited in practice.

In the following, we consider several current techniques for executing service workflows in distributed systems. To this end, Figure 2.5 shows a basic taxonomy of this work. Here, we distinguish between three principal approaches that are prevalent in the literature: *manual service selection*, where workflows and services are selected by hand, *dynamic composition*, where complete workflows are synthesised at run-time using high-level goal descriptions, and *dynamic provisioning*, where abstract workflows are instantiated by concrete services automatically at run-time. We discuss each of these below, justify why the former two are not suitable for the systems we consider and so describe work in the area of dynamic provisioning in particular detail.

2.4.1 Manual Service Selection

Although Web services and the use of WS-BPEL in particular are gaining in popularity with a wealth of commercial products now available, many current applications do not select services dynamically, as is envisaged by the research literature in service-oriented computing. Rather, human programmers specify manually not only the high-level workflows of applications, but also bind them to concrete Web services at design time (Zimmermann et al. (2003); Pautasso and Alonso (2005)). For this reason, most contemporary WS-BPEL development tools and execution engines do not directly support the dynamic discovery and selection of services at run-time. For example, IBM's WebSphere Integration Developer[13] allows users to build WS-BPEL workflows, but requires all abstract WSDL interfaces to be manually bound to specific services before they can be executed. The same applies to other WS-BPEL workflow engines, such as Oracle's BPEL Process Manager[14], the open source ActiveBPEL engine[15] or Apache ODE[16].

Similarly, workflows are usually executed in a naïve manner — the execution engine simply ensures that services are invoked in the correct order without addressing their reliability or availability. In effect, the human workflow designer is assumed to have already chosen the most suitable and reliable services. Hence, WS-BPEL offers no facility for proactively addressing service failures, nor does it in any way consider most of the features that characterise service providers in distributed systems (such as heterogeneity, the need for remuneration, negotiation and dynamic availability).

Despite these shortcomings, it does offer facilities for reactively handling failures (Requirement A.2.a). These are based on traditional exception handling mechanisms that follow pre-defined procedures for mitigating or correcting a problem before continuing the workflow (forward recovery), or that roll-back previous tasks of the workflow to terminate it in a consistent state (backward recovery) (Garcia-Molina and Salem (1987); Eder and Liebhart (1995); Casati et al. (1999)). The latter approach is supported by transaction mechanisms for Web services, and, in particular, the WS-Transaction specification (Curbera et al. (2003)), which explicitly provides mechanisms for cancelling (undoing) previous tasks when failures occur.

To support both forward and backward recovery, WS-BPEL allows workflow designers to specify fault and compensation handlers that are invoked when failures occur during workflow execution. However, we believe that neither is a satisfactory approach for addressing failures and uncertainty. More specifically, supporting transactions requires the provider to surrender some of its autonomy to the consumer by giving it the option to retrospectively relinquish any commitments. This may result in losses to the provider if it has already started processing a task, and so we believe that most service providers will be reluctant to offer facilities for rolling-back

[13] `http://www.ibm.com/software/integration/wid/`
[14] `http://www.oracle.com/technology/bpel/`
[15] `http://www.active-endpoints.com/active-bpel-engine-overview.htm`
[16] `http://ode.apache.org/`

tasks. Forward recovery may be a more viable option (e.g., by substituting providers at runtime), but fault handlers are typically specified manually and so require a human designer to predict failures and hard-code possible solutions in advance.

A more flexible solution to this is proposed by Zeng et al. (2005) and Erradi et al. (2006), who advocate the use of generic exception policies. These specify general procedures that the workflow engine should carry out when certain failure conditions are encountered during execution. For example, these procedures might include re-trying the same service several times, switching to other (possibly redundant) services or terminating the workflow. However, they still need to be manually specified and do not proactively address services failures.

In summary, WS-BPEL relies heavily on human effort and thus exemplifies a common trend in a number of widely used workflow languages and execution engines for service-oriented systems. These include, for example, the Java CoG kit (Laszewski and Hategan (2005)) that is part of the Globus toolkit, and Taverna, which is a workflow engine specifically developed for enabling the workflows that bioinformaticians face (Oinn et al. (2004)).

This concludes our review of workflow execution approaches that rely on the manual specification of services. We have seen that these are currently being employed in commercial and academic environments, and that software supporting them is readily available today. They already offer expressive mechanisms (most notably WS-BPEL) to describe complex workflows, meeting our Requirement W.1. However, they generally assume highly reliable, or at least cooperative and benevolent, services. Each item in the workflow is handled by a single service that is pre-defined by a human programmer, and when failures occur, these are treated as exceptions that are handled by manually coded procedures. Some systems retry different providers upon failure, but this is purely reactive and without regard for the potential costs that might be incurred. As such approaches are clearly insufficient for the environments we consider (services are unreliable and require remuneration), we look at the current state of research in the area of dynamic service composition in the following section.

2.4.2 Dynamic Service Composition

While the work discussed so far relies heavily on human effort, the field of dynamic service composition represents the other extreme. Specifically, research in this area is concerned with synthesising entirely new workflows by composing atomic services to achieve some overall goals (McIlraith et al. (2001); Srivastava and Koehler (2003)). It typically applies AI planning techniques, which take an initial state, a goal state and a set of operators and then search for a sequence of operator applications that will result in the desired goal state (Ghallab et al. (2004)). Now, the operators in this case are service descriptions using such formalisms as OWL-S (which already contains constructs commonly used in planning) and the result of the composition is a

workflow consisting of concrete service invocations. Hence, such an approach would take a considerable burden from human users who no longer need to be concerned with the construction of workflows, but rather state their intentions as simple high-level goals.

As planning is difficult in open environments such as the Internet, where the consumer agent does not have complete knowledge of the domain and where such knowledge has to be actively gathered during planning and execution, service composition is still very much an open research problem. Example approaches include the work by McIlraith and Son (2002), who use logic programming to compose services. McDermott (2002) and Klusch et al. (2005) adapt existing planners and the widely used Planning Domain Definition Language (PDDL) (Edelkamp and Hoffmann (2003)) to Web service scenarios, and Wu et al. (2003) concentrate specifically on composition approaches that use common Semantic Web technologies such as OWL-S and OWL.

These composition approaches are very flexible, because they do not rely on static workflows. Hence, they deal naturally with the dynamism of an open system (Requirement M.5), as they use only the services that are available at a given time. However, current composition techniques select services implicitly as part of their planning algorithm and will therefore generally pick the first service that helps fulfil the goal. For this reason, these approaches do not proactively address potentially unreliable or even malicious services (Requirement A.2.b), but rather assume that service providers publish truthful descriptions that are always adhered to. Again, failures are assumed exceptional and usually solved by expensive replanning (Klusch et al. (2005)). Overall, service composition is unlikely to scale to larger systems (Requirement A.3) due to the inherent complexity of planning (Bylander (1994); Erol et al. (1995)). This is a particularly pressing concern for systems where there might be hundreds or thousands of competing providers.

Hence, we believe that neither completely manual workflow execution nor the automatic composition of new workflows are realistic approaches in the uncertain and dynamic systems we consider. Due to this reason, we now turn our attention towards work in the area of dynamic service provisioning.

2.4.3 Dynamic Service Provisioning

To address the complexity inherent in fully automatic service composition and to overcome the restrictions of manually specified workflows, some research has suggested the use of abstract workflows, which are dynamically instantiated, or provisioned, at run-time (McIlraith and Son (2002); Mandell and McIlraith (2003); Sirin et al. (2005)). This work assumes that workflows for particular objectives usually follow the same basic steps, even if the choice of service instances is different each time, depending on the user's personal constraints and current service availability. More specifically, such abstract workflows usually include a number of semantically annotated abstract tasks (e.g., using generic OWL-S descriptions). At run-time, these abstract

task descriptions are used to automatically discover service instances, which can then be provisioned for the tasks of the workflow. If necessary, additional planning is used to combine or substitute abstract workflow fragments (Sirin et al. (2005)) or to add intermediate services, e.g., to translate between heterogeneous data representations (Mandell and McIlraith (2003)).

We believe that this approach is promising, because it does not require the service consumer to plan from scratch, but still allows it significant flexibility to account for the changing availability of services. Furthermore, it is well suited for dealing with uncertainty and failures, as portions of a workflow can easily be re-provisioned when necessary without the need for expensive re-planning. Finally, it allows us to take into account and reason explicitly about the non-functional characteristics of services, such as their cost or reliability, and use this to guide the agent's decision-making.

Hence, we concentrate on this process of dynamically provisioning services for an abstract workflow in our work. As described above and in Section 1.5, we refer to service provisioning as *the selection of particular service instances for specific tasks of a workflow*. It should be noted that, unlike Jennings et al. (1996), we do not necessarily equate provisioning here with advance negotiation. Rather, a service consumer may provision services on demand when they are required, for example by invoking a Web service. Similarly, the consumer might provision a service tacitly in advance, but defer its negotiation to a later time.

Against this background, we now look at existing work in this area. This typically assumes that a service consumer has already discovered a set of potential services for each task using, for example, service registries such as UDDI or by reasoning over semantic service descriptions (and this will also be one of the assumptions we make in our own work). We continue to follow the taxonomy shown in Figure 2.5 and begin by examining constraint-based provisioning approaches.

2.4.3.1 Constraint-Based Service Provisioning

The first approach we discuss uses decision rules to filter appropriate services. Here, the user specifies additional constraints on the services it requires, which are then used to differentiate between multiple offerings. These constraints are often based on non-functional information about the services and usually make binary decisions whether to accept each service or not. When more than one service matches, a random choice is made or some tie-breaking rule is applied.

As an example of this approach, Keidl et al. (2003) define two types of constraints for each task — *preferences* and *conditions*. Both are logical conditions on the meta-data of potential services and may contain disjunctive or conjunctive constraints (for example, a constraint might be that the service must be offered by a company based in the UK and use a particular encryption standard). Here, the conditions define which services are eligible and the preferences are softer constraints that are applied when more than one service satisfies the conditions. Patel

et al. (2004) use a similar approach, but express constraints as rules, which can be re-used and composed for different tasks. For example, there might be rules that define what is considered a *cheap* service (say, $cost < £10$) and what is a *reliable* service ($reliability > 0.95$). A particular task might then require a cheap *and* reliable service.

To give another example of provisioning with constraints, Mandell and McIlraith (2003) present an interesting approach that extends BPEL4WS, the predecessor of WS-BPEL, to dynamically select services based on DAML-S profile descriptions. Here, a theorem prover is employed to discover concrete services that satisfy the DAML-S profiles given in their modified BPEL4WS workflows. Due to its reasoning capabilities, the system is also able to add appropriate mapping services if required (e.g., to convert currencies or different date representations) and handles additional user constraints on the semantic service profiles (e.g., that only UK-based services should be used or that the end result must be in a particular format).

Now, the problem with most constraint-based approaches is that they simply narrow down the choice of appropriate services based on local (task-specific) binary decisions. They do not dynamically and rationally choose appropriate thresholds for these rules, but rather depend on a human programmer to make this decision. Furthermore, they are very rigid — while a programmer might introduce a rule that all services should be highly reliable in order to address uncertainty, this may simply result in unsatisfiable workflows where no services are sufficiently reliable. In practice, it is necessary to strike a balance between different service parameters rather than set hard limits for all instances. For example, some tasks may be inherently difficult to achieve and so the consumer must accept some unreliability, but may be able to balance this by choosing only highly reliable services for other tasks. Similarly, a marginally more reliable service might be substantially more expensive than other services, but it is difficult to write appropriate rules that make the best decisions in such scenarios.

In the next section, we will look at some approaches that have considered these issues in more detail and proposed provisioning techniques based on comparing and ranking services using their performance criteria.

2.4.3.2 Quality-of-Service Optimisation

A large body of research has considered the provisioning of services based not only on hard constraints, but also on preferences for different QoS characteristics. As shown in Figure 2.5, we distinguish here between those approaches that examine and provision tasks in isolation and those that consider the impact of each service on the whole workflow. We discuss these local and global techniques separately in the following.

In their work on using Semantic Web technologies to instantiate abstract workflows, Sirin et al. (2005) consider the case when many services match a given task. Rather than imposing hard constraints that might result in no or too many matches, they assume preferential independence between these parameters and then pick a service that is Pareto optimal regarding all dimensions

(i.e., it selects a service, such that there is no other service that offers a better performance along one dimension without also offering worse performance along another). This guarantees that there is always a matching service and it removes clearly inferior candidates. However, it may also easily result in arbitrary decisions. For example, if we assume that there are three services, as shown in Table 2.1, all of these are Pareto optimal (but the consumer is probably best advised to choose the second one).

Service	**Cost**	**Reliability**
A	\$1	1%
B	\$1.5	99%
C	\$100	99.1%

TABLE 2.1: Example services

In the context of Grid computing, Condor and the related Condor-G use simple ranking schemes along with constraints for the local provisioning of workflow tasks. Condor is a framework for allowing consumers to execute computational jobs over a distributed system of workstations (Frey et al. (2001); Thain et al. (2003)). When submitting jobs, consumers specify the types of resources they need by giving both a set of requirements and a ranking expression. These are similar to the conditions and preferences described by Keidl et al. (2003), but the ranking expression might simply be a non-functional property to be maximised. For example, the consumer might require a Unix-based system with a memory of least 512 MB RAM, but when several resources satisfy this, it will select the one with the highest processor speed.

Now, Condor is particularly interesting, because it also offers some powerful techniques for tolerating failures. As its primary objective was to harness the computational resources of idle workstations, it closely monitors all submitted jobs and regularly saves their progress. When a job is interrupted (usually when a user reclaims the computer), Condor re-distributes it to a different workstation, where it is then continued. While this job migration is an effective method for addressing service failures reactively, Condor relies on a largely cooperative environment (initially, it was deployed for use within one particular organisation). As such, service consumers are expected to report accurately on their progress and provide intermediate results. Similarly, there is no explicit notion of costs or remuneration and service consumers can simply retry until their job succeeds.

While the work so far has looked at each task in isolation, other research has attempted to consider the impact of provisioning on the overall workflow. This is important, because provisioning a single unreliable service may jeopardise the whole workflow and keeping to an overall budget may be more important than controlling the expenditure on each single task. To enable this, a number of QoS aggregation mechanisms have been proposed, in order to calculate overall performance metrics for workflows, based on the services selected for each task (Cardoso et al. (2004); Jaeger et al. (2004)). These are typically simple calculations — for example, the overall cost of a workflow is the sum of all service costs and its reliability is the overall product of all service reliability values.

The goal of a large body of research has been to optimise these aggregated QoS values while satisfying some overall constraints, such as a deadline or fixed budget (Zeng et al. (2004); Aggarwal et al. (2004); Canfora et al. (2005); Xiao and Boutaba (2005)). As an example, consider the simple workflow given in Figure 2.6. It is assumed that the agent needs to complete all four tasks in the order indicated (t_2 and t_3 can be invoked in parallel). Table 2.2 contains a list of services suitable for each task. Now, at the simplest level, a QoS-based agent might provision providers in order to optimise one of the criteria. As in Xiao and Boutaba (2005), we may be interested in minimising the overall cost while ensuring the overall duration is less than 100. In this case, an optimal solution is to provision s_{11}, s_{21}, s_{31} and s_{41} for a total cost of 41, a combined reliability of 0.27 and a duration of 30. A similar approach of optimising one particular performance measure is taken in the work of Deelman et al. (2003b), who use planning techniques to minimise the overall time of large Grid workflows.

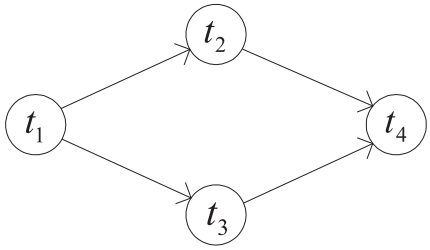


FIGURE 2.6: Simple workflow consisting of four tasks.

Task	Service	Cost	Reliability	Duration
t_1	s_{11}	5	0.5	10
	s_{12}	10	0.9	5
t_2	s_{21}	1	0.75	5
	s_{22}	7	1	30
t_3	s_{31}	10	0.9	10
t_4	s_{41}	25	0.8	10
	s_{42}	10	0.99	100

TABLE 2.2: Suitable services for each task

Now, in realistic scenarios, a consumer will be unlikely to optimise only along one of the QoS dimensions but will rather want to find a good balance. Hence, it is common to optimise a weighted sum of all performance measures (Gu and Nahrstedt (2002); Zeng et al. (2004); Canfora et al. (2005); Yu and Lin (2005); Ardagna and Pernici (2007); Jaeger and Mühl (2007)). This is typically done by first normalising each of the aggregated measures, for example to range between 1 and 0 (indicating the best and worst values possible respectively). Then a weight $\breve{w}_i$ is attached to each measure with $\sum_i \breve{w}_i = 1$, and the overall weighted sum is optimised. In other words, if ρ is vector of provisioned services for all tasks (e.g., $\rho = [s_{11}, s_{22}, s_{31}, s_{41}]$) and $q_i(\rho)$ is the ith aggregated QoS value resulting from ρ (e.g., the overall duration or reliability), then a QoS-based agent will provision services by solving the following optimisation problem

(where $\hat{Q}_i$ is a constraint for the ith quality):

$$\begin{array}{lll} \text{maximise} & Q = \sum_i \check{w}_i q_i(\rho) & \\ \text{subject to} & q_i(\rho) \geq \hat{Q}_i & \text{for all } i \end{array} \tag{2.6}$$

To continue the example above, if we weigh all performance measures equally, then the optimal solution becomes $[s_{12}, s_{21}, s_{31}, s_{41}]$ with cost 46, reliability 0.486 and duration 25. Unfortunately, such a problem cannot be solved efficiently in general, as we can reduce the 0/1 knapsack problem to it and so prove that it is NP-hard. In practice, integer linear programming applications are usually employed to solve the QoS problem (Zeng et al. (2004); Aggarwal et al. (2004); Yu and Lin (2005)), although other approaches, such as genetic algorithms, are occasionally used for complex scenarios (Canfora et al. (2005); Jaeger and Mühl (2007)).

In order to address services failures and execution uncertainty in these QoS-based provisioning approaches, Zeng et al. (2004) as well as Canfora et al. (2005) suggest adaptive replanning mechanisms. These monitor the execution of a provisioned workflow by constantly checking the progress of services and calculating their impact on the overall aggregated QoS values. When these breach the overall constraints, the remainder of the workflow is re-provisioned to again satisfy the constraints. For example, a provisioned service may take much longer than expected and thereby lead to a breach of the workflow deadline. In this case, the consumer re-provisions the remaining tasks, using faster services where possible.

While this replanning is purely reactive, Jaeger and Ladner (2005) use the concept of redundancy to improve provisioned workflows before execution. Assuming that a workflow has already been fully provisioned, they show how the addition of redundant services can improve overall qualities such as the reliability or maximum duration of the workflow. However, their approach has several weaknesses. Although the authors suggest that redundancy should be added to particularly weak parts of the workflow, they do not discuss the decision-making procedures necessary to decide which and how many services to add. Furthermore, their approach retains all initially provisioned services and so it does not consider the case where several cheap, unreliable service may offer a better overall quality than the original service. Finally, it uses questionable aggregation methods that are difficult to justify in practice. For example, Table 2.3 shows how their method combines two example services invoked in parallel — the new reliability is the probability that at least one service is successful and the new maximum duration is the smaller duration of the two services. Clearly, this method is unrealistic, resulting in aggregated values that overestimate the performance of the services.

Service	**Reliability**	**Maximum Duration**
A	1%	1s
B	99%	100s
A and B	99.01%	1s

TABLE 2.3: Example of redundantly provisioned services (Jaeger and Ladner (2005)).

In conclusion, these QoS-based approaches show some promise. They acknowledge some uncertainty about provider behaviour in the form of reliability measures (Requirement M.1.a), but usually assume certainty about service durations and other quality measures. By modelling each service explicitly with different quality values, these approaches address the inherent heterogeneity of such services (Requirement M.4). Service costs can also be included in the calculations (Requirement M.2.a), and most of the workflows considered are expressive (Requirement W.1), containing parallel, sequential and conditional branches (Cardoso et al. (2004)). The reward models are simple as they rely on linear combinations of QoS values, but they take into account global solution qualities such as the workflow completion time, can be adjusted flexibly by altering the weight vector and may include complex performance constraints (Requirement W.2). Finally, QoS provisioning addresses service failures reactively by replanning (A.2.a) and proactively by taking into consideration an overall reliability measure (A.2.b). There is even some initial work on including redundancy.

Despite making some progress towards meeting our overall research requirements, we believe that QoS-based provisioning in its current form is not usable in the environments we consider. In particular, we note the following shortcomings:

- The weighted QoS function that is optimised is very simple and assumes that issues are linear, additive and independent. In particular, reliability is treated as just another issue that is substitutable, at a constant rate, with other qualities of the solution. Such behaviour is not rational (as defined in Section 2.2.1), will require careful manipulation of the appropriate constraints and weights for each workflow, and so largely defeats the purpose of designing an agent to automate the execution of workflows (our central research aim).
- The approach does not offer a good solution for generally highly unreliable services. While it can optimise the overall reliability, the workflow will still fail when all services in the system are unreliable or when the workflow is simply very long. For example, when it provisions services with a reliability of 99% each for a workflow consisting of 100 tasks, the overall success probability is just under 37%. Redundancy may help with this issue, but current work is insufficient for the reasons outlined above.
- Although some of the above approaches suggest reactive re-planning in case of failures, they do not reason about this in advance. This is a major shortcoming. For example, constant re-planning may be expensive if services demand some payment for each invocation, and so it may result in a large loss for the agent if it still fails to complete the workflow in time. On the other hand, if services are cheap and plentiful, the agent may succeed with a high likelihood despite a low initial reliability for the overall workflow. However, it would need to plan ahead and leave sufficient time in its schedule to attempt some tasks several times before its deadline.

Another potential criticism is that QoS approaches rely on information that is provided to them through service descriptions, which may be unreliable or even manipulated in order to

entice consumers to provision particular services over others. However, as we mentioned in Section 2.2.3, we believe that such issues are being addressed by trust and reputation mechanisms. In fact, work by Sreenath and Singh (2004) is concerned with collaborative service provisioning based on ratings by other agents, and in other work, Maximilien and Singh (2005) consider a trust mechanism for QoS-based service provisioning.

To conclude our review of current service provisioning techniques, we look at an area that we find particularly promising and which begins to consider some of the agent-based work presented in Section 2.2.

2.4.3.3 Decision-Theoretic Provisioning

So far, we have discussed approaches that provision services based on logical constraints and rules, or that optimise some numerical parameters of a possible solution. Despite the fact that services are envisaged to be used in economic contexts, that they will contribute significantly to the operations of organisations, and that they are consumed in uncertain and competitive environments, little work has used decision-theoretic principles for provisioning services for complex workflows.

An exception to this is the work carried out on top of MAGNET, which we introduced in Section 2.2.4. Collins et al. (2001) consider simple workflows of interdependent tasks, represented as directed acyclic graphs, that a consumer wishes to complete before a fixed deadline. Here, the consumer uses decision theory by assigning utilities to various outcomes of the workflow (e.g., not attempting the workflow at all, failure after the nth task or overall success), and then calculates the expected utility of the workflow by multiplying the utility of each outcome with its respective probability (as we discussed in Section 2.2.1). Because services cost money and the successful completion of a workflow is assumed to have an explicit monetary value to the customer, utilities are calculated directly from the loss or profit that each potential outcome entails[17].

Such utility calculations are used by the service consumer in two ways. First, they help the agent determine a good preliminary schedule for the workflow, which is then used to solicit bids from suppliers. Here, the agent might delay expensive tasks to reduce the probability that the workflow fails after these tasks have been started. Such delays have an impact on the expected utility, because leveled commitments allow the consumer to withdraw from a deal if the workflow fails before the task is started (Babanov et al. (2004)). At this stage, the agent also attempts to balance the need to create a tight schedule and finish within the deadline with an appropriate amount of flexibility to solicit the maximum number of bids of suppliers (Collins et al. (1999)). With this preliminary schedule, the consumer proceeds to organise a reverse combinatorial auction (as outlined in Section 2.2.4).

[17] This is common in decision theory — in fact, the monetary outcomes of gambles are often equated with utility measures for risk-neutral agents. When agents are risk-sensitive, a non-linear function is usually applied to translate between them (Raiffa (1968)).

The second application of utility calculations happens during the provisioning phase. This is when the consumer receives bids from the suppliers to carry out the requested tasks at the specified times. The consumer then chooses the set of bids that maximises the expected utility of its workflow. Due to the complexity of this task, the agent uses simulated annealing rather than an exhaustive search, which has the added benefit of being an anytime algorithm that can be stopped when a pre-defined time-limit is reached.

Relating this work back to our requirements, we note that, in addition to addressing most of the issues covered by QoS-based approaches, it offers a feasible approach towards implementing an agent that makes (boundedly) rational decisions (Requirement A.1). Furthermore, it demonstrates how a non-trivial negotiation protocol can be used to provision services in advance (Requirements M.2.b and M.3.b).

Nevertheless, the work is lacking in several areas:

- By relying only on advance negotiation, the work is not applicable to current service-oriented systems, where interactions are mostly carried out through one-to-one on-demand invocation mechanisms. Eventually, we envisage systems to offer varied forms of negotiations that coexist (Requirement M.3.b) rather than relying on one particular mechanism.
- Similarly, the service consumer provisions an entire workflow at once, which requires a high initial investment that may be lost when a single service fails. For this reason, it is also slow to respond to failures, as it has to organise a new auction for the remainder of the workflow. In dynamic environments, this is not desirable, especially when the agent has to work towards a fixed deadline. Also, the authors do not explicitly describe how such re-provisioning should proceed and how the agent might reason about it in advance.
- As in the QoS-based approach, workflows are still vulnerable in certain scenarios. Again, we can consider generally unreliable services that will serve as bottlenecks or extremely large workflows that pose a risk even if the individual providers are highly reliable.

This concludes our literature review of the basic technologies and existing techniques for workflow execution and service provisioning. In the final section of this chapter, we briefly summarise our main findings and evaluate the extent to which our requirements are met by the current literature.

2.5 Summary

As discussed in this chapter, the research community has devised many solid techniques for providing services over computer networks, for negotiating about the terms on which the services are provided and for making good decisions under uncertainty and with limited computational

resources. However, little work combines these in order to enable computational agents to execute complex workflows autonomously on behalf of their owner in uncertain and competitive environments — a key problem that is encountered in business and in academia alike.

Some existing work addresses parts of our requirements in isolation. Often, this will provide us with the necessary tools to tackle our overall research challenge, as is the case, for example, with the work on probabilistic trust models by Teacy et al. (2006). In other cases, existing solutions for some of our requirements may be infeasible when considering our overall aims, as, for example, the use of manually specified exception-handling routines in WS-BPEL demonstrates. Hence, in this section we conclude the literature review by briefly summarising our main conclusions and evaluating what existing work to build on.

2.5.1 Model Requirements

Current work in Web services, Grid computing and peer-to-peer systems offer infrastructures for providing and consuming services in distributed scenarios. By using common, platform-independent data formats and protocols, they allow heterogeneous agents to interact and, by employing emerging techniques from the Semantic Web, to discover each other. As such, these are the basic enabling technologies that we can build upon, but they do not model explicitly the uncertainty or dynamism that is inherent in the systems we consider.

Recent work on QoS in service-oriented computing models uncertainty using probabilistic measures, which can be obtained either by a centralised observer or through trust and reputation mechanisms. We believe that this is a promising approach for satisfying Requirement M.1 in order to represent the possibility that providers may defect or offer their services with varying qualities. Similarly, such measures can be used to distinguish between heterogeneous agents (Requirement M.4) and there is some work to consider the dynamism of service-oriented sytems by tracking changes in performance over time (Requirement M.5).

Currently, interaction mechanisms in service-oriented systems are simple and rely mostly on on-demand service invocation (Requirement M.3.a). Despite some initial work on the description of service level agreements, there are no satisfactory mechanisms for automated service remuneration and advance agreements (Requirements M.2.b and M.3.b). Such issues are addressed by separate work in agent-based negotiations, which offers a spectrum of negotiation protocols for different settings. In this context, we find the leveled commitment contract net protocol particularly promising, because it follows similar contracting models in the real world, is easily implemented and offers the contracting agent flexibility in choosing which offers to accept.

2.5.2 Workflow Requirements

As workflows have been widely studied in the context of business processes, there is extensive work on the expressivity of workflows and their semantics. We will draw on this work when designing the workflows that service consumers face (Requirement W.1).

However, because the workflows considered in that work are usually designed by humans, they do not provide associated reward models that might guide an autonomous agent. Work on the MAGNET system awards agents a fixed price for successful completion. This is a promising approach as it lends itself well to a decision-theoretic analysis, where cost, payoff and uncertainty are balanced. However, it is a very simple model that takes into account only the binary outcome of finishing in time or not. In reality, cumulative penalties might apply to late completion (Requirement W.2). Nevertheless, the approach allows a user to flexibly determine the value of a workflow and can be extended to cover more expressive reward functions.

2.5.3 Agent Requirements

Service failures are addressed in the literature in several ways. Most workflow engines contain explicitly specified failure-handling routines that deal with failures reactively (Requirement A.2.a). Such an approach requires a human programmer to foresee problems and is therefore not applicable for our work. In Condor and most peer-to-peer systems, failures are addressed in a more appropriate manner by automatically choosing substitute services. Similarly, some approaches use global workflow replanning to react not only to failures, but also to other events that breach workflow constraints (e.g., when services take longer than expected). Hence, these approaches are adaptive, but only react when breaches have already occurred and do not exploit opportunities (Requirement A.4). Furthermore, this reactive behaviour could be infeasible in unreliable environments with workflow deadlines, where the consumer is under time-pressure and cannot retry indefinitely.

Some approaches take a more proactive approach towards dealing with failures (Requirement A.2.b). Especially, work in decision-theoretic provisioning explicitly models reliability and strikes a balance between choosing more reliable services and the associated costs. In P2P systems and the deployment of Web services, techniques from reliability engineering are used and service redundancy is exploited to provide higher overall reliability, but this is often determined manually or formulates the redundancy allocation as a static optimisation problem with given cost or reliability constraints.

Work in decision theory offers some valuable tools for making good decisions within the limits of a computationally bounded agent (Requirement A.1). Some of these results have already been applied to a particular provisioning scenario and show some promise for our work. However, much work on service provisioning aims to solve combinatorial problems, which are generally

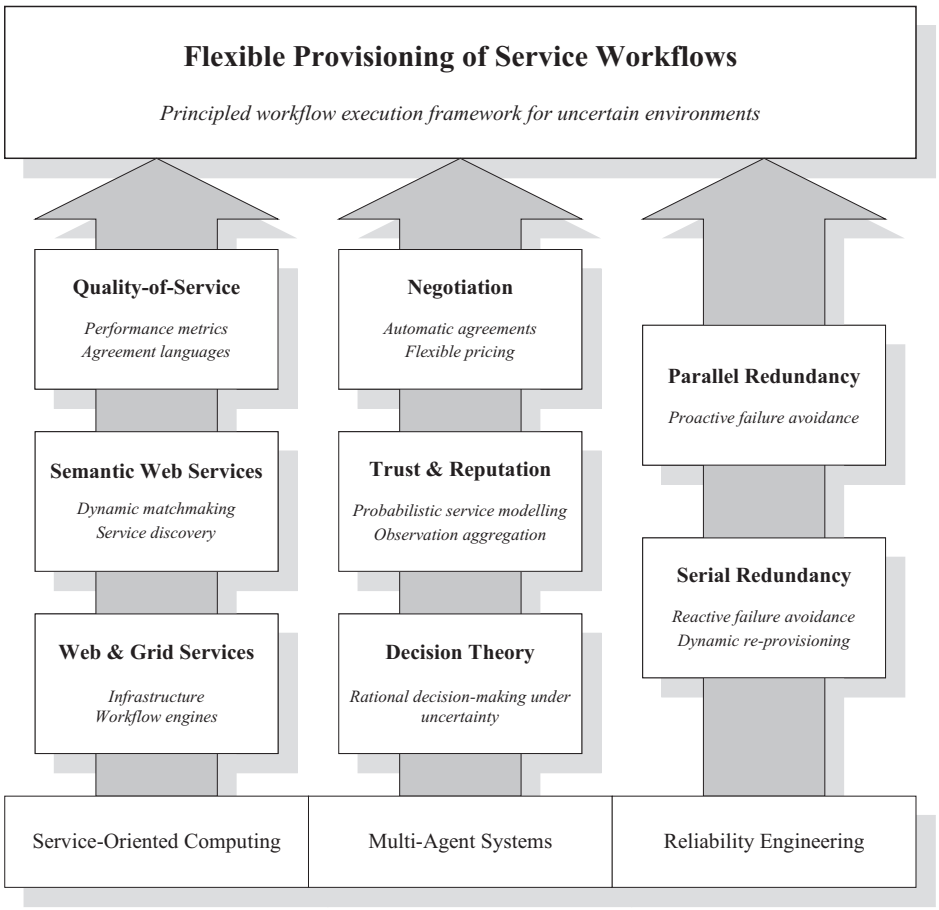


FIGURE 2.7: Summary of the work we build upon in this thesis.

intractable for large cases. Approaches that use heuristic methods seem most promising here to satisfy our requirement for scalability (Requirement A.3).

In summary, many of our requirements have been considered in isolation or in the context of different research. Hence, there are a number of tools that we can draw upon for our research problem. However, there is currently no effective general strategy for provisioning services in realistic distributed environments, where services are neither provided for free nor behave in a reliable manner. To address this, we build on the work presented in this chapter (summarised in Figure 2.7) and first outline an abstract model of a service-oriented system in the next chapter. Then, we consider a range of service-oriented environments, where varying amounts of service performance information is known to the consumer and where different negotiation mechanisms are used. In order to address these separately, exploiting the specific characteristics of each environment, we develop several novel service provisioning strategies in Chapters 4–6.

Chapter 3

Modelling a Service-Oriented System

To frame the remainder of this thesis, we begin by describing in more detail the systems we consider, based both on our original requirements given in Chapter 1 and on current service-oriented technologies outlined in Chapter 2. The purpose of this discussion is to introduce a number of common assumptions that are used throughout the thesis, and to provide a high-level description of a service consuming agent and its possible interactions with service providers. This will form a general system model, which we extend and base our work on in later chapters.

More specifically, we begin in Section 3.1 by defining the basic terminology of our model, and in Section 3.2, we describe the lifecycle and structure of a workflow. This is followed by an outline of how service providers behave and the information that is available about them in Section 3.3. We give a high-level algorithm that formalises the behaviour of a service consuming agent (Section 3.4), and we briefly introduce an illustrative workflow from the bioinformatics domain that will serve as a running example throughout the thesis (Section 3.5). Finally, we conclude our framework by discussing some of its limitations in Section 3.6.

3.1 Basic Terminology

Our model describes a distributed, service-oriented environment, where actors can exhibit a varying degree of reliability, timeliness and autonomy in providing and consuming services. It assumes several basic concepts (shown in Figure 3.1):

- All participants in service-oriented systems are autonomous **agents**, i.e., self-interested entities that seek to maximise their private utility (Jennings (2000)). We distinguish between two different types of agents:
  - **Providers** offer their capabilities to other agents in the system, usually in exchange for financial remuneration.

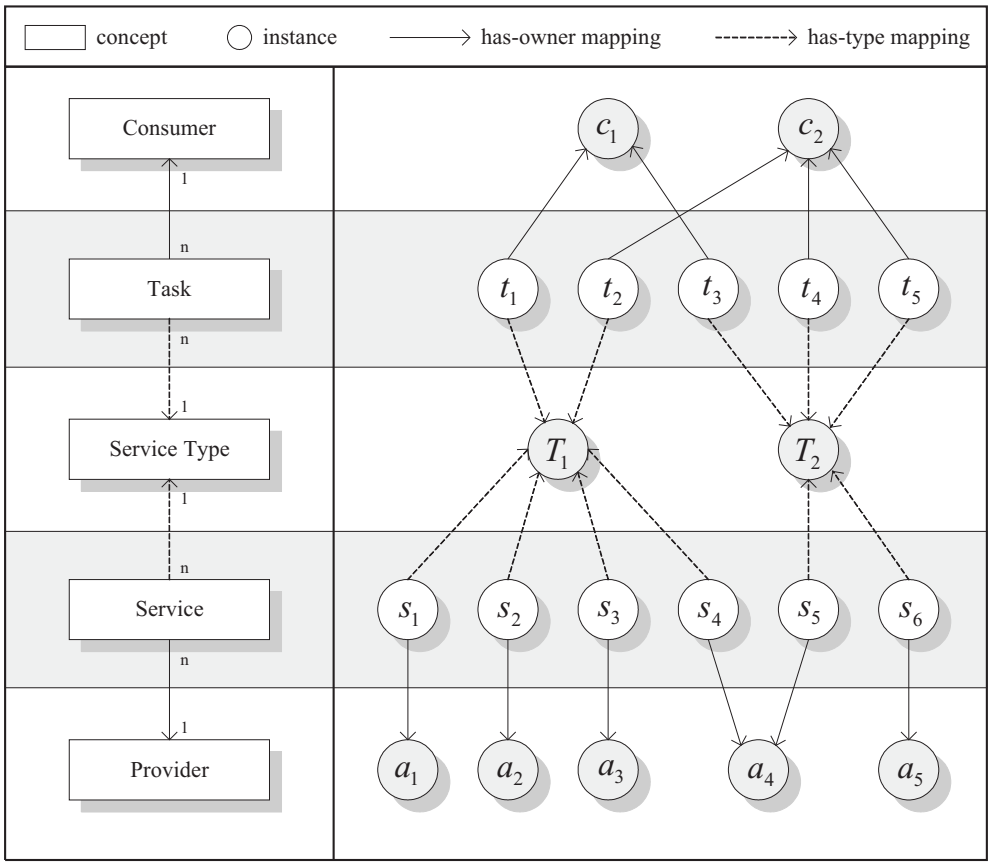


FIGURE 3.1: Basic model concepts (on left) with an example (on right). Arrows indicate functional (many-to-one) relationships.

  - **Consumers** make use of the capabilities offered by providers in order to achieve their goals.

- **Tasks** are problem instances that a consumer faces in a particular context. Such a task can be seen as a desired change in the current state of the agent's environment or in its current knowledge. Generally, we will concentrate on tasks that are solved by the transfer of data or information — for example, such tasks might include compressing a large dataset, finding the solution to a complex optimisation problem, or comparing a nucleotide sequence to a database comprising millions of genes. To a lesser extent, our work also applies to tasks that require actions with a tangible physical effect on the world, such as the delivery, manufacturing or processing of goods, but we are not concerned with the associated logistic problems and assume free disposal of unwanted goods (this assumption is explained in more detail in Section 3.6).

  Furthermore, tasks are not further decomposable by their owners and have to be delegated to providers that are able to solve them. Due to this task delegation, we assume that, from the owner's perspective, tasks are always in one of two states — completed or uncompleted. We also assume that there is only one transition from being uncompleted to

completed when a task is solved, after which it becomes irrelevant to the owner's current needs.

- Each task is associated with a **service type**. This is an abstract description of the type of service that is required to solve the task. Different tasks can be associated with the same service type, but each task has exactly one type. For example, the task of searching for a specific nucleotide sequence in a genome database could be associated with an abstract data comparison service type.

- **Services** (or **service instances**) are concrete implementations of a given service type. There may be many implementations of a service type, but each service has exactly one type. These services are behaviours that service-providing agents offer to consumers in order to help them solve tasks of the appropriate type. In this work, such services are treated as atomic problem solving units, whose internal realisations are considered as black boxes and not further considered in this work[1]. Furthermore, they are generic and repeatable, that is, they can be procured by different consumers for different tasks, but each service instance has exactly one provider. An example of a service could be a particular implementation of a genome comparison algorithm offered as a Grid service by a biological research laboratory (O'Brien et al. (2004)).

The above concepts form the basic terminology of our model. As we are interested in developing strategies for a single service consumer, we now describe in more detail the behaviour of such a consumer and the workflows it executes.

3.2 Workflow Model

Service consumers in distributed systems often face multiple inter-dependent tasks, which together achieve a more complex objective. For example, several data processing tasks may be required to sequence and analyse a gene (see Section 3.5), or different parts and materials may need to be purchased in a procurement scenario. As described in Chapter 2, these tasks and their dependencies are usually expressed as workflows, and so this notion is central to our work. To this end, we first outline the lifecycle of an abstract workflow (Section 3.2.1), and then formalise its structure (Section 3.2.2).

[1] In practice, it may be necessary for the consumer to exchange several messages with the provider in order to effect the desired behaviour. For example, a book-ordering service may require the consumer to first obtain a unique book identifier from its catalogue, create a virtual shopping basket, add the book and finally provide payment details. We do not explicitly cover such detailed interactions in our model, because they depend highly on the implementation of a particular service and are typically indivisible (i.e., it is generally not possible to order the book from one service but provide payment details to another).

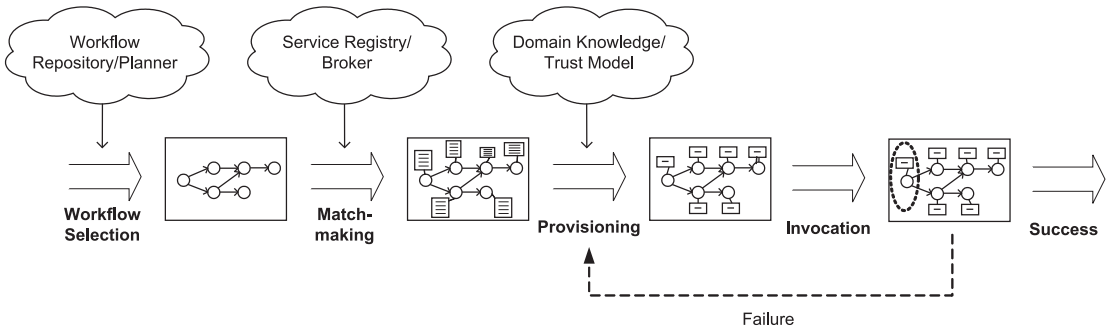


FIGURE 3.2: Lifecycle of a workflow

3.2.1 Workflow Lifecycle

Building on the work on abstract workflows outlined in Chapter 2, a service consumer in our model proceeds through four stages when executing a workflow (see Figure 3.2):

1. **Workflow Selection:** First, an abstract workflow is chosen to suit the consumer's current objectives. This is generally created either manually by domain experts or automatically by a planner that uses abstract templates of common service types. Due to the complexity of generating workflows, this may take place offline, allowing the consumer to retrieve suitable workflows from a repository. At this stage, tasks are only associated with their abstract service types (e.g., in the form of semantic meta-data) and not yet with any concrete service instances.

2. **Matchmaking:** Once an abstract workflow has been selected, tasks are mapped to candidate service instances via a matchmaking process. Here, the consumer searches a public service registry or requests matching services from a broker. This step uses the service type annotations provided by the abstract workflow to find suitable service instances. Additionally, the agent may, at this stage, apply security policies to filter the set of service instances (e.g., to remove services that do not adhere to certain protocols or encryption methods, or that cannot provide the necessary security certificates).

3. **Provisioning:** Given lists of matching services, the consumer now provisions individual service instances for each task of the workflow. This decision may constitute a tacit intention by the consumer to invoke the provisioned services for the respective tasks, and so it is not necessarily a binding commitment. The purpose of this stage is to allow the consumer to make predictions about the performance of a provisioned workflow, and to explore the space of candidate provisioned workflows. Specifically, it is possible for the consumer to evaluate and optimise the provisioned workflow using an appropriate utility function that encodes the value of successfully completing the workflow. During this stage, the consumer can make use of its own domain knowledge and possibly service performance information that is available from external sources, to identify particularly failure-prone

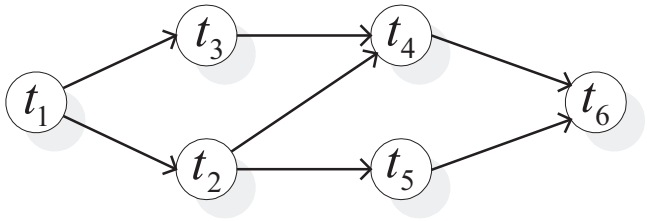


FIGURE 3.3: Example workflow consisting of six interdependent tasks. Circles represent the tasks in T and arrows represent the dependencies as given by E (transitive dependencies are omitted for readability).

tasks, and to proactively provision additional or more reliable services where necessary and where this increases the expected utility of the provisioned workflow.

4. **Invocation:** When appropriate services have been provisioned, the consumer starts to invoke the chosen services as dictated by the ordering constraints of the workflow. If services fail to complete their tasks, the consumer may provision other services, until the workflow is successfully completed.

In the following section, we formalise the concept of a workflow and show how to describe its value to the service consumer.

3.2.2 Workflow Structure

As discussed in Chapter 2, a workflow is typically a collection of tasks with appropriate ordering constraints. For this reason, we model it as a directed acyclic graph. Formally, a workflow is a tuple W:

$$W = (T, E, \tau, u) \tag{3.1}$$

where

- $T = \{t_1, t_2, t_3, \ldots, t_{|T|}\}$ is the set of tasks that make up the workflow.
- $E : \mathcal{P}(T \times T)$ is a strict partial order over T, denoting the precedence constraints. An element $(t_1, t_2) \in E$ means that completion of t_1 is necessary for t_2 to be started.
- $\tau : T \to \mathcal{T}$ maps each task to an abstract service type, where $\mathcal{T}$ is the set of all such descriptions.
- $u : \mathbb{R} \to \mathbb{R}$ is a utility function that maps the total completion time of a workflow to the related reward for the consumer.

To give an example, Figure 3.3 shows a workflow consisting of six tasks with some dependencies. Here, task t_1 has to complete successfully before any other tasks can be started. Task t_4

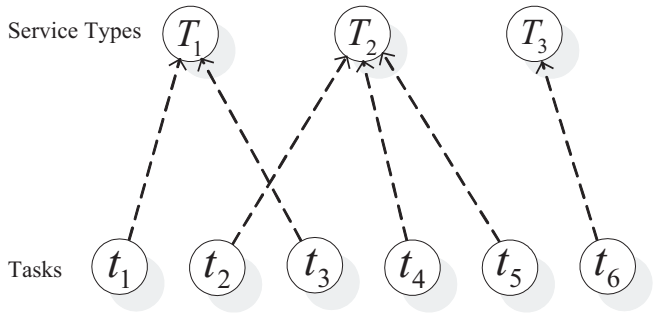


FIGURE 3.4: Relationships between tasks in workflow and abstract service types.

can only be started once tasks t_3, t_2 and (by transitivity) t_1 are completed. While this figure only shows the tasks T and edges E, Figure 3.4 highlights the relationship between tasks in the workflow and abstract service types (as given by function τ). In this example, several tasks share the same service type (for example t_1 and t_3).

The utility function u defines how the service consuming agent is rewarded for the successful completion of a workflow. This represents the value that the agent (or its owner) attaches to the workflow and may, in practice, be the expected financial gain of completing the workflow, or simply a private utility value, as commonly used in decision theory (Raiffa (1968)). Here, we assume that the reward is given only when the whole workflow is completed and that the amount of the reward depends on the time at which it is completed[2]. Hence, we use a general utility function that awards a maximum utility u_{max} when the workflow is completed within a given deadline t_{max}. When this deadline is exceeded, a penalty rate δ is deducted from u_{max} for every unit time step that the agent is late, until the agent gains no more positive utility, in which case it receives a reward of zero, regardless of whether the workflow is completed at a later stage or not. Formally, we express the utility function u as follows (with $u_{\text{max}} \geq 0$, $t_{\text{max}} \geq 0$ and $\delta > 0$):

$$u(t) = \begin{cases} u_{\text{max}} & \text{if } t \leq t_{\text{max}} \\ u_{\text{max}} - \delta(t - t_{\text{max}}) & \text{if } t > t_{\text{max}} \text{ and } t < t_{\text{max}} + u_{\text{max}}/\delta \\ 0 & \text{if } t \geq t_{\text{max}} + u_{\text{max}}/\delta \end{cases} \tag{3.2}$$

In this context, we use t_{zero} to denote the first integer time step at which the consumer no longer gains any reward, i.e., $t_{\text{zero}} = \lceil t_{\text{max}} + u_{\text{max}}/\delta \rceil$. In practice, when the consumer has not completed the workflow at time step t_{zero}, we treat it as failed and assume that execution will stop immediately (as doing otherwise is clearly irrational and may lead to infinite execution times).

To illustrate this, Figure 3.5 contains some example utility functions. The function labelled $u_1(x)$ rewards the consumer with $u_{\text{max}} = 400$ up to the deadline $t_{\text{max}} = 100$. When this deadline is exceeded, the utility of the workflow decreases slowly, with a penalty of only $\delta = 4$, thus representing a case where a small delay does not significantly penalise the consumer. In

[2]This is consistent with much previous work in the area — Collins et al. (2001) reward an agent with a fixed payoff for completed workflows, while Arunachalam and Sadeh (2004) and Irwin et al. (2004) describe utility functions that depend on the time of completion.

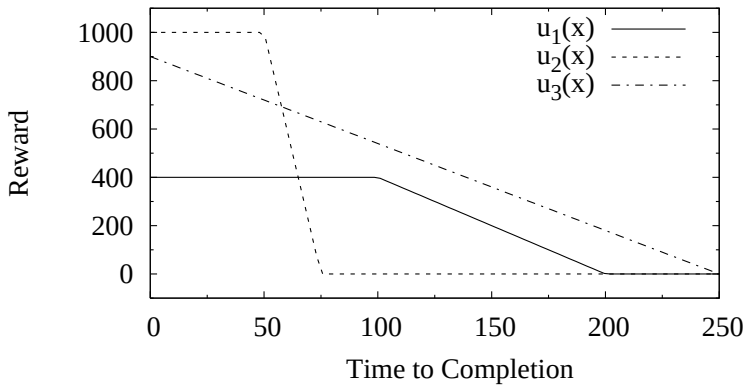


FIGURE 3.5: Examples of some representative utility functions.

contrast to this, $u_2(x)$ represents an example where time is more critical. Here, $\delta = 40$ and so the consumer loses all utility even if it is only 25 time steps late. Finally, $u_3(x)$ has no specific deadline ($t_{\max} = 0$), rewarding the agent purely based on the amount of time taken.

Given our formal model of a workflow, we now discuss the behaviour of service providing agents.

3.3 Service Provider Model

In this section, we give an overview of how service providers interact with the consumer, and how we describe the performance of their services. As we will consider different market mechanisms and varying amounts of knowledge about services, we restrict our description here to a brief summary and expand on it in more detail in later chapters.

First, as described above in Section 3.2.1, we assume that there is a mechanism for consumers to discover the available service instances for each task in T. To this end, we let $\mathcal{S} = \{s_1,\ s_2,\ \ldots,\ s_{|\mathcal{S}|}\}$ be the set of all services and we formalise the matchmaking phase in Figure 3.2 as a function, $\mu : \mathcal{T} \rightarrow \mathcal{P}(\mathcal{S})$, that maps abstract service types to sets of suitable services. For brevity, we let $S_i = \mu(\tau(t_i))$ (e.g., in Figure 3.1, we have $S_1 = \{s_1, s_2, s_3, s_4\}$).

Given this information about service instances, the consumer may then decide to invoke them for the appropriate workflow tasks. This happens either through *on demand* invocation, where the consumer requests the service only when it is required (this is the focus of Chapters 4 and 5), or through advance agreements (discussed in Chapter 6), where the consumer and provider first negotiate an explicit contract about when and how the service should be provided. In both cases, we assume that the consumer incurs some cost for invoking services.

Once a service is invoked for a given task, we consider two main outcomes: success and failure. If the service succeeds, the consumer receives notification of this some time after invocation. However, the exact time for this is usually uncertain, as it depends on network delays, the

complexity of a given task and the provider's decisions on how to allocate resources between competing consumers. If the service fails, we assume that no explicit notification is returned to the consumer — hence, we primarily consider *crash* failures in our work (Cristian (1991)). In Section 3.6, we briefly outline other types of failures that we do not currently address, and we show in Chapter 6 how this model can be extended to include explicit failure messages.

An important aspect of our model is that we assume multiple services can be invoked for a single task. This means that the consumer may invoke several in parallel, for example to increase the overall likelihood that the task will be completed by one of them. Similarly, it may delegate a failed task to a different provider. In this context, we assume that a single success is sufficient and so a task is considered completed as soon as the first invoked service is successful. However, we also assume that the consumer generally has to pay for all invoked services, regardless of the outcome, but in Chapter 6, we consider scenarios where providers refund or even compensate the consumer for service failures. Furthermore, throughout the thesis, we assume that outcomes of different services (or of the same service, but for different tasks) are independent. Again, we briefly return to this assumption in Section 3.6 and discuss some cases where it does not necessarily hold.

As outlined in Section 3.2.1, the consumer uses some performance information about service instances to decide which ones to provision for the tasks of the workflow. Although we will cover this in more detail in later chapters, we consider a number of basic parameters that describe each service:

- $c(s_i) \in \mathbb{R}$ is the invocation cost of service s_i. Usually, this will be a financial remuneration paid by the consumer to the service provider, but could also represent a communication cost. We assume that it is expressed in the same units as the workflow reward (given by utility function $u(t)$).
- $f(s_i) \in [0, 1]$ is the failure probability of service s_i. This is the probability that invoking the service for a particular task will result in failure (as described above).
- $d(s_i, x) \in \mathbb{R}$ is the duration function of service s_i. This is a probability density function representing the time between sending a request to invoke service s_i and receiving notification of a successful outcome (as observed by the consumer and conditional on overall success). As such, this function encapsulates the general uncertainty in the service execution time, including factors such as network propagation delays, competition with other consumers and task uncertainty. For convenience, we denote the associated cumulative density function as $D(s_i, x) = \int_0^x d(s_i, y)\, \mathrm{d}y$ (i.e., $D(s_i, x)$ is the probability that the time between requesting service s_i and being notified of its success is x or less). Intuitively, we assume that durations are always strictly positive ($\forall x < 0 \cdot d(s_i, x) = 0$).

In the following section, we present a general algorithm that sketches the behaviour of a service consuming agent. This further formalises the consumer's interactions with the providers and forms the basis for our proposed strategies in Chapters 4 – 6.

3.4 Basic Service Consumer Algorithm

Algorithm 3.1 shows the general behaviour of a service consumer. Here, we concentrate on the sequence of actions to give us a basic framework for our work, leaving the actual decision-making procedures (INITIALISE, UPDATE, STOPCONDITION and ENGAGESERVICES) to later chapters. In this algorithm and throughout our work, we assume that time passes in discrete, uniform time steps, representing the intervals at which the consumer senses its environment. To this end, we denote the first time step as $\hat{t} = 0$, the second as $\hat{t} = 1$, and so on.

At the beginning of the algorithm, two variables are first initialised to keep track of the current time (line 2) and the overall profit the consumer has accumulated (line 3). Then, in line 4, the consumer selects an appropriate workflow, W, to achieve its current objective (denoted by Λ). This corresponds to the workflow selection stage described in Section 3.2.1 (however, as we concentrate on the provisioning of a given workflow, we do not cover this stage in more detail). Given W, the consumer next performs an initial decision-making procedure, INITIALISE (line 5). During this, it may discover available services (corresponding to the matchmaking phase) and make initial decisions on which services to invoke for the tasks of its workflow (the provisioning phase).

Lines 7 – 24 constitute the main loop of the algorithm, with each iteration representing the actions performed during a single time step. In more detail, the variable $\mathcal{O}$ in line 8 is first set to contain information about the most recent service outcomes that occurred between the current and the previous time step (this is later used to update the consumer's state). For now, we assume that $\mathcal{O} : \mathcal{P}(T \times \mathcal{S})$ is simply a set of tuples that indicate the tasks and associated services that have successfully been completed in that time interval (in Chapter 6, we consider other outcomes as well). Next, in line 9, any penalties for failed services are paid to the consumer (we only consider this in Chapter 6). If the outcomes in $\mathcal{O}$ suggest that all the tasks have been completed, the overall profit is calculated using $u(\hat{t})$ and the algorithm terminates by returning the profit and a status message to indicate success (line 11). On the other hand, if the workflow is not complete and will no longer result in a positive, non-zero reward, the algorithm also terminates (line 13).

If some tasks are still uncompleted, the consumer updates its internal state based on the service outcomes (line 15). This involves updating the progress of the workflow and possibly adapting its initial provisioning decisions. Next, when service providers demand explicit contract negotiations (as discussed in Chapter 6), the consumer may negotiate with service providers (line 16). After this, the consumer may abandon the workflow, for example when it seems infeasible to complete it in time (line 17). This is followed by service invocations, during which the consumer requests any provisioned services to be started (line 20). This corresponds to the invocation stage detailed in Section 3.2.1. Finally, the profit is updated to take into account any costs incurred during negotiation and invocation (line 21) and the time is advanced (line 22).

Algorithm 3.1 Service consumer behaviour.

procedure SERVICECONSUMER
  $\hat{t} \leftarrow 0$ ▷ Current time
  $\hat{p} \leftarrow 0$ ▷ Current profit
  $W \leftarrow$ SELECTWORKFLOW(Λ) ▷ Select workflow
  INITIALISE(W) ▷ Initial matchmaking and provisioning
  $\mathcal{O} \leftarrow \emptyset$ ▷ Variable to hold most recent service outcomes
  loop ▷ Main loop
    $\mathcal{O} \leftarrow$ recent service outcomes
    $\hat{p} \leftarrow \hat{p} +$ penalties ▷ Pay out penalties for failed services
    if all tasks completed **then**
      return $(\hat{p} + u(\hat{t}), \texttt{success})$ ▷ Successfully completed workflow
    else if $u(\hat{t}+1) \leq 0$ **then**
      return $(\hat{p}, \texttt{failed})$ ▷ Failed to complete workflow in time
    else
      UPDATE($\mathcal{O}$) ▷ Update consumer with outcomes
      NEGOTIATESERVICES ▷ Negotiate service provisions
      if STOPCONDITION $=$ `true` **then**
        return $(\hat{p}, \texttt{abandoned})$ ▷ Abandoned workflow
      end if
      INVOKESERVICES ▷ Invoke services
      $\hat{p} \leftarrow \hat{p} -$ costs ▷ Accumulate costs of provisioned/invoked services
      $\hat{t} \leftarrow \hat{t} + 1$ ▷ Advance time
    end if
  end loop
end procedure

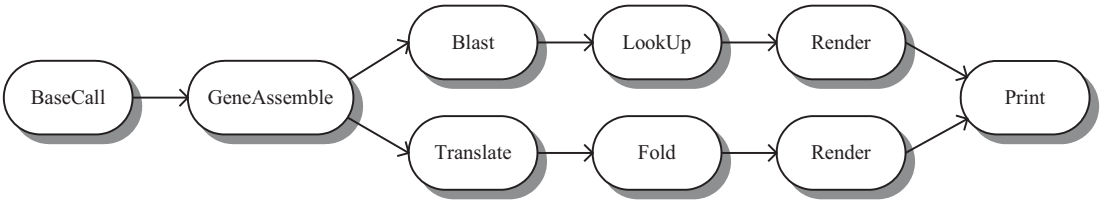


FIGURE 3.6: Example bioinformatics workflow, based on workflows described by Smith et al. (1997), Kochut et al. (2003) and O'Brien et al. (2004).

To further illustrate the types of workflows that a consumer in a service-oriented system may face, we briefly discuss an example workflow in the next section.

3.5 Illustrative Workflow

Throughout this thesis, we illustrate our work using a simple workflow from the bioinformatics domain — an area that relies heavily on computationally intensive services and that has increasingly seen the establishment of large distributed Grid systems for sharing resources, as exemplified by the GriPhyN (Deelman et al. (2003a)), myGrid (Oinn et al. (2006)) and Comb*e*Chem (Coles et al. (2005)) projects. For our example, we assume that a scientist has just sequenced a previously unknown gene of a bacterium, and is now interested in visualising the shape of the associated protein. For this, she has to carry out a number of tasks, which are shown in Figure 3.6.

Her initial data comprises a large set of overlapping DNA fragments in the form of chromatograms, as is common in shotgun DNA sequencing (Ewing et al. (1998)). These show characteristic light traces at different wavelengths, corresponding to the four bases found in a DNA sequence. As these traces typically contain some noise and errors, the scientist first needs to run a base-calling service (*BaseCall*). This translates the chromatograms to the corresponding base sequences, attaching a quality value to each base in the process that denotes how accurate the assignment of the base is. The resulting base sequences are then assembled to a single continuous DNA sequence by identifying and merging overlapping fragments, using the quality values to find and repair errors. This task is performed by a sequence-assembling service, which also identifies and isolates the coding region of the gene (*GeneAssemble*).

When the coding region of the gene has been assembled, it is then translated to the corresponding amino acid sequence using a simple translation service (*Translate*). As the primary structure of the protein, this forms the input to the computationally-intensive folding service (*Fold*), which predicts the 3-dimensional shape of the protein based on a search for the conformation with the lowest free energy. The output of this — a file containing the tertiary structural data — is then rendered in high resolution using an appropriate graphics service (*Render*). In parallel with the folding simulation, the scientist is also interested in comparing the new gene to previously discovered sequences. To this end, she searches through public collections of known proteins to

find the closest match using a specialised service (*Blast*), and then accesses commercial database services to retrieve structural information about the protein (*LookUp*). This is rendered again, and both images are printed as part of a report on a local printer (*Print*).

Now, some service types in this example require a significant computational effort and may take a considerable amount of time to complete. In this context, our utility function, as given by Equation 3.2, allows the owner of the service consuming agent (the scientist in this example) to succinctly encode the overall utility of the workflow and how this relates to the time taken. For example, if the scientist needs the results later in the day, but is not overly concerned about waiting a bit longer, a utility function with a low penalty δ, such as u_1 in Figure 3.5, is appropriate. If, on the other hand, the results are critical for a presentation she is giving to a funding committee in the next 90 minutes, a utility function such as u_2 expresses the urgency and high value of the workflow more suitably.

These examples serve to highlight some of the challenges we seek to address in our work. Especially in the latter case, uncertain service durations can easily jeopardise the successful completion of the workflow — for example, when one of the provisioned service instances is in high demand and therefore takes longer than expected. Similarly, service failures can lead to missed deadlines and to higher costs (as replacement services may need to be found and invoked). Furthermore, the service types discussed above may be offered by many heterogeneous agents — for example, there may be instances for the *Render* service type that are very expensive and reliable (perhaps because they run on dedicated graphics workstations), but also others that are cheap, unreliable and usually much slower (these may be executing on simple desktop machines).

Hence, we need a decision mechanism that can anticipate some of the potential problems and mitigate them by provisioning the workflow in a flexible manner, for example, by provisioning multiple providers for a given task, by re-provisioning failed tasks and by choosing appropriately among several heterogeneous providers. Before discussing our proposed algorithms for this in detail, we conclude this chapter by detailing some of the limitations of the model we have adopted. This discussion is necessary, as we have made some assumptions that may not always hold in practice.

3.6 Model Assumptions and Limitations

Although we have striven to present a model that is applicable to a large range of service-oriented scenarios, we have had to make a number of simplifying assumptions about our problem domain that may not hold in all potential application areas. On the one hand, these assumptions were necessary to produce a formal model that is amenable to efficient mathematical analysis, and on the other hand, they allowed us to present and deal with a general problem rather than concentrate on domain-specific constraints that may occur in a concrete application. We believe

that our assumptions are reasonable in most large distributed systems and that our model constitutes a solid basis for more specific extensions. In this section, we explicitly list and justify the assumptions we have made.

- **Failure Model:** We have chosen to focus mainly on silent crash failures at this time. Compared to explicit failure messages, silent failures are more challenging to deal with (clearly, a consumer receiving such messages will perform at least as well as one that does not). Furthermore, they are realistic in distributed environments, where service providers do not reveal their internal state, and where network or machine failures can lead to communication losses. However, we currently do not deal with Byzantine failures, which include the return of corrupt service results. Hence, we must assume that service results can be tested for correctness (in fact, many intractable problems can be efficiently verified), but we plan to relax this limitation in future work.

  We also assume that failures (and durations) of different services are independent of each other. We believe that this is generally the case in large-scale distributed systems, where services reside on physically separate machines, use different implementations and do not directly interfere with each other. Despite this, failures may occasionally be correlated — e.g., when two services rely on a common third service, or when several systems are attacked by the same virus.

  Furthermore, failures between separate tasks may not always be independent either. For example, when provisioning the same instance for several tasks, it is possible that there will be some correlation between the outcomes of these tasks. In some cases, the failure of certain tasks may also require the consumer to repeat previous tasks (e.g., when the service input data was first converted by another service to a specific format). However, we believe that this usually happens within the context of a single task, where several operations of the same service have to be invoked to achieve the overall objective (and where a failure would imply repeating these operations with a different provider). While other dependencies are also possible, we chose not to include such constraints in our current model for conciseness.

  Finally, we do not explicitly consider transient or intermittent failures — hence, we do not attempt to repeatedly invoke the same failed service for a given task several times (however, if such a behaviour does not incur additional costs, it can easily be incorporated into the overall failure probability and duration distribution of a task).

- **Performance Information:** As we concentrate on the provisioning problem rather than learning techniques, we assume that the service consumer has accurate performance information about the providers for each task[3]. In practice, such information may be domain knowledge provided by experts during workflow generation (Ng and Abramson (1990)), by inference over the task descriptions and related data (Maximilien and Singh (2004)), or by statistical estimation based on previous interactions with similar services, possibly

[3]However, we show in Appendix A that our proposed approach is robust to moderate inaccuracies.

provided by a trusted monitoring service (Teacy et al. (2006)). However, obtaining this knowledge is clearly non-trivial and has been the subject of much ongoing research. Furthermore, there may be tangible costs associated with obtaining such trust information (e.g., when a monitoring service charges for its information, or when a consumer has to actively explore the population of providers to gather statistical averages). We do not cover these costs of querying and maintaining trust information at this time, but we envisage that existing work on the value of information in uncertain environments can be used to extend our model in the future (Dearden et al. (1999); Teacy et al. (2008)).

Moreover, we currently represent uncertain service durations using simple non-conditional probability density functions. This is a common approach for modelling stochastic systems, but it is possible to envisage more detailed joint distributions to be available, for example to model varying service durations at different times of the day, on weekends, or based on observations about current network traffic.

- **Payment Model:** Our model assumes that the service consumer is charged a fixed price per invocation. We believe that this is realistic in many dynamic service-oriented systems, where providers and consumers form only loose short-term agreements. However, it should be noted that other pricing schemes have been proposed, including some that allow multiple invocations of the same service over a certain period of time (Dan et al. (2004)).

  Additionally, we currently assume free disposal of unwanted services, i.e., that several successful service invocations for the same task do not incur additional penalties above their normal cost. This may be realistic in Grid scenarios, where the results of data processing services can be disregarded without costs, but in a supply-chain application, the disposal of unused goods may incur additional charges (especially for chemicals or dangerous materials).

- **Reward Model:** Our reward function encodes the value of completing a workflow at a given time, and it intuitively follows the general form of many contracts in other domains. However, certain application scenarios might require a more expressive function that depends on multiple dimensions (e.g., the overall time and the perceived quality of some end-product).

- **Model Scope:** To obtain a general system model, we currently do not consider specific domain-dependent constraints that may occur in particular workflow applications. For example, we do not cover cases where service instances have mutually exclusive side-effects or where there are dependencies between the instances provisioned for several tasks[4]. The latter case might occur in scenarios where the choice of an earlier service instance dictates the applicability of services for subsequent tasks. We also represent workflows as directed acyclic graphs, which is consistent with much related work, but we

[4]Most commonly, such dependencies occur when invoking several operations on a service to achieve some higher-level objective (such as the selection and payment operations when ordering goods online). As described in Section 3.1, these low-level dependencies are subsumed by the high-level service concept we use in our model.

note that realistic applications often require more complex structures, including branches and loops.

Furthermore, we do not currently consider explicit transaction mechanisms (i.e., mechanisms for coordinating multiple inter-dependent service invocations in order to ensure a consistent outcome). We believe that such mechanisms are typically subsumed by our high-level view of tasks that will often include multiple messages between the consumer and provider, and that may be underpinned by a transaction mechanism (e.g., to ensure that a book order takes place only when all constituent operations are successful). Despite this, there are also cases where several high-level tasks may be coordinated via a long-running transaction. This is common in domains where services are highly interdependent and where the consumer may need to retain the freedom to retract previous service requests (e.g., when booking hotels and airline tickets in the travel domain). However, as argued in Section 2.4.1, we believe that such transaction mechanisms cannot generally be relied on for cancelling previous tasks and thereby take a more pessimistic approach in modelling our system without transactional support.

Finally, in line with the overall aim of this thesis, we focus solely on uncertainty in the behaviour of service providers. Hence, we assume that workflows are correct, that appropriate matchmaking algorithms correctly identify suitable providers and that the consumer is able to translate between heterogeneous data formats. In practice, such problems are far from trivial, but they are not the focus of this work.

In Chapter 7, we will re-examine some of these limitations and show how our model can be extended to handle them.

3.7 Summary

In this chapter, we have outlined a general system model and agent framework, which will form the basis of our work. In doing so, we have concentrated on formalising the workflows a consumer faces, and we have detailed some assumptions about how the consumer may interact with service providers. We have also briefly introduced the basic behaviour of providers and discussed how we quantify the uncertain outcomes of services using probabilistic measures. In the following chapters, we will elaborate on this model to cover different environments. More specifically, in Chapter 4, we develop a strategy for cases where services are invoked on demand, but where the service consumer has no detailed performance information about individual services. Then, in Chapter 5, we look at environments where such information is available to the consumer. Finally, in Chapter 6, we consider environments where explicit service contracts are negotiated in advance and where the availability and performance characteristics of services may change dynamically. We decided to address these scenarios separately, as this allows us to best exploit the specific characteristics of each one. For example, when limited information is available, we can perform particularly fast calculations. When considering more complex

environments, on the other hand, we propose decision algorithms that deal better with the larger decision spaces. Taken together, these techniques therefore represent a set of algorithms and tools that can be used in a range of different environments.

Chapter 4

Service Provisioning with Limited Performance Information

Having devised a model for service-oriented systems in the previous chapter, we now outline a number of strategies for provisioning services for abstract workflows. In this chapter, we concentrate on systems where services are invoked purely on demand (i.e., without the need for explicit advance agreements) and where the information about the performance of services is highly limited (Chapters 5 and 6 will deal with systems where more detailed information is available and where services are provisioned in advance). To this end, in Section 4.1, we formalise these assumptions by extending our system model. This is followed by a discussion of a number of provisioning techniques: in Section 4.2, we begin by outlining a *naïve* strategy that formalises many current approaches towards service provisioning that do not consider service uncertainty. Then, in Section 4.3, we describe three strategies that rely on multiple services to satisfy single tasks (*parallel(n)*, *serial(w)* and *hybrid(n,w)*) and that are broadly based on simple redundant strategies found in related work. These are then combined, in Section 4.4, into a *flexible* provisioning strategy that reasons explicitly about its provisioning decisions and that constitutes the main contribution of this chapter. The chapter is concluded by a thorough empirical investigation, in Section 4.5, into the performance of our proposed strategies.

In devising the *flexible* strategy, we address four of our agent requirements outlined in Section 1.4.3. Specifically, the strategy reacts dynamically to failures by re-provisioning services (Requirement A.2.a) and it avoids failures proactively by redundantly provisioning services where appropriate (Requirement A.2.b). Furthermore, it makes flexible, automatic decisions with the aim of maximising the agent's utility (Requirement A.1) and our approach uses heuristic approximations that make it suitable for large problem instances (Requirement A.3).

4.1 Model Extension

We begin by looking at a simple system model, which builds closely on that described in the previous chapter. As before, we assume that the consumer knows the overall set of services that may satisfy a given task (denoted S_i for task t_i). However, we make a number of assumptions about how the consumer is able to interact with these services and the information it has about them:

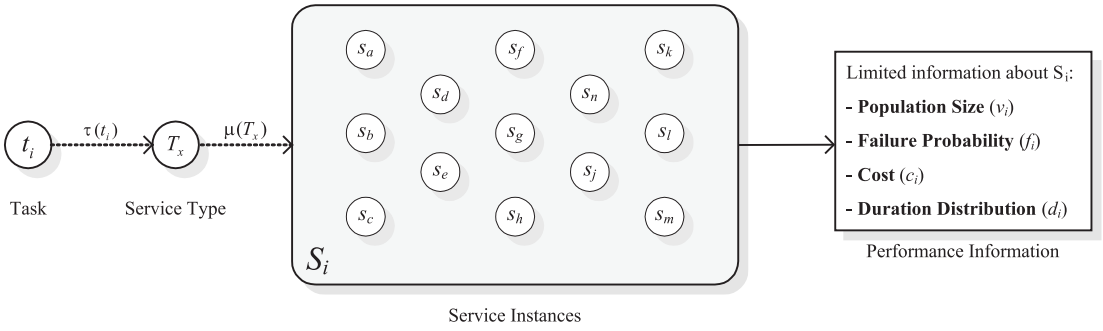


FIGURE 4.1: Information that is available about the services available for each task.

- **On demand invocation:** In this chapter, we assume that services are always invoked on demand. To this end, when the consumer decides that execution of a task t_i should start (provided all predecessors of t_i have been completed), it simply sends a request to any of the members of S_i during the INVOKESERVICES procedure of Algorithm 3.1.
- **Limited performance information:** We assume that the consumer does not have detailed performance information about each individual service. Rather, this is restricted to probabilistic estimates and distributions about the set S_i as a whole. Specifically, we assume the following to be available (as shown in Figure 4.1):
  - t_i is a task in the workflow.
  - $T_x = \tau(t_i)$ is the service type associated with the task.
  - $S_i = \mu(T_x)$ is the set of valid service instances that are capable of completing the task.
  - $v_i = |S_i|$ is the number of valid service instances.
  - f_i is the failure probability of a randomly drawn member of S_i.
  - c_i is the cost[1] of each service in S_i.
  - d_i is the duration distribution function of a randomly drawn, successful member of S_i, and D_i is the associated cumulative distribution function.

[1]We assume here that the cost of services is homogeneous within S_i, i.e., that all services cost exactly c_i. However, the techniques developed in this chapter apply similarly when there is some uncertainty in the cost of each service, with c_i representing the average cost.

We make these assumption because they apply in a range of realistic application scenarios. Specifically, on demand invocation is the predominant invocation mechanism in many current service-oriented systems, where service interfaces are published in registries and then simply invoked when needed, much like remote procedure calls (as outlined in Chapter 2). Regarding our second assumption, performance information may be limited for several reasons. First, more detailed information may simply not be available, for example in the absence of a reliable reputation mechanism and when the consumer has not had the benefit of a large number of previous interactions with all providers. Second, the service-oriented systems we consider are open and dynamic, which may make it difficult to collect specific performance information, as services enter and leave at will and may even change their identity. Finally, services may also be homogeneous or highly similar, for example if they rely on the same algorithms or implementations.

4.2 The Naïve Strategy

We begin by looking at the currently predominant approach to service provisioning in the literature. This gives us a basic benchmark against which we can evaluate the strategies we develop in this section, and, in doing so, serves to highlight the shortcomings of current work.

Now, as described in Chapter 2, most of the current work on Web services focusses solely on the functional descriptions of services. In such research, descriptions are typically assumed to be truthful and deterministic, and thus service-consuming agents do not explicitly consider the provisioning stage, but rather pick *any* single service that matches their requirements. Since such a strategy does not consider service failures, we term it *naïve* and describe it more formally as follows:

Definition 2 (Naïve Strategy)**.** A consumer agent following a ***naïve*** strategy always provisions a single randomly chosen service of the correct type for each task.

Algorithm 4.1 formalises this strategy as an implementation of the abstract procedures introduced in Algorithm 3.1. The first procedure, NAÏVE-INITIALISE, in lines 1 – 11 constitutes the main decision-making logic. Here, the agent initialises a set, $\wp$, which will contain a mapping from tasks to services (line 3). This is then populated by finding appropriate service instances for each task using the matchmaking function μ (line 5) and then provisioning a service that is picked uniformly at random from the set of matching instances (line 7).

The remaining procedures are straight-forward — NAÏVE-UPDATE keeps track of any successfully completed tasks, NAÏVE-STOPCONDITION always returns `false` as the strategy does not reason about the feasibility of the workflow, NAÏVE-NEGOTIATESERVICES does nothing as negotiations are not necessary, and NAÏVE-INVOKESERVICES invokes the services selected during the initial provisioning.

Algorithm 4.1 *Naïve* strategy that selects a single valid service for each task.

```
1:  procedure NAÏVE-INITIALISE(W)
2:      T_comp ← ∅                                  ▷ Keeps track of completed tasks
3:      ℘ ← ∅                                       ▷ Provisioning decisions
4:      for all t_i ∈ T do                          ▷ Iterate through tasks
5:          S_i ← μ(τ(t_i))                         ▷ Matchmaking
6:          if S_i ≠ ∅ then
7:              s_x ∈ S_i                           ▷ Choose random service
8:              ℘ ← ℘ ∪ {(t_i, s_x)}                ▷ Store provision decision for t_i
9:          end if
10:     end for
11: end procedure

12: procedure NAÏVE-UPDATE(O)
13:     T_new ← {t_i | ∃s_x · (t_i, s_x) ∈ O}       ▷ Recently completed tasks
14:     T_comp ← T_comp ∪ T_new                     ▷ Add to completed tasks
15: end procedure

16: procedure NAÏVE-STOPCONDITION
17:     return false                                ▷ Never abandon
18: end procedure

19: procedure NAÏVE-NEGOTIATESERVICES
20:     do nothing                                  ▷ Not necessary here
21: end procedure

22: procedure NAÏVE-INVOKESERVICES
23:     for all (t_i, s_x) ∈ ℘ do                   ▷ Iterate through ℘
24:         if ∀(t_j, t_i) ∈ E · t_j ∈ T_comp then  ▷ Is t_i executable?
25:             INVOKE(s_x, t_i)                    ▷ Invoke service s_x for task t_i
26:             ℘ ← ℘ \ {(t_i, s_x)}                ▷ Remove to avoid re-invocation
27:         end if
28:     end for
29: end procedure
```

A major shortcoming of this *naïve* strategy is that it is highly vulnerable to service failures. A single failure means that the whole workflow is lost, along with all investments already made. To reduce this risk, we discuss several simple techniques in the following sections for dealing with service failures.

4.3 Robust Provisioning Strategies

We proceed in this section by presenting three strategies for provisioning services in a manner that anticipates failures and attempts to reduce their impact on the consumer's workflows. All of the following strategies require some manual intervention (by adjusting parameters that dictate their behaviour) and so are not very well suited to large complex systems, where automation is not only desirable, but perhaps necessary due to the scales involved. However, the strategies present the basic techniques that we will use in Section 4.4 to design a flexible, automated provisioning technique. During our empirical investigation in Section 4.5, they also serve to highlight and quantify the potential benefits of actively dealing with service failures.

4.3.1 Parallel Provisioning

The first strategy we discuss in this context uses parallel provisioning to *proactively* control the effect of unreliable services, thereby addressing our Requirement A.2.b. As discussed in the previous chapter, a feature of service-oriented systems is the fact that several service instances may match a single semantic service description. For this reason, a consumer may benefit by delegating each of its tasks to several providers at the same time, rather than relying on a single service.

To highlight the advantage of this approach, let $X_n \in \{\text{success}, \text{failure}\}$ be a random variable indicating the outcome for a task t_i when n services are invoked in parallel for this task. The probability that a single service ($n = 1$) successfully completes the task is then $P(X_1 = \text{success}) = 1 - f_i$. When invoking two service instances in parallel ($n = 2$), we have a success probability $P(X_2 = \text{success}\ ) = 1 - f_i^2$. For the general case with n services, we thus have:

$$P(X_n = \text{success}) = 1 - f_i^n \tag{4.1}$$

This means that the probability of success increases as more providers are provisioned for a single task. However, if a non-zero cost is associated with each provision, then the total cost incurred rises with n. Based on this, we can formulate a strategy that uses parallel provisioning to reduce the probability of workflow failures:

Definition 3 (Parallel(n) Strategy)**.** A consumer following a ***parallel(n)*** strategy always provisions exactly n randomly chosen services of the correct type for each task (n is a single constant for all tasks).

Algorithm 4.2 *Parallel(n)* strategy that provisions n valid services for each task.

procedure PARALLEL-INITIALISE(W)
  $n \leftarrow$ constant specified by agent owner ▷ Number of parallel services
  $T_{\text{comp}} \leftarrow \emptyset$ ▷ Keeps track of completed tasks
  $\wp \leftarrow \emptyset$ ▷ Provisioning decisions
  for all $t_i \in T$ **do** ▷ Iterate through tasks
    $S_i \leftarrow \mu(\tau(t_i))$ ▷ Matchmaking
    PARALLEL-PROVISION(i, n) ▷ Provisioning
  end for
end procedure

procedure PARALLEL-UPDATE($\mathcal{O}$)
  $T_{\text{new}} \leftarrow \{t_i \mid \exists s_x \cdot (t_i, s_x) \in \mathcal{O}\}$ ▷ Recently completed tasks
  $T_{\text{comp}} \leftarrow T_{\text{comp}} \cup T_{\text{new}}$ ▷ Add to completed tasks
end procedure

procedure PARALLEL-INVOKESERVICES
  for all $(t_i, \hat{S}_x) \in \wp$ **do** ▷ Iterate through $\wp$
    if $\forall (t_j, t_i) \in E \cdot t_j \in T_{\text{comp}}$ **then** ▷ Is t_i executable?
      for all $s_y \in \hat{S}_x$ **do** ▷ Iterate through services in $\hat{S}_x$
        INVOKE(s_y, t_i) ▷ Invoke service s_y for task t_i
      end for
      $\wp \leftarrow \wp \setminus \{(t_i, \hat{S}_x)\}$ ▷ Remove to avoid re-invocation
    end if
  end for
end procedure

procedure PARALLEL-PROVISION(i, n)
  if $S_i \neq \emptyset$ **then**
    $\hat{S}_x \leftarrow \emptyset$ ▷ Set of chosen services, initially empty
    $n' \leftarrow \min(|S_i|, n)$ ▷ Number of services to provision
    for $j = 1$ to n' **do**
      $s_y \in S_i$ ▷ Choose random service
      $\hat{S}_x \leftarrow \hat{S}_x \cup \{s_y\}$ ▷ Add service to chosen set
      $S_i \leftarrow S_i \setminus \{s_y\}$ ▷ Remove from set of available services
    end for
    $\wp \leftarrow \wp \cup \{(t_i, \hat{S}_x)\}$ ▷ Store provision decision for t_i
  end if
end procedure

For this strategy, n is a fixed constant that is determined by a human user. The strategy *parallel(1)* is equivalent to the *naïve* strategy, and a higher value for n implies a generally higher resilience against failures. In Algorithm 4.2, we show how the strategy is implemented. This is similar to the *naïve* strategy, but it now selects up to n services in lines 28 – 32. The PARALLEL-INVOKESERVICES has been adapted to reflect this, but all other procedures remain unchanged (PARALLEL-STOPCONDITION and PARALLEL-NEGOTIATESERVICES are not shown here for brevity).

While reducing the probability of workflow failures, the strategy discussed in this section lacks any capacity to react to failures *after* they have occurred (Requirement A.2.a). This is addressed by the strategy in the following section.

4.3.2 Serial Provisioning

The second strategy we propose deals *reactively* with service failures (Requirement A.2.a). Rather than relying on parallel provisioning, it re-provisions services when it becomes likely that a previously provisioned service has failed. To this end, the consumer first provisions a single service and then waits for some time. If the service has not been successful after this time, the consumer assumes it has failed[2] and tries a different one, repeating the process if necessary, until the task has been completed. However, as providers have non-deterministic duration times and because they do not notify the consumer of failure, the consumer has to choose an appropriate waiting period. This period should give the service a reasonable time to finish, but should not waste unnecessary time when it has most likely already failed.

With this in mind, let $X_{s,w} \in \{\text{success}, \text{failure}\}$ be a random variable indicating the outcome of invoking up to s service instances in series for a task t_i. Here, s is the number of services that are available in total, as the consumer will continue invoking services until the task is successful (hence, $s = v_i = |S_i|$), and w is the chosen waiting period. To calculate the success probability of a single service in this case, we can use the cumulative density function D_i, derived from d_i. Hence, we have $P(X_{1,w} = \text{success}) = (1 - f_i) \cdot D_i(w)$, where $1 - f_i$ is that probability that the service will succeed, and $D_i(w)$ is the probability that this will happen within w time steps. Generalising this for invoking s services in sequence, we get the overall success probability

[2]Here and in the remainder of this chapter, any services that are assumed to have failed in this way are subsequently ignored by the consumer (even if they succeed at a later time). We make this assumption for two reasons: first, such time-out behaviour is common in many distributed applications and often explicitly part of service-oriented frameworks (such as CORBA or HTTP-based Web services); second, it allows us to make efficient predictions about service performance by considering a single invocation at a time (rather than many interleaved invocations). However, in Chapter 5, we show how this assumption can be relaxed.

when invoking s services in series:

$$\begin{aligned} P(X_{s,w} = \text{success}) &= 1 - P(X_{s,w} = \text{failure}) \\ &= 1 - P(X_{1,w} = \text{failure})^s \\ &= 1 - (1 - P(X_{1,w} = \text{success}))^s \\ &= 1 - (1 - (1 - f_i) \cdot D_i(w))^s \end{aligned} \tag{4.2}$$

This is generally less than the success probability of invoking the same number of services in parallel, and the average time taken will also be higher for serial provisioning because of the additional waiting time that is introduced. On the other hand, the average cost drops, because costs are only incurred at the time of invocation.

Hence, we define a new reactive strategy as follows:

Definition 4 (Serial(w) Strategy)**.** A consumer following a ***serial(w)*** strategy always provisions exactly one randomly chosen service of the correct type for each task. After a waiting period of w time units, if no success has been registered yet and if there are still more available services, the agent re-provisions a new, randomly chosen service and continues in this manner until the task is completed or no more services are left (w is a single constant for all tasks).

This strategy is illustrated by Algorithm 4.3. The procedure SERIAL-INITIALISE (lines 1 – 10) now contains a new constant w, which is the waiting time before invoking a new service[3]. Furthermore, we have added a variable, $T_{\text{inv}} : T \rightarrow \mathbb{R}_0^+$, which keeps track of the invocation times of tasks (line 3). This is checked at each time step, in order to identify and re-provision any tasks that have timed out (lines 17 – 22). New invocation times are added to T_{inv} during invocation (line 26) and removed when the task is eventually successful (line 14).

The two approaches discussed in the preceding sections, *serial(w)* and *parallel(n)*, cover two of our original requirements, A.2.a and A.2.b respectively. However, they are currently separate from each other and so may be less useful in practice. Instead, it is more desirable to devise a single strategy that addresses both requirements at the same time. For that reason, the following section generalises the preceding strategies and provides us with a basic foundation for developing a more flexible approach in Section 4.4.

4.3.3 The Hybrid Strategy

In order to address service failures proactively, but also react to failures as they occur, a consumer agent can provision multiple services in parallel, and then re-provision more services for the same task when a failure has occurred. Such a strategy increases the probability that a task is completed on the first attempt, but also includes a mechanism for responding to failures.

[3]The special case of *serial(∞)* is equivalent to the *naïve* strategy.

Algorithm 4.3 *Serial(w)* strategy that re-provisions unsuccessful tasks after w time units.

procedure SERIAL-INITIALISE(W)
  $w \leftarrow$ constant specified by agent owner ▷ Time-out for re-provisioning
  $T_{\text{inv}} \leftarrow \emptyset$ ▷ Keeps track of invocation times
  $T_{\text{comp}} \leftarrow \emptyset$ ▷ Keeps track of completed tasks
  $\wp \leftarrow \emptyset$ ▷ Provisioning decisions
  for all $t_i \in T$ **do** ▷ Iterate through tasks
    $S_i \leftarrow \mu(\tau(t_i))$ ▷ Matchmaking
    SERIAL-PROVISION(i) ▷ Provision (see procedure below)
  end for
end procedure

procedure SERIAL-UPDATE($\mathcal{O}$)
  $T_{\text{new}} \leftarrow \{t_i \mid \exists s_x \cdot (t_i, s_x) \in \mathcal{O}\}$ ▷ Recently completed tasks
  $T_{\text{comp}} \leftarrow T_{\text{comp}} \cup T_{\text{new}}$ ▷ Add to completed tasks
  $T_{\text{inv}} \leftarrow \{(t_i, y) \mid (t_i, y) \in T_{\text{inv}} \wedge t_i \notin T_{\text{new}}\}$ ▷ Remove completed from T_{inv}
end procedure

procedure SERIAL-INVOKESERVICES
  for all $(t_i, y) \in T_{\text{inv}}$ **do** ▷ Check for timed out tasks
    if $\hat{t} - y \geq w$ **then** ▷ Invoked at least w time steps earlier?
      $T_{\text{inv}} \leftarrow T_{\text{inv}} \setminus \{(t_i, y)\}$ ▷ Timed out
      SERIAL-PROVISION(i) ▷ Re-provision task
    end if
  end for
  for all $(t_i, s_x) \in \wp$ **do** ▷ Iterate through $\wp$
    if $\forall (t_j, t_i) \in E \cdot t_j \in T_{\text{comp}}$ **then** ▷ Is t_i executable?
      INVOKE(s_x, t_i) ▷ Invoke service s_x for task t_i
      $T_{\text{inv}} \leftarrow T_{\text{inv}} \cup \{(t_i, \hat{t})\}$ ▷ Store invocation time
      $\wp \leftarrow \wp \setminus \{(t_i, s_x)\}$ ▷ Remove for now
    end if
  end for
end procedure

procedure SERIAL-PROVISION(i)
  if $S_i \neq \emptyset$ **then**
    $s_x \in S_i$ ▷ Select random service
    $S_i \leftarrow S_i \setminus \{s_x\}$ ▷ Remove from available services
    $\wp \leftarrow \wp \cup \{(t_i, s_x)\}$ ▷ Store provision decision for t_i
  end if
end procedure

The overall success probability of this approach is the same as that of *serial* provisioning given in Equation (4.2) (when using identical time-out values). This is because although some services are executed in parallel, their individual success probabilities are unchanged. However, this hybrid approach allows the consumer to achieve lower execution times than the *serial* strategy at the expense of incurring higher invocation costs.

We define this hybrid strategy as follows:

Definition 5 (Hybrid(n,w) Strategy)**.** A consumer following a ***hybrid(n,w)*** strategy always provisions exactly n randomly chosen services of the correct type for each task (or as many as available if these are less than n). After a waiting period of w time units, if no success has been registered yet and if there is still at least one available service, the agent repeats this process with a new set of n services until the task is completed or until no more services are left (both w and n are constants that apply similarly to all tasks).

The *hybrid* strategy is formalised by Algorithm 4.4, which follows closely the structure of the previous algorithms, combining the time-out mechanism of the *serial* strategy with the multiple provisions of the *parallel* strategy. In fact, the *hybrid* strategy subsumes all previous strategies (*naïve*, *parallel* and *serial*). In so doing, it addresses two of our main original requirements, A.2.a and A.2.b. However, it has several shortcomings that make it less useful for automating the provisioning of complex workflows:

1. The choice of n and w are probably critical to the performance of the strategy. When services are generally unreliable, a high n might be called for, while the choice of w depends on the duration distributions of services and the deadline of the workflow. Currently, this choice needs to be taken by the owner of the consumer agent, hence shifting the burden of making rational decisions to a human user. In dynamic environments, with thousands of services and complex workflows, the choice of n and w will be time-intensive and not trivial. Hence, such manual intervention is highly undesirable and detracts from our Requirement A.1 for building an agent that takes rational decisions on behalf of the user.

2. Currently, specifying n and w as global constants leads to a highly constrained decision space. In realistic application scenarios, it is likely that some services will be fast and reliable (e.g., a DNS lookup request taking a fraction of a second), while others could be time-consuming and unreliable (e.g., running an enzyme folding simulation on an idle workstation for several hours). In such scenarios, where services are highly variable, specifying global values for n and w will be unsatisfactory because some services may benefit more from over-provisioning (higher n) than others and because the time taken for some services will be fundamentally different from others (and hence require different values for w).

Algorithm 4.4 *Hybrid(n,w)* strategy that provision n parallel services and re-provisions unsuccessful tasks after w time units.

procedure HYBRID-INITIALISE(W)
  $n \leftarrow$ constant specified by agent owner ▷ Number of parallel services
  $w \leftarrow$ constant specified by agent owner ▷ Time-out for re-provisioning
  $T_{\text{inv}} \leftarrow \emptyset$ ▷ Keeps track of invocation times
  $T_{\text{comp}} \leftarrow \emptyset$ ▷ Keeps track of completed tasks
  $\wp \leftarrow \emptyset$ ▷ Provisioning decisions
  for all $t_i \in T$ **do** ▷ Iterate through tasks
    $S_i \leftarrow \mu(\tau(t_i))$ ▷ Matchmaking
    PARALLEL-PROVISION(i, n) ▷ Provision (see Algorithm 4.2)
  end for
end procedure

procedure HYBRID-UPDATE($\mathcal{O}$)
  $T_{\text{new}} \leftarrow \{t_i \mid \exists s_x \cdot (t_i, s_x) \in \mathcal{O}\}$ ▷ Recently completed tasks
  $T_{\text{comp}} \leftarrow T_{\text{comp}} \cup T_{\text{new}}$ ▷ Add to completed tasks
  $T_{\text{inv}} \leftarrow \{(t_i, y) \mid (t_i, y) \in T_{\text{inv}} \wedge t_i \notin T_{\text{new}}\}$ ▷ Remove completed from T_{inv}
end procedure

procedure HYBRID-INVOKESERVICES
  for all $(t_i, y) \in T_{\text{inv}}$ **do** ▷ Check for timed out tasks
    if $\hat{t} - y \geq w$ **then** ▷ Invoked at least w time steps earlier?
      $T_{\text{inv}} \leftarrow T_{\text{inv}} \setminus \{(t_i, y)\}$ ▷ Timed out
      PARALLEL-PROVISION(i, n) ▷ Re-provision (see Algorithm 4.2)
    end if
  end for
  for all $(t_i, \hat{S}_x) \in \wp$ **do** ▷ Iterate through $\wp$
    if $\forall (t_j, t_i) \in E \cdot t_j \in T_{\text{comp}}$ **then** ▷ Is t_i executable?
      for all $s_y \in \hat{S}_x$ **do** ▷ Iterate through services in $\hat{S}_x$
        INVOKE(s_y, t_i) ▷ Invoke service s_y for task t_i
      end for
      $T_{\text{inv}} \leftarrow T_{\text{inv}} \cup \{(t_i, \hat{t})\}$ ▷ Store invocation time
      $\wp \leftarrow \wp \setminus \{(t_i, \hat{S}_x)\}$ ▷ Remove to avoid re-invocation
    end if
  end for
end procedure

To address these two critical shortcomings, in the following section, we develop a novel strategy that provisions multiple services for tasks in a flexible manner. This approach takes into consideration the performance characteristics of services and the structure of the workflow and then provisions services based on a heuristic approach.

4.4 Flexible Service Provisioning

Building on the techniques presented in the previous section, we now introduce a novel algorithm for flexibly provisioning services that are part of complex workflows. Unlike our previous strategies, this approach determines automatically how many services to invoke in parallel and it also chooses an appropriate time-out value. It does this by considering a more finely-grained decision problem than thus far considered and by using some information about the expected performance of services in the system. This approach allows the agent to vary its strategy according to the current system conditions, without the need for human intervention. In devising the strategy we address not only the requirements covered in previous sections (A.2.a and A.2.b), but also consider the need for making rational decisions (A.1). We also show that our approach is suitable for larger environments and workflows (A.3).

Due to its autonomous decision-making process that adjusts the agent's behaviour to its environment, we term this approach the *flexible* strategy and summarise it as follows:

Definition 6 (Flexible Strategy)**.** A consumer following a ***flexible*** strategy makes appropriate decisions to provision services for its workflow. To this end, the agent finds suitable numbers of providers and time-out values for each task in the workflow, so that the agent's predicted profit is maximised.

We begin this section by describing our problem as an optimisation task (Section 4.4.1). As solving this turns out to be intractable in practice, we then provide a heuristic approach for provisioning services (Section 4.4.2).

4.4.1 Problem Formulation

To address the shortcomings of the *hybrid(n,w)* strategy, outlined in Section 4.3.3, we first formulate a more fine-grained decision problem than so far considered. Instead of choosing global values for n and w, as in the *hybrid* approach, we define them as vectors, $\vec{n}$ and $\vec{w}$, corresponding to the tasks in the workflow. In this notation, the ith element of vector $\vec{n}$, n_i, is the number of services to be invoked for task t_i. Similarly, w_i is the associated time-out value, indicating how long the consumer will wait before invoking another set of n_i services for task t_i.

Now, we are interested in choosing vectors $\vec{n}$ and $\vec{w}$, so that the expected profit $\bar{u}(\vec{n}, \vec{w})$ is maximised (the profit is the difference between the reward gained from completing the workflow

and the costs incurred from all service invocations). More formally, we let $\bar{u_t}(\vec{n}, \vec{w})$ be the expected reward and $\bar{c}(\vec{n}, \vec{w})$ the expected cost. Then we can define the expected profit as:

$$\bar{u}(\vec{n}, \vec{w}) = \bar{u}_t(\vec{n}, \vec{w}) - \bar{c}(\vec{n}, \vec{w}) \tag{4.3}$$

With this, we can specify the service provisioning problem as an optimisation task:

$$\max_{\vec{n},\vec{w}\in\mathbb{N}^{|T|}} \quad \bar{u}(\vec{n}, \vec{w}) \tag{4.4}$$

However, finding a solution for this optimisation problem is far from easy. Simply verifying a possible solution, i.e., computing the expected profit $\bar{u}(\vec{n}, \vec{w})$ for given vectors $\vec{n}$ and $\vec{w}$ is very hard. This is because calculating the distribution of the workflow completion time (needed for $\bar{u_t}$) involves the convolution of several probability functions (the duration functions given by $\tilde{d}$), which is further complicated by the fact that there are usually interdependencies between the task completion times (as tasks in the workflow depend on their predecessors). In fact, there is currently no known efficient method[4] to solve this problem exactly for arbitrary distributions (Dodin (1985); Baccelli et al. (1993)).

For this reason, we decided to simplify the problem and devise a heuristic algorithm that sacrifices theoretical optimality in favour of a tractable decision algorithm that produces good results in practice. In particular, we employ a heuristic function for estimating the expected profit, $\tilde{u}(\vec{n}, \vec{w})$. Despite this simplification, we are still faced with the difficult nonlinear integer programming problem of optimising $\tilde{u}(\vec{n}, \vec{w})$. To address this, we find a good allocation for $\vec{n}$ and $\vec{w}$ by carrying out steepest-ascent hill-climbing (Russell and Norvig (2003)), as described in the following section.

4.4.2 Generic Algorithm for Flexible Service Provisioning

We decided to use a local search algorithm to find a good allocation, because this technique is widely employed for intractable optimisation problems (Michalewicz and Fogel (2004)). While the particular method of local search is not central to our work, we carried out experiments with a range of existing algorithms, including steepest-ascent hill-climbing, simple hill-climbing (where the first better solution is chosen at each iteration), hill-climbing with random restarts and simulated annealing. We found that these techniques achieved a similar average profit over a range of environments, but varied in the number of required parameters (e.g., the annealing temperature or the number of restarts). As steepest-ascent hill-climbing required no such parameters and as we observed a generally faster convergence to an optimum compared to simple hill-climbing, we decided to adopt this approach in our work.

[4] We consider the hardness of the provisioning problem more formally in Appendix C.

Algorithm 4.5 formalises our hill-climbing approach. It starts by generating a random initial provisioning allocation, using the procedure GENERATE-INITIAL shown in lines 19 – 27. For each task, this procedure first randomly picks the number of parallel services (n_i) from the distribution $\mathcal{U}_d(1, \min(v_i, \varphi_i))$, where we use $\mathcal{U}_d(a, b)$ to denote a discrete uniform distribution over all integers in the interval $[a, b]$, with φ_i chosen so that it represents the smallest number of services required to ensure a success probability of at least 0.999 (based on Equation 4.1). When this is less than 10, we set $\varphi_i = 10$, to include the possibility that services are provisioned redundantly. Choosing n_i in this way means that we do not start with an unnecessarily high number of services when there are many available in the system. Next, the procedure determines an initial waiting time for each task (w_i) by sampling from the respective duration distribution d_i.

Given this initial allocation for $\vec{n}$ and $\vec{w}$, the algorithm next estimates its expected utility using the heuristic function $\tilde{u}$ (line 3), which we will cover in detail in Section 4.4.3. Then, it begins to iteratively improve the initial allocation in the loop given in lines 4 – 16. Here, the algorithm first generates a random set of neighbours of the current allocation using the procedure GENERATE-NEIGHBOURS (lines 28 – 43). These neighbours are generated by increasing and decreasing exactly one component of either vector — both in unit steps and in random[5] steps, as shown in lines 32 – 39 (ignoring any neighbours with $n_{x,i} < 1$ or $n_{x,i} > v_i$). We chose this particular neighbour generation function, because it allows the algorithm to explore close neighbours (by varying the parameters in unit steps, which usually results in only small changes in the expected utility), but also because it quickly traverses larger parts of the search space when necessary (by considering random steps). The algorithm then estimates the expected utility of each of the generated neighbours and adopts the best of these as the current solution. This continues until no generated neighbour of the current solution results in a higher expected utility.

This hill-climbing procedure is placed in the context of our abstract consumer model in Algorithm 4.6. It is called in the procedure FLEXIBLE-INITIALISE, where it provisions all tasks in the workflow (line 5). In line 6 the agent then decides if it should proceed with the workflow, which depends on whether it expects to gain a positive utility (the decision d_{stop} is used by the updated FLEXIBLE-STOPCONDITION procedure in lines 21 – 23). The remainder of the algorithm is mostly identical to the *hybrid(n,w)* strategy, with the exception that the agent now uses the vectors $\vec{n}$ and $\vec{w}$ to guide its provisioning decisions, rather than globally set constants. Specifically, if the allocation is promising, the agent proceeds to discover services and provision them in lines 10 – 13. These are then invoked in the updated FLEXIBLE-INVOKESERVICES procedure, which now uses $\vec{w}$ and $\vec{n}$ to time out tasks and to re-provision.

So far, we have given an algorithm for a flexible service consumer based on a hill-climbing approach. However, we have not discussed the heuristic utility function, $\tilde{u}(\vec{n}, \vec{w})$, which is central to the algorithm. This shortcoming is addressed in the following section.

[5]For n_i, we select the change from uniform distributions. For w_i, we sample from the duration distribution (using the inverse of the cumulative density function, D_i^{-1}, to generate a sample that is larger and one that is smaller than the current waiting time).

Algorithm 4.5 Hill-climbing algorithm for provisioning services.

procedure FIND-ALLOCATION(W)
  $\vec{n}, \vec{w} \leftarrow$ GENERATE-INITIAL(W) ▷ Generate initial allocation
  $u \leftarrow \tilde{u}(\vec{n}, \vec{w})$ ▷ Estimate utility
  repeat ▷ Main hill-climbing loop
    $u_{\text{old}} \leftarrow u$ ▷ Store utility at start of iteration
    $\vec{n}^*, \vec{w}^* \leftarrow \vec{n}, \vec{w}$ ▷ Best neighbour found so far
    $\mathcal{N} \leftarrow$GENERATE-NEIGHBOURS($\vec{n}, \vec{w}$) ▷ Randomly generate neighbours
    for all $(\vec{n}', \vec{w}') \in \mathcal{N}$ **do** ▷ Check all neighbours
      $u' \leftarrow \tilde{u}(\vec{n}', \vec{w}')$ ▷ Utility of neighbour
      if $u' > u$ **then** ▷ If neighbour is more promising...
        $u \leftarrow u'$ ▷ ...update
        $\vec{n}^*, \vec{w}^* \leftarrow \vec{n}', \vec{w}'$
      end if
    end for
    $\vec{n}, \vec{w} \leftarrow \vec{n}^*, \vec{w}^*$ ▷ Choose best overall neighbour
  until $u = u_{\text{old}}$ ▷ Until no more improvement is made
  return $(\vec{n}, \vec{w})$ ▷ Return best allocation found
end procedure

procedure GENERATE-INITIAL(W)
  $\vec{n}, \vec{w} \leftarrow$ vectors of size $|T|$
  for $i = 1$ to $|T|$ **do**
    $\varphi_i \leftarrow \max(10, \lceil -3/\log_{10}(f_i) \rceil)$ ▷ Maximum number of initial services
    $n_i \leftarrow$ sample from $\mathcal{U}_d(1, \min(v_i, \varphi_i))$ ▷ Random number of providers
    $w_i \leftarrow$ sample from d_i ▷ Random waiting time
  end for
  return $\vec{n}, \vec{w}$
end procedure

procedure GENERATE-NEIGHBOURS($\vec{n}, \vec{w}$)
  $\mathcal{N} \leftarrow \emptyset$ ▷ Set of neighbours
  for $i = 1$ to $|T|$ **do** ▷ Consider each task
    $(\vec{n}_1, \vec{w}_1), \ldots, (\vec{n}_8, \vec{w}_8) \leftarrow (\vec{n}, \vec{w})$ ▷ Create eight copies
    $n_{1,i} \leftarrow n_i + 1$ ▷ Now slightly modify each copy
    $n_{2,i} \leftarrow n_i - 1$ ▷ $n_{j,i}$ is the ith component of $\vec{n}_j$
    $w_{3,i} \leftarrow w_i + 1$ ▷ $w_{j,i}$ is the ith component of $\vec{w}_j$
    $w_{4,i} \leftarrow w_i - 1$
    $n_{5,i} \leftarrow n_i + x$, where x is sampled from $\mathcal{U}_d(2, v_i - n_i)$
    $n_{6,i} \leftarrow n_i - x$, where x is sampled from $\mathcal{U}_d(2, n_i - 1)$
    $w_{7,i} \leftarrow \left\lceil D_i^{-1}(x) \right\rceil$, where x is sampled from $\mathcal{U}_c(D_i(w_i), 1)$
    $w_{8,i} \leftarrow \left\lceil D_i^{-1}(x) \right\rceil$, where x is sampled from $\mathcal{U}(0, D_i(w_i))$
    $\mathcal{N} \leftarrow \mathcal{N} \cup \{(\vec{n}_1, \vec{w}_1), \ldots, (\vec{n}_8, \vec{w}_8)\}$ ▷ Add copies to $\mathcal{N}$
  end for
  return $\mathcal{N}$
end procedure

Algorithm 4.6 *Flexible* strategy that provisions services based on a heuristic function.

procedure FLEXIBLE-INITIALISE(W)
  $T_{\text{inv}} \leftarrow \emptyset$ ▷ Keeps track of invocation times
  $T_{\text{comp}} \leftarrow \emptyset$ ▷ Keeps track of completed tasks
  $\wp \leftarrow \emptyset$ ▷ Provisioning decisions
  $\vec{n}, \vec{w} \leftarrow$ FIND-ALLOCATION(M) ▷ Find best allocation
  if $\tilde{u}(\vec{n}, \vec{w}) \leq 0$ **then** ▷ Is utility estimate non-positive?
    $d_{\text{stop}} \leftarrow$ `true` ▷ ...then abandon workflow
  else ▷ ...otherwise continue
    $d_{\text{stop}} \leftarrow$ `false`
    for all $t_i \in T$ **do** ▷ Iterate through tasks
      $S_i \leftarrow \mu(\tau(t_i))$ ▷ Matchmaking
      PARALLEL-PROVISION(i, n_i) ▷ Provision (see Algorithm 4.2)
    end for
  end if
end procedure

procedure FLEXIBLE-UPDATE($\mathcal{O}$)
  $T_{\text{new}} \leftarrow \{t_i \mid \exists s_x \cdot (t_i, s_x) \in \mathcal{O}\}$ ▷ Recently completed tasks
  $T_{\text{comp}} \leftarrow T_{\text{comp}} \cup T_{\text{new}}$ ▷ Add to completed tasks
  $T_{\text{inv}} \leftarrow \{(t_i, y) \mid (t_i, y) \in T_{\text{inv}} \wedge t_i \notin T_{\text{new}}\}$ ▷ Remove completed from T_{inv}
end procedure

procedure FLEXIBLE-STOPCONDITION
  return d_{stop} ▷ Abandon if allocation yields non-positive utility
end procedure

procedure FLEXIBLE-INVOKESERVICES
  for all $(t_i, y) \in T_{\text{inv}}$ **do** ▷ Check for timed out tasks
    if $\hat{t} - y \geq w_i$ **then** ▷ Invoked at least w_i time steps earlier?
      $T_{\text{inv}} \leftarrow T_{\text{inv}} \setminus \{(t_i, y)\}$ ▷ Timed out
      PARALLEL-PROVISION(i, n_i) ▷ Re-provision (see Algorithm 4.2)
    end if
  end for
  for all $(t_i, \hat{S}_x) \in \wp$ **do** ▷ Iterate through $\wp$
    if $\forall (t_j, t_i) \in E \cdot t_j \in T_{\text{comp}}$ **then** ▷ Is t_i executable?
      for all $s_y \in \hat{S}_x$ **do** ▷ Iterate through services in $\hat{S}_x$
        INVOKE(s_y, t_i) ▷ Invoke service s_y for task t_i
      end for
      $T_{\text{inv}} \leftarrow T_{\text{inv}} \cup \{(t_i, \hat{t})\}$ ▷ Store invocation time
      $\wp \leftarrow \wp \setminus \{(t_i, \hat{S}_x)\}$ ▷ Remove to avoid re-invocation
    end if
  end for
end procedure

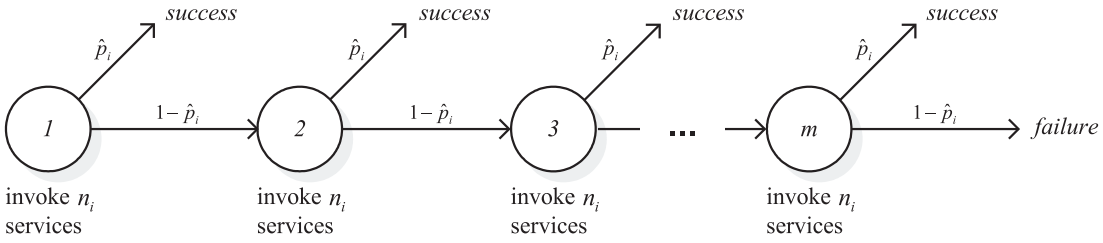


FIGURE 4.2: Possible state transitions as consumer invokes services in sequence.

4.4.3 Utility Prediction

As discussed above, we use a heuristic function, $\tilde{u}(\vec{n}, \vec{w})$, to approximate the expected utility of an allocation. We need such an approximation due to the inherent difficulty of calculating the distribution of the workflow completion time. Based closely on Equation 4.3, we define the heuristic utility function as (omitting the parameters for brevity):

$$\tilde{u} = \tilde{r} - \tilde{c} \tag{4.5}$$

Here, $\tilde{r}$ and $\tilde{c}$ are estimates of the expected reward and cost of the allocation, respectively (both unconditional on overall success of the workflow). In the following, we describe how these estimates are calculated from a number of parameters for the individual tasks — the success probability p_i, expected cost $\bar{c}_i$, expected completion time $\bar{t}_i$ and variance σ_i^2. First, in Section 4.4.3.1, we outline how the parameters are calculated for each task t_i. Then, in Section 4.4.3.2, we show how these calculations are used to derive the overall values for $\tilde{r}$ and $\tilde{c}$.

4.4.3.1 Local Prediction

Given the probabilistic information about service instances discussed in Section 4.1 and an allocation, (n_i, w_i), we begin by calculating the success probability of a task, p_i. This is the overall probability that the task will eventually be successfully completed when following the allocation (n_i, w_i). This does not depend on n_i, because it is irrelevant for the success probability whether services are invoked in series or in parallel. Hence, we use Equation 4.2 to determine p_i as follows:

$$p_i = 1 - (1 - (1 - f_i) \cdot D_i(w_i))^{v_i} \tag{4.6}$$

Next, we calculate the expected cost $\bar{c}_i$, which depends on the expected number of invocations that are carried out for the task, before it is successful. To illustrate this, Figure 4.2 shows the possible state transitions of a service-consuming agent. In state 1, the agent invokes the first set of n_i services. With probability $\hat{p}_i = 1 - (1 - (1 - f_i) \cdot D_i(w_i))^{n_i}$ at least one of these is successful, but with probability $1-\hat{p}_i$ none of them will succeed. In the latter case, the consumer then invokes a new set of n_i services (in state 2). This process repeats until one invocation is

successful or no more services are available (for now, we assume that $v_i \bmod n_i = 0$, so that there are up to $m = v_i/n_i$ invocations of exactly n_i services each).

We note from this diagram that the consumer is guaranteed to pay the full cost of invoking all n_i services for task t_i ($n_i c_i$) at least once. After this, the consumer generally has to pay again if the previously invoked set of services has failed (each with probability $1 - \hat{p}_i$). Formally, we let $\hat{f}_i = 1 - \hat{p}_i$ and give the expected cost for task t_i as follows[6]:

$$\bar{c}_i = \underbrace{n_i c_i + \hat{f}_i \cdot \left(n_i c_i + \hat{f}_i \cdot \left(n_i c_i + \hat{f}_i \cdot \left(\ldots + \hat{f}_i \cdot (n_i c_i) \ldots \right) \right) \right)}_{m \text{ instances of } n_i c_i} \tag{4.7}$$

$$= n_i c_i \cdot \sum_{k=0}^{m-1} \hat{f}_i^k \tag{4.8}$$

$$= n_i c_i \cdot \frac{1 - \hat{f}_i^m}{1 - \hat{f}_i} \tag{4.9}$$

Equation 4.9 is the expected cost for task t_i, assuming that $v_i \bmod n_i = 0$. To generalise this result for cases where $v_i \bmod n_i \neq 0$, we note that the consumer will invoke all remaining services on its last try. For this case, we let $m = \lfloor v_i/n_i \rfloor$ be the number of full invocations (n_i services each) and $r = v_i \bmod n_i$ be the remaining number of services after m invocations. Then, the consumer will pay $c_r = c_i r$ for the last invocation if all previous services have failed (which happens with probability $\hat{f}_i^m$). To generalise Equation 4.9, we simply include this cost:

$$\bar{c}_i = n_i c_i \cdot \frac{1 - \hat{f}_i^m}{1 - \hat{f}_i} + \hat{f}_i^m c_i r \tag{4.10}$$

Next, we are interested in calculating the expected time $\bar{t}_i$ until the task is completed. We define this as the mean time until the first service completes the task successfully, conditional on overall success (i.e., that at least one service is successful). First, we let μ_i be the mean duration of a single invocation, conditional on overall success. In other words, given that n_i services are invoked and that at least one completes successfully before time-out w_i, μ_i is the expected duration of the fastest successful service.

To calculate μ_i, we first let $\hat{D}_i(x)$ be the cumulative (non-conditional) probability that at least one out of n_i services has finished successfully by time x:

$$\hat{D}_i(x) = 1 - (1 - (1 - f_i) \cdot D_i(x))^{n_i} \tag{4.11}$$

[6]For the sake of readability, we do not provide a full derivation here, but rather refer the reader to Appendix D. This appendix contains a more thorough treatment of this and other equations throughout the thesis (we will indicate this in the text where appropriate).

With this, we calculate μ_i as follows:

$$\mu_i = \frac{1}{\hat{D}_i(w_i)} \sum_{k=1}^{w_i} k \cdot \left(\hat{D}_i(k) - \hat{D}_i(k-1) \right) \tag{4.12}$$

Now, to calculate the overall expected time of the task, we again assume that $v_i \bmod n_i = 0$ and follow similar reasoning as for the expected cost by considering Figure 4.2. When the consumer succeeds after state 1, its expected duration is then μ_i, and if it succeeds after state 2, the expected duration is $w_i + \mu_i$. We can formulate the general case, after the kth invocation as:

$$\bar{d}_k = (k-1) \cdot w_i + \mu_i \tag{4.13}$$

The associated non-conditional probability of this event (succeeding after the kth invocation) is $\hat{f}_i^{k-1} \left(1 - \hat{f}_i\right)$. Using this, and conditioning on an overall success, we can now write the expected time for task t_i as[7]:

$$\begin{aligned} \bar{t}_i &= \frac{1}{p_i} \cdot \sum_{k=1}^{m} \bar{d}_k \hat{f}_i^{k-1} \left(1 - \hat{f}_i\right) \\ &= \frac{1}{p_i} \cdot \sum_{k=0}^{m-1} (k \cdot w_i + \mu_i) \cdot \hat{f}_i^{k} \left(1 - \hat{f}_i\right) \\ &= \frac{1}{p_i} \cdot \left(\mu_i (1 - \hat{f}_i^m) + w_i \frac{\hat{f}_i - m\hat{f}_i^m + (m-1)\hat{f}_i^{m+1}}{1 - \hat{f}_i} \right) \end{aligned} \tag{4.14}$$

To generalise this when $v_i \bmod n_i \neq 0$, we again let $m = \lfloor v_i / n_i \rfloor$ be the number of full invocations and $r = v_i \bmod n_i$ the remaining services. We also let λ_i be the mean duration to the first success when r services are invoked (calculated analogously to μ_i in Equation 4.12), and we let $\check{f}_r$ be the probability of failure when invoking r services in parallel. Then we can add the impact of the remaining services to extend Equation 4.14:

$$\bar{t}_i = \frac{1}{p_i} \left(\mu_i (1 - \hat{f}_i^m) + w_i \frac{\hat{f}_i - m\hat{f}_i^m + (m-1)\hat{f}_i^{m+1}}{1 - \hat{f}_i} + \hat{f}_i^m (1 - \check{f}_r)(\lambda_i + m w_i) \right) \tag{4.15}$$

Finally, to calculate the variance, σ_i^2, of the task, we let C_i be a random variable representing the duration of the task, conditional on its success (note, its expected value, $\mathrm{E}(C_i)$, is equal to $\bar{t}_i$). We are interested in the variance of this variable, $\mathrm{VAR}(C_i)$, which we calculate as follows:

$$\begin{aligned} \sigma_i^2 &= \mathrm{VAR}(C_i) \\ &= \mathrm{E}(C_i^2) - \mathrm{E}(C_i)^2 \end{aligned} \tag{4.16}$$

[7] See Appendix D for a detailed derivation.

We can calculate $\mathrm{E}(C_i)^2$ as given by Equation 4.15, but to calculate $\mathrm{E}(C_i^2)$, further steps are necessary. First, we consider two cases, as before: (1) the task is successful during the first $m = \lfloor v_i/n_i \rfloor$ full invocations, and (2) the task is successful in the last invocation with $r = v_i \bmod n_i$ parallel services (if $r \neq 0$). We use two random variables to denote the durations in each case — A_i and B_i, respectively (again, these are conditional on the task being successful in each case). In order to treat both cases separately, we can now re-write $\mathrm{E}(C_i^2)$, letting P_A be the probability that case (1) occurs, and P_B the probability that case (2) occurs, both conditional on overall success:

$$\begin{aligned}\mathrm{E}(C_i^2) &= P_A\mathrm{E}(A_i^2) + P_B\mathrm{E}(B_i^2) \\ &= \frac{1-\hat{f}_i^m}{1-\check{f}_r\hat{f}_i^m}\mathrm{E}(A_i^2) + \frac{\hat{f}_i^m(1-\check{f}_r)}{1-\check{f}_r\hat{f}_i^m}\mathrm{E}(B_i^2) \end{aligned} \tag{4.17}$$

Furthermore, we separate each of these durations into the total time spent waiting for unsuccessful invocations that are timed-out (we denote these as A_{Wi} and B_{Wi}) and the time that passes during the last invocation before the first service is successful (denoted as A_{Di} and B_{Di}), and we note that these two components are independent of each other in our model. Beginning with the first case, we thus write:

$$\begin{aligned}\mathrm{E}(A_i^2) &= \mathrm{VAR}(A_i) + \mathrm{E}(A_i)^2 \\ &= \mathrm{VAR}(A_{Wi}) + \mathrm{VAR}(A_{Di}) + (\mathrm{E}(A_{Wi}) + \mathrm{E}(A_{Di}))^2 \\ &= \mathrm{E}(A_{Wi}^2) - \mathrm{E}(A_{Wi})^2 + \mathrm{E}(A_{Di}^2) - \mathrm{E}(A_{Di})^2 + (\mathrm{E}(A_{Wi}) + \mathrm{E}(A_{Di}))^2 \\ &= \mathrm{E}(A_{Wi}^2) + \mathrm{E}(A_{Di}^2) + 2\mathrm{E}(A_{Wi})\mathrm{E}(A_{Di}) \end{aligned} \tag{4.18}$$

The expected duration of a single invocation, $\mathrm{E}(A_{Di})$, is equal to μ_i, which we calculate using Equation 4.12. The expected squared duration, $\mathrm{E}(A_{Di}^2)$, is similarly calculated by multiplying the term inside the summation by k^2 instead of k. The expected waiting time, $\mathrm{E}(A_{Wi})$, is obtained from Equation 4.14:

$$\mathrm{E}(A_{Wi}) = \frac{w_i}{(1-\hat{f}_i)(1-\hat{f}_i^m)}(\hat{f}_i - m\hat{f}_i^m + (m-1)\hat{f}_i^{m+1}) \tag{4.19}$$

To derive[8] the expected squared waiting time, $\mathrm{E}(A_{Wi}^2)$, we follow similar reasoning as for Equation 4.14:

$$\begin{aligned}\mathrm{E}(A_{Wi}^2) &= \frac{(1-\hat{f}_i)w_i^2}{1-\hat{f}_i^m}\sum_{k=0}^{m-1} k^2\hat{f}_i^k \\ &= \frac{w_i^2}{(1-\hat{f}_i^m)(1-\hat{f}_i)^2}(\hat{f}_i + \hat{f}_i^2 - m^2\hat{f}_i^m - \\ &\quad (2m+1-2m^2)\hat{f}_i^{m+1} + (2m-1-m^2)\hat{f}_i^{m+2}) \end{aligned} \tag{4.20}$$

[8] See Appendix D for a detailed derivation.

Next, when $v_i \bmod n_i \neq 0$, we also need to calculate the expected squared duration if the consumer is successful on the last invocation, $\mathrm{E}(B_i^2)$. This is done analogously to Equation 4.18, simplified by the fact that a constant waiting time (mw_i) is associated with the last invocation:

$$\begin{aligned} \mathrm{E}(B_i^2) &= \mathrm{VAR}(B_i) + \mathrm{E}(B_i)^2 = \mathrm{E}(B_{Wi}^2) + \mathrm{E}(B_{Di}^2) + 2\mathrm{E}(B_{Wi})\mathrm{E}(B_{Di}) \\ &= (mw_i)^2 + \mathrm{E}(B_{Di}^2) + 2mw_i\mathrm{E}(B_{Di}) \end{aligned} \tag{4.21}$$

The remaining terms, $\mathrm{E}(B_{Di})$ and $\mathrm{E}(B_{Di}^2)$, are calculated as $\mathrm{E}(A_{Di})$ and $\mathrm{E}(A_{Di}^2)$, discussed above.

We have now finished analysing the performance characteristics of a single task t_i given an allocation (n_i, w_i) and some knowledge about the services available for the task. In particular, we can calculate the success probability of the task (p_i in Equation 4.6), the expected cost of attempting the task ($\bar{c}_i$ in Equation 4.10), the expected completion time of the task, conditional on its success ($\bar{t}_i$ in Equation 4.15), and its variance (σ^2 in Equation 4.16). Given these calculations for each task, we are now interested in estimating the expected reward $\tilde{r}$ and the expected cost $\tilde{c}$ for the overall workflow, which are required for our heuristic utility function given in Equation 4.5.

4.4.3.2 Global Prediction

We now calculate a number of global performance characteristics for the overall workflow, beginning with the estimated total cost, $\tilde{c}$. This is the sum of all task costs, each multiplied by the respective success probabilities of their predecessors in the workflow (where r_i is the probability that task t_i is ever reached):

$$\tilde{c} = \sum_{\{i|t_i \in T\}} r_i \bar{c}_i \tag{4.22}$$

$$r_i = \begin{cases} 1 & \text{if } \forall t_j \cdot ((t_j, t_i) \notin E) \\ \prod_{\{j|(t_j,t_i)\in E\}} p_j & \text{otherwise} \end{cases} \tag{4.23}$$

Next, to estimate the expected reward of the allocation, we need a duration distribution for the complete workflow (again, conditional on overall success). To this end, we employ a technique from operations research (Malcolm et al. (1959)), and evaluate the *critical path* of the workflow (i.e., the path that maximises the sum of all mean task durations along it). To obtain an estimated distribution for the duration of this path, we 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. This approach exploits the central limit theorem, which states that the sum of arbitrary independent random variables can be approximated using such a distribution[9]. Hence, the corresponding probability density function for the workflow duration

[9]This theorem holds when the number of variables approaches infinity and makes some assumptions about the variables, e.g., that their third moments must be finite (DeGroot and Shervish (2002)). However, we have verified that this approximation works well in practice, even when considering small workflows (see Section 4.5.7).

is:

$$d_W(x) = \frac{1}{\sqrt{v_W 2\pi}} \, \mathrm{e}^{-\frac{(x-\lambda_W)^2}{2v_W}} \tag{4.24}$$

with

$$\lambda_W = \sum_{\{i|t_i \in \mathcal{C}\}} \bar{t}_i \tag{4.25}$$

$$v_W = \sum_{\{i|t_i \in \mathcal{C}\}} \sigma_i^2 \tag{4.26}$$

where $\mathcal{C}$ is the set of tasks on the critical path.

Next, we use the distribution $d_W(x)$ to estimate the expected reward of the allocation. In so doing, we assume that workflow finishing times can be continuous, in order to derive a closed, analytical solution. This introduces a small error in the results, as the time model we employ is actually discrete. However, we believe this error is negligible, as time steps will typically be small compared to the overall workflow duration (and our experimental results support this). To this end, we assume overall success and denote the corresponding expected reward with $\tilde{r}_s$:

$$\tilde{r}_s = \int_0^\infty d_W(x) u(x) \, \mathrm{d}x \tag{4.27}$$

In order to calculate this, we let $D_W(x) = \int_{-\infty}^{x} d_W(y) \, \mathrm{d}y$ be the cumulative probability function[10] of $d_W(x)$, we let $D_{\text{max}} = D_W(t_{\text{max}})$ be the probability that the workflow will finish no later than the deadline t_{max} and $D_{\text{late}} = D_W(t_{\text{last}}) - D_W(t_{\text{max}})$ the probability that the workflow will finish after the deadline but no later than time $t_{\text{last}} = \frac{u_{\text{max}}}{\delta} + t_{\text{max}}$ (both conditional on overall success).

Next, we consider three distinct cases, as derived from Equation 3.2 for $u(t)$. First, the workflow may finish within the deadline t_{max} — in this case, which happens with probability D_{max}, the consumer will receive the full reward, u_{max}. Second, the workflow may finish after t_{last} — this happens with probability $1 - D_W(t_{\text{last}})$, and here the consumer receives no reward (and so we can ignore it). Finally, the workflow may finish between these two times, which happens with probability D_{late}. Because $u(t)$ is linear on this interval, we can calculate the expected reward in this case by applying $u(t)$ to the mean time on the interval, which we denote by $\bar{t}_{\text{late}}$. Hence, we can re-write Equation 4.27:

$$\tilde{r}_s = D_{\text{max}} \cdot u_{\text{max}} + D_{\text{late}} \cdot u(\bar{t}_{\text{late}}) \tag{4.28}$$

[10] This is a common function that is usually approximated numerically. In our implementation, we use the SSJ library (`http://www.iro.umontreal.ca/~simardr/ssj`).

Now, we calculate $\bar{t}_{\text{late}}$:

$$\begin{aligned} \bar{t}_{\text{late}} &= \frac{1}{D_{\text{late}}} \int_{t_{\text{max}}}^{t_{\text{last}}} d_W(x) x \, \mathrm{d}x \\ &= \lambda_W + (e^{\frac{-(t_{\text{max}}-\lambda_W)^2}{2v_W}} - e^{\frac{-(t_{\text{last}}-\lambda_W)^2}{2v_W}}) \frac{\sqrt{v_W}}{D_{\text{late}} \cdot \sqrt{2\pi}} \end{aligned} \tag{4.29}$$

Finally, this reward ($\tilde{r}_s$) is only obtained when the workflow is successful. Hence, we calculate the overall probability of success, p, as the product of all p_i:

$$p = \prod_{\{i|t_i \in T\}} p_i \tag{4.30}$$

This allows us to summarise our heuristic utility function as follows:

$$\tilde{u} = p \cdot (D_{\text{max}} \cdot u_{\text{max}} + D_{\text{late}} \cdot u(\bar{t}_{\text{late}})) - \tilde{c} \tag{4.31}$$

Using this heuristic function, it is now possible to use steepest-ascent hill-climbing as described at the beginning of this section. Through observations, we have seen that our hill-climbing algorithm quickly converges to a good solution[11]. In particular, the heuristic function $\tilde{u}$ can be solved efficiently in quadratic time. The bottleneck here is the calculation for Equations 4.22 and 4.23. However, after the initial calculation, only small adjustments need to be made at each iteration of the hill-climbing procedure, further reducing the run-time of calculating $\tilde{u}$. In this case, it is bounded by the critical path problem used in Equations 4.25 and 4.26, which has a run-time in $O(|T| + |\mathcal{E}|)$ where $|T|$ is the number of tasks in the workflow and $|\mathcal{E}|$ the number of direct, non-transitive edges[12].

To illustrate the behaviour of our *flexible* strategy, we briefly outline the provisioning of an example workflow in the following section.

4.4.4 Illustrative Example

In this section, we discuss how the example workflow introduced in Section 3.5 is provisioned by our algorithm, and how the various performance measures from the previous section are calculated and used in practice. For this example, the appropriate service types are detailed in Table 4.1, along with their failure probabilities, invocation costs, numbers available, their

[11] On average, around six iterations are needed per task in the workflow. During the experimental evaluation of our algorithm (see Section 4.5), a solution was typically found within 250ms (10 tasks) or 5s (50 tasks) on a 3GHz Pentium 4 with 1GB RAM.

[12] We also assume that the probability density functions of service invocation durations and related expected values, as calculated in Equation 4.12, can be efficiently calculated (or else approximated).

respective duration distributions[13] and associated means and variances. These were chosen to represent a set of services with variable performance characteristics — for example, *Translate* is a cheap, fast and unreliable service type, while *Render* is expensive, slow and reliable.

Service	**Fail. Prob.**	**Cost ($)**	**Number**	**Duration**	**Mean (min.)**	**Var.**
BaseCall	0.2	1	50	Gamma(1.5,2)	3	6
GeneAssemble	0.1	5	50	Gamma(5,2)	10	20
Blast	0.3	2	500	Gamma(5,3)	15	45
LookUp	0.5	5	10	Gamma(1.5,1.5)	2.25	3.375
Render	0.1	10	25	Gamma(30,3)	90	270
Translate	0.7	0.5	200	Gamma(1,1)	1	1
Fold	0.2	10	5	Gamma(3,30)	90	2700
Print	0.2	2	20	Gamma(2,3)	6	18

TABLE 4.1: Service types used in the example workflow.

Now, for our illustrative example, we assume that the scientist has a deadline of four hours, and values the workflow at $150, which decreases by $1 for each minute that it is late. Figure 4.3 shows the initial allocation for the workflow. As outlined in Section 4.4.2, the algorithm begins here by randomly provisioning service instances for the constituent tasks of the workflow.

To illustrate the calculations[14] our algorithm performs on this allocation, we consider the upper *Render* task in the workflow (t_4). Here, the algorithm first calculates the probability of success for the task, p_4, using Equation 4.6. As there are a number of service instances ($v_4 = 25$), this probability is $p_4 = 1 - (1 - (1 - 0.1) \cdot 0.62)^{25} = 1.00$. Next, the algorithm calculates the expected cost, $\bar{c}_4$, using Equation 4.10. This is high ($\bar{c}_4 = 1 \cdot 10 \cdot \frac{1-0.4437^{25}}{1-0.4437} = 17.98$), because the initial allocation will ignore any services that finish after the mean duration (even if they are successful). Finally, the expected completion time, $\bar{t}_4$, is calculated using Equation 4.15. Again, this is high ($\bar{t}_4 = \frac{1}{1} \cdot (80.22155 \cdot (1 - \hat{f}_4^{25}) + 94 \cdot (\hat{f}_4 - 25 \cdot \hat{f}_4^{25} + (25-1) \cdot \hat{f}_4^{25+1}) \cdot \frac{1}{1-\hat{f}_4}) = 155.19$, where $\hat{f}_4 = 0.44367$) for the same reason as the expected cost.

Given these values for all tasks in the workflow, the algorithm next derives the overall expected performance measures for the workflow (these are summarised in the box to the right of the workflow). First, the overall success probability, p, is calculated using Equation 4.30. This is low, due to the inappropriate time-out value for the *Fold* task (t_6), which results in a high failure probability of that task ($p = \prod_{\{i|t_i \in T\}} p_i = 1.00^7 \cdot 0.26 = 0.26$). The expected cost, $\tilde{c}$, is estimated next using Equation 4.22. In this case, we derive an estimated cost of $\tilde{c} = \sum_{\{i|t_i \in T\}} r_i \bar{c}_i = 175.25$ for the whole workflow. After this, the algorithm estimates

[13] We assume that services in this example follow a gamma distribution Gamma(k, θ) with pdf $p(x, k, \theta) = x^{k-1} e^{-\frac{x}{\theta}} \Gamma(k)^{-1} \theta^{-k}$, which has been chosen because it is well suited for uncertain service times that are always positive, but are not usually bounded above. The gamma distribution also includes common other distributions such as the exponential and Erlang distributions, both of which are often used in the analysis of service and queueing times (Trivedi (2001)). However, this choice is only for illustrative purposes — in practice, an arbitrary distribution can be used to model service durations.

[14] For readability, all values presented here are reported to two decimal places, except where additional precision is necessary during the calculations.

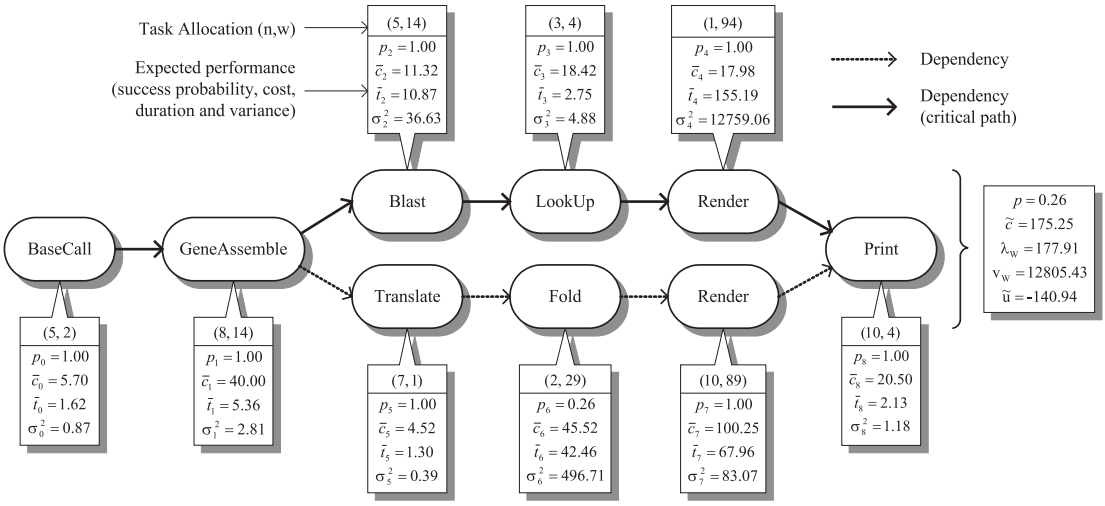


FIGURE 4.3: Initial provisioning allocation.

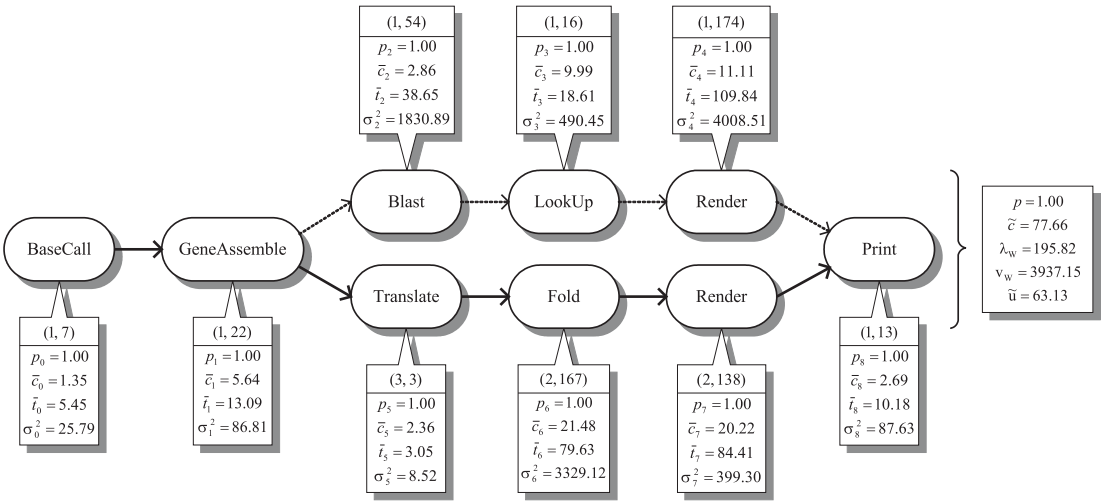


FIGURE 4.4: Finally provisioned workflow.

the distribution of the overall completion time by summing the expected completions times and variances along the critical path, using Equations 4.25 and 4.26. This yields a mean of $\lambda_W = \sum_{\{i|t_i \in \mathcal{P}\}} \bar{t}_i = 1.620{+}5.356{+}10.867{+}2.747{+}155.187{+}2.130 = 177.91$ and a variance of $v_W = \sum_{\{i|t_i \in \mathcal{P}\}} \sigma_i^2 = 0.87{+}2.81{+}36.63{+}4.88{+}12759.06{+}1.18 = 12805.43$. Using these as the mean and variance of a normal distribution ($d_W(x)$ in Equation 4.24, which was derived using the central limit theorem), we estimate that the workflow will finish within the deadline $t_{\max}$ with probability $D_{\max} = \int_{-\infty}^{t_{\max}} d_W(y)\mathrm{d}y = 0.708395$. We also estimate that the probability of finishing between the deadline and t_{last} is $D_{\text{late}} = \int_{t_{\max}}^{t_{\text{last}}} d_W(y)\mathrm{d}y = 0.261157$. In the latter case, we calculate the expected completion time using Equation 4.29 ($\bar{t}_{\text{late}} = 296.766592$). Finally, using these intermediate values in Equation 4.31 yields a total utility estimate of $\tilde{u} = 0.262624 \cdot (0.708395 \cdot 150 + 0.261157 \cdot u(296.766592)) - 175.245220 = -140.94$. This is low because of the high degree of parallelism in the workflow (resulting in unnecessary expenses) and the low overall success probability (resulting in a low estimated reward).

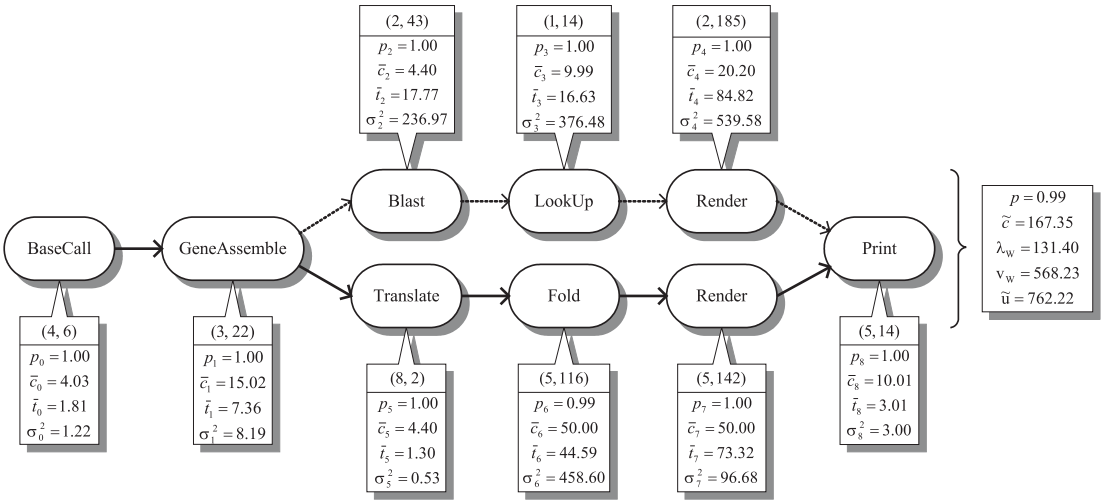


FIGURE 4.5: Provisioned workflow with shorter deadline and higher reward.

To improve this initial allocation, our algorithm now repeatedly considers a number of neighbour allocations and, at each iteration, picks the one that promises the highest estimated profit. This is repeated until no more improvements can be made. Figure 4.4 shows the final allocation found by our algorithm, which includes several tasks where providers have been provisioned in parallel, but mostly relies on serial provisioning as this saves money. Contrasting this with the initial allocation, the improvements are clearly visible — for example, the expected cost of the *Render* task in the upper branch (t_4) has now been reduced from $\bar{c}_4 = 17.98$ to 11.11 and its expected duration has been lowered, simply by choosing a more appropriate waiting time (from $\bar{t}_4 = 155.19$ to 109.84). It is also evident that the structure of the workflow has been taken into account — two providers have been provisioned in parallel for the lower *Render* task (t_7), despite being the same type of service. This means that the task is faster ($\bar{t}_7 = 84.41$), but also more expensive ($\bar{c}_7 = 20.22$) than its counterpart in the upper branch. This is beneficial, because the durations of the lower tasks are generally longer, and so the consumer has to invest more resources in order to meet its workflow deadline. Overall, the consumer now expects to finish within the deadline $t_{\max} = 240$ with probability $D_{\max} = 0.7593$, and between the deadline and $t_{\text{last}} = 390$ with probability $D_{\text{late}} = 0.2397$. In the latter case, its expected finishing time is $\bar{t}_{\text{late}} = 276.4548$, leading to an overall estimated utility of $\tilde{u} = 0.9977 \cdot (0.7593 \cdot 150 + 0.2397 \cdot u(276.4548)) - 77.6572 = 63.13$.

To give a second example, Figure 4.5 shows the same workflow in a scenario where the scientist requires her results in a far shorter time period (within 150 minutes), where she values the outcome more highly (the value is now $\$1000$), and where the penalty is higher than in the previous example ($\$20$ per minute). Here, our algorithm is using a far higher level of redundancy than previously, because that allows the agent to finish more quickly and reliably. For example, for the *Render* task in the lower branch, the algorithm has now provisioned 5 services in parallel, which is very expensive ($\bar{c}_7 = 50.00$), but also results in a low expected duration ($\bar{t}_7 = 73.32$) necessary to meet the overall deadline. Nevertheless, the algorithm still chooses to provision a single service for the *LookUp* task. As before, this is because the tasks on the lower branch

take longer, and so the consumer can save some costs by executing the upper tasks in series. Overall, the consumer now expects to finish within the deadline $t_{\max} = 150$ with probability $D_{\max} = 0.78$ and it is late with probability $D_{\text{late}} = 0.22$ (in which case its expected finishing time is $\bar{t}_{\text{late}} = 163.23$). Due to the high levels of redundancy, the estimated expected cost has now more than doubled compared to the previous case ($\tilde{c} = 167.35$), but the overall higher reward results in a high estimated utility of $\tilde{u} = 762.22$ that justifies the expenses.

In order to evaluate this strategy and to compare it against less flexible approaches, in the following section, we describe a set of experiments that we carried out to this end.

4.5 Empirical Evaluation

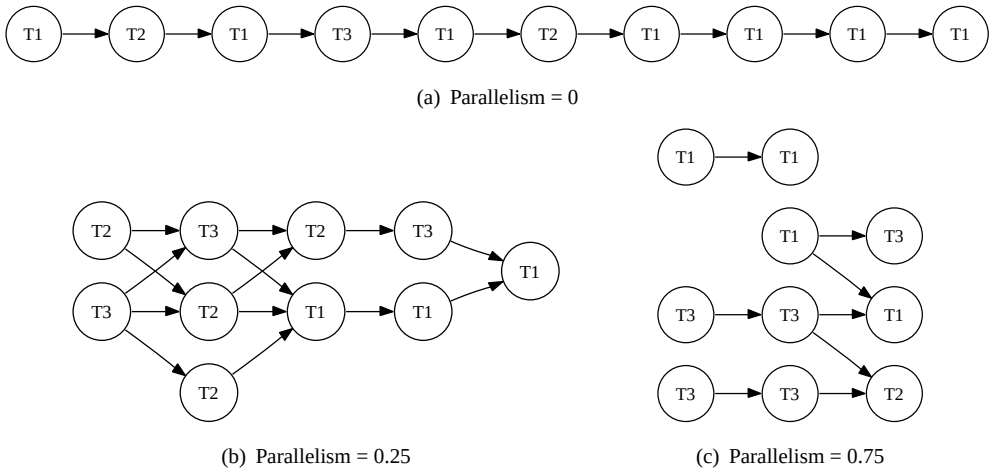
In this section, we experimentally compare our proposed strategies to the currently predominant *naïve* approach[15]. The aim of this part of our work is to compare the performance of our strategies to current approaches when there is some uncertainty in the behaviour of services. We also intend to verify that our flexible strategy in particular takes appropriate decisions and makes an overall profit over a variety of environments while achieving high success rates. We decided to conduct an experimental study (rather than an analytical one), because of the inherent difficulty of calculating workflow completion distributions (see Section 4.4.1).

To this end, we investigate the average profit gained by all strategies, as well as the average proportion of successfully completed workflows. We begin in Section 4.5.1 by describing our experimental testbed and our methodology. In Section 4.5.2, we outline a set of hypotheses to guide our experiments and in Sections 4.5.3–4.5.5 we present our results. Then, in Section 4.5.6, we show how our strategy deals with larger workflows, and in Section 4.5.7, we compare it to the optimal strategy (for a simplified scenario).

4.5.1 Testbed and Methodology

In order to analyse our strategies experimentally, we developed a computer simulation of a service-oriented system. In this simulation, the system is populated by agents offering services, as described in Chapter 3. During each experimental run, a random workflow is first created according to some pre-defined variables. These include the number of tasks in the workflow, the service types that should be included, and a parameter indicating the parallelism of the workflow. The latter is a variable ranging from 0 to 1, where 0 results in completely linear workflows (i.e., the task dependencies form a total order), while 1 causes workflows to be completely parallel (i.e., there are no dependencies between tasks). Any intermediate value indicates the number

[15] As we assume limited information about each task, the *naïve* strategy also subsumes a number of other QoS optimisation approaches that were discussed in Chapter 2. This is because they rely on more detailed information about individual service instances and user-specified constraints that are not available in our model.



(a) Parallelism = 0

(b) Parallelism = 0.25

(c) Parallelism = 0.75

FIGURE 4.6: Several random workflows with 10 tasks, 3 different services types (indicated by the task labels) and varying degrees of parallelism.

of edges that should be introduced as a proportion of the number of edges possible[16] (Figure 4.6 shows some example workflows[17]). This workflow is then executed by a service-consuming agent using one of the strategies outlined earlier in this chapter. These runs are episodic and each involves the execution of exactly one workflow, with no interactions between successive runs.

To analyse the performance of a particular strategy, our simulation executes a large number of experimental runs (the data in this section was collected using 1,000 runs for each experimental setup) and then records the following statistics[18]:

- The proportion of successful workflows for the strategy (where the strategy completes the workflow within time t, so that $u(t) > 0$).
- The average profit of the strategy (the profit of a workflow execution is the difference between the utility reward $u(t)$ for completing the workflow and the incurred cost).

These indicate the extent to which the consumer agent manages to complete its workflows within the given time-constraints and whether it manages to achieve a high average profit at the same time, without making an overall loss.

For the data presented in Sections 4.5.3 – 4.5.5, we used workflows with 10 tasks and a parallelism parameter of 0 (i.e., without parallel tasks). This means that the experiments presented here are particularly relevant to scenarios where workflows are highly interdependent. By using such linear workflows, we were also able to check some of our results analytically to verify that

[16] We implement this by randomly populating an adjacency matrix until the given threshold is reached.

[17] To avoid confusion, it should be noted that Figure 4.6(c) represents a single workflow with 10 tasks, four of which are immediately executable. Parts of the workflow are entirely disconnected in this case, because of the high level of parallelism.

[18] To test for statistical significance, we also record the variances of all averages.

our simulation is correct (in particular, we verified the results presented in Sections 4.5.3 and 4.5.4).

Furthermore, we assumed that there were 1,000 services for every task with each service having a cost of 10 and a gamma distribution with shape $k = 2$ and scale $\theta = 10$ as the probability distribution of the service duration. We set a deadline of 400 time units for each workflow, an associated maximum utility of 1,000 and a penalty of 10 per time unit. We also performed similar experiments in a variety of environments, including heterogeneous and parallel tasks, and observed the same broad trends that are presented in the following section (some of these results are presented in Section 4.5.6).

To prove the statistical significance of our results, we averaged data over 1,000 test runs and performed an analysis of variance (ANOVA) where appropriate to determine whether the strategies we tested produced significantly different results (Cohen (1995)). When this was the case, we carried out pairwise comparisons using the least significant difference (LSD) test. Thus, all results reported in the following sections are statistically significant (at the $p = 0.001$ level).

4.5.2 Hypotheses

Before discussing the results of our experiments, we outline four hypotheses that drive our investigation. The first two are concerned with the effects of the two basic, non-flexible strategies, *parallel(n)* and *serial(w)*. The aim of these hypotheses is to show that it is possible to achieve better results using simple techniques for handling failures than when relying on the *naïve* strategy.

Hypothesis 1. Adopting strategy *parallel(n)* in uncertain environments can lead to an improvement in the average profit over the *naïve* strategy.

Hypothesis 2. Adopting strategy *serial(w)* in uncertain environments can lead to an improvement in the average profit over the *naïve* strategy.

The other two hypotheses are concerned with evaluating the *flexible* strategy. Here, we present two hypotheses concerned with the average profit and the success probability. This presents the flexible strategy in more detail than the previous two strategies due to its importance to our research.

Hypothesis 3. The *flexible* strategy produces a higher profit than any of the other examined strategies, averaged over all cases.

Hypothesis 4. The *flexible* strategy successfully completes a higher proportion of workflows than any of the other examined strategies, averaged over all cases.

To evaluate Hypotheses 1 – 4, we tested each of the four strategies *naïve*, *parallel(n)*, *serial(w)* and *flexible* using the same experimental variables (as outlined in Section 4.5.1). We summarise the results by discussing each hypothesis separately.

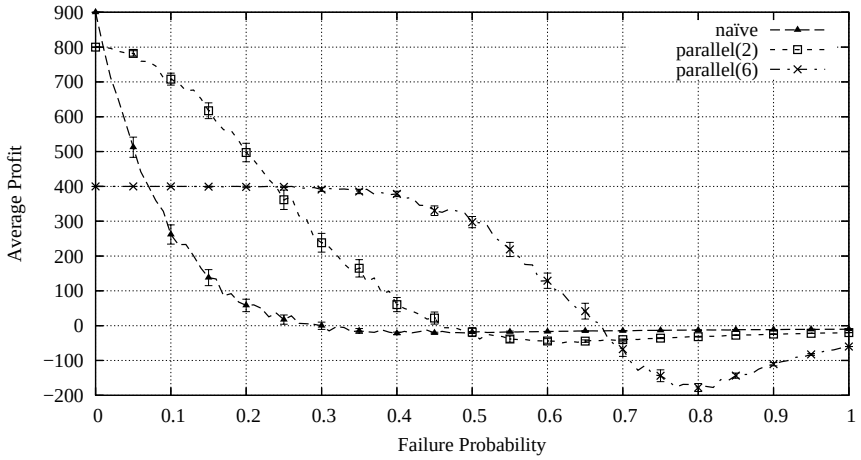


FIGURE 4.7: Effect of provisioning different numbers of services in parallel (data shown with 95% confidence intervals).

4.5.3 Parallel Provisioning (Hypothesis 1)

In our first experiment, we compared the performance of strategy *parallel(n)*[19] with the *naïve* approach in environments where services have a varying probability of failure, as shown in Figure 4.7 (throughout this section, we vary the failure probability in steps of 0.01 from 0 to 1). From this, it is clear that there is a considerable difference in performance between the different strategies — the average profit gained by the *naïve* strategy falls dramatically as soon as failures are introduced into the system. In this case, the average profit gained by provisioning single services falls to around 0 when the failure probability of services is only 0.3. A statistical analysis reveals that the *naïve* strategy dominates the other two when there is no uncertainty in the system. However, as soon as the failure probability is raised to 0.02, *parallel(2)* begins to dominate the other strategies. Between 0.35 and 0.65, *parallel(6)* then becomes the dominant strategy as increased service redundancy leads to a higher probability of success. Above this, the parallel strategies do not yield better results than the *naïve* strategy as they also begin to fail in most cases.

Summarising these trends, it is obvious that parallel provisioning yields a considerable improvement over the *naïve* approach in a range of environments. For example, when the failure probability is 0.2, provisioning two services results in an average profit of 497.2 ± 26.6 (with 95% confidence interval), while the *naïve* strategy achieves only 58.2 ± 17.9. This leads us to conclude that the *parallel(n)* strategy can indeed lead to an improvement and, hence, that Hypothesis 1 holds. However, no parallel strategy dominates the other and they all eventually make losses when the probability of failure increases to such an extent that the chosen redundancy levels do not suffice to ensure success. In this context, it is interesting to note the losses of each strategy become smaller again after a certain minimum is passed (e.g., *parallel(6)* reaches a minimum

[19]Here, we arbitrarily chose $n = 2$ and $n = 6$ as representative of the general trends displayed by the strategy as more services are provisioned.

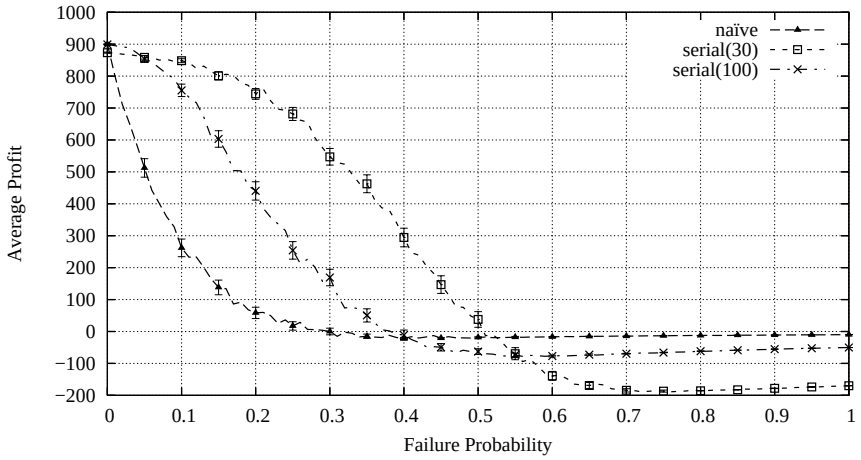


FIGURE 4.8: Effect of different amounts of waiting times for serial provisioning (data shown with 95% confidence intervals).

when the failure probability is around 0.8). This is because the strategies fail earlier in the workflow and therefore lose a lower investment. In conclusion, parallel provisioning is sensitive to the right choice of n and might even lead to an overall loss if the wrong parameter is chosen.

4.5.4 Serial Provisioning (Hypothesis 2)

We carried out a similar experiment to verify the advantage of serial provisioning over the *naïve* strategy (see Figure 4.8). Here, again, there is a marked improvement over the *naïve* strategy for failure probabilities up to and including 0.5. This improvement is due to the fact that serial provisioning responds to failures as they occur, while only paying for additional services when necessary. However, as the failure probability rises, this strategy begins to miss its deadlines and hence incurs increasingly large losses.

Overall, a significant improvement in the average profit for some environments leads us to conclude that Hypothesis 2 holds. Again, the strategy is sensitive to the choice of parameter w, but this time, *serial(30)* dominates *serial(100)* when there is uncertainty, until both make a loss.

4.5.5 Flexible Provisioning (Hypotheses 3 and 4)

To show how the *flexible* strategy compares against the *naïve* provisioning approach and our non-flexible strategies, Figure 4.9 plots the average profit of various strategies against the service failure probabilities. Here, it is clear that the flexible approach performs better than any of the other strategies. This is due, in part, to the flexibility of the strategy that allows it to provision more services for later parts of the workflow, where success becomes more critical as a higher investment has already been made. The flexible approach also combines the benefits of the other strategies, allowing the agent to choose between parallel (e.g., when there is little time) and serial provisioning (e.g., when the agent can afford the extra waiting time) or a mixture

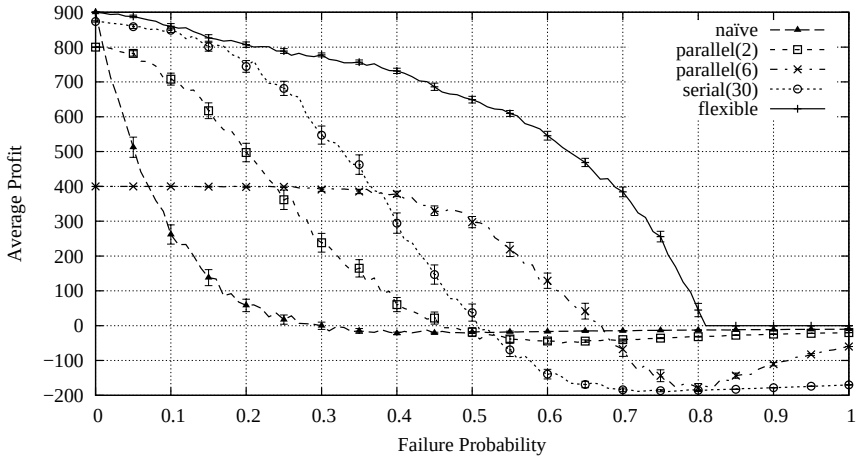


FIGURE 4.9: Average profit of flexible strategy (data shown with 95% confidence intervals).

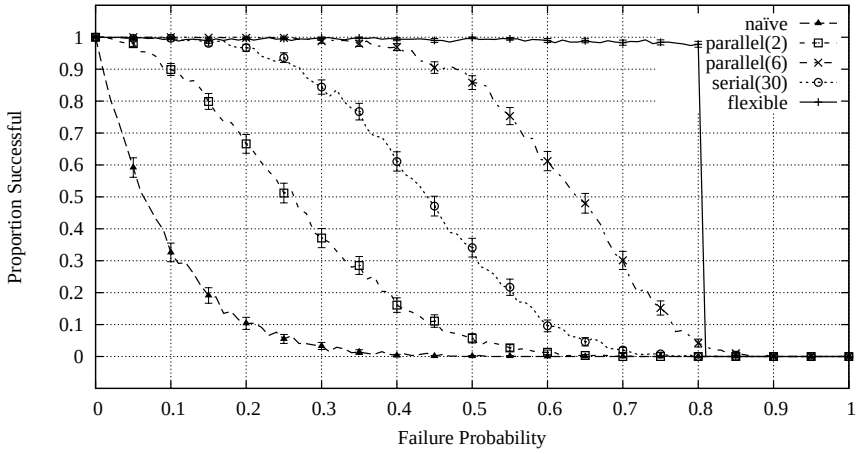


FIGURE 4.10: Success probability of flexible strategy (data shown with 95% confidence intervals).

of the two. Although performance degrades as services become more failure-prone, flexible provisioning retains a relatively high average profit when all other strategies start to make a loss. Furthermore, the strategy avoids making an overall loss due to its prediction mechanism, which ignores a workflow when it seems infeasible.

In Figure 4.10, we plot the success probability of each strategy against the service failure probabilities. While maximising the workflow success probability was not the primary aim of devising the *flexible* strategy, the results show that the strategy performs very well over a range of environments. More specifically, it initially completes almost all workflows successfully, and maintains this trend up to a failure probability of 0.8, by which all other approaches have large failure rates. When this failure probability is exceeded, the strategy suddenly begins to ignore all workflows, because it cannot find a feasible allocation to offer a positive return. While the *parallel(6)* strategy still succeeds in a small fraction of workflows, it is incurring significant losses, as explained in the previous sections.

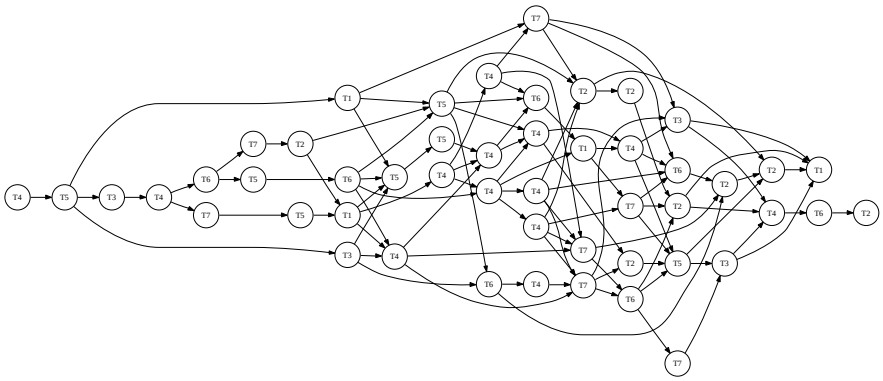


FIGURE 4.11: An example workflow consisting of 50 tasks.

From these results, it is clear that hypotheses 3 and 4 hold. While there are some cases where other strategies achieve similar results (e.g., when services never fail), the *flexible* strategy achieves consistently good results, and, averaged over all results discussed in Sections 4.5.3–4.5.5, dominates all other strategies. This is summarised in Table 4.2, which contains the performance statistics of our representative strategies, averaged over all environments we tested (using the same data as in Figures 4.7–4.10). These results highlight the benefits of our strategies, and show that the *flexible* strategy by far outperforms the *naïve* approach. In particular, we achieve an improvement of approximately 700% in average profit and successfully complete around 80% of all workflows. To show that these results also hold in other scenarios, in the next section, we consider a more complex case than the workflows discussed so far.

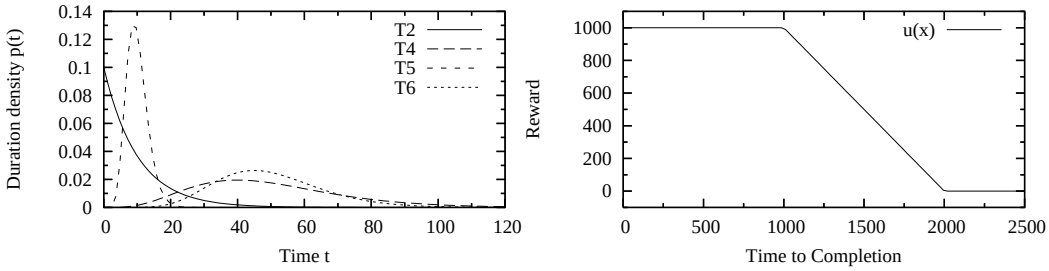
Strategy	Average Profit u_c	Profit vs naïve	Success rate p_s
naïve	65.16 ± 1.68	1	0.095 ± 0.002
serial(100)	142.47 ± 2.46	2.19 ± 0.07	0.258 ± 0.003
parallel(2)	177.98 ± 2.37	2.73 ± 0.08	0.272 ± 0.003
parallel(6)	180.06 ± 1.86	2.76 ± 0.08	0.626 ± 0.003
serial(30)	217.12 ± 3.06	3.33 ± 0.10	0.439 ± 0.003
flexible	523.90 ± 2.20	8.04 ± 0.21	0.795 ± 0.003

TABLE 4.2: Summary of results with 95% confidence intervals

4.5.6 Performance in Complex Environments (Hypotheses 3 and 4)

In the previous section, we examined the performance of our strategies in the context of a small, sequential workflow with only one type of service. As mentioned above, this allowed us to verify some results analytically. In this section, we briefly present the results of a more complex problem, and, in doing so, demonstrate that the same overall trends can be observed.

For this experiment, we created random workflows that consist of 50 tasks and have a parallelism parameter of 0.25. We also chose a random service type for each task from a set of seven types that are detailed in Table 4.3. These service types were chosen to display a variety of parameters. For example, T_1 is extremely fast and will almost certainly complete by the next



(a) Example service durations (b) Utility function

FIGURE 4.12: Experimental settings: (a) shows some service duration functions and (b) gives the utility function we use.

time step following its invocation, while, at the other end of the scale, T_4 and T_6 both have a mean duration of 50 time units (Figure 4.12(a) shows the duration functions for some of the services). Services of type T_1 are also very cheap (0.1 units), while those of T_7 cost 20 times as much.

Service	**Cost ($)**	**Duration**	**Mean (min.)**	**Var.**
T_1	0.1	Gamma(1,0.1)	0.1	0.01
T_2	0.1	Gamma(1,10)	10	100
T_3	1	Gamma(5,1)	5	5
T_4	1	Gamma(5,10)	50	500
T_5	2	Gamma(10,1)	10	10
T_6	2	Gamma(10,5)	50	250
T_7	2	Gamma(100,0.1)	10	1

TABLE 4.3: Service types used to test complex workflows.

Furthermore, we assumed that there were 100 instances of each service type, and we used a utility function with a deadline of 1,000 time units, a penalty of 1 per time unit and a maximum utility of 1,000 (this is shown in Figure 4.12(b)). Again, we tested our strategies in environments where services have different failure probabilities (0,0.01,0.02,...,1), but this time we included some variance in the failure probabilities of different service types. Specifically, during each experimental run for a particular average failure probability f, we assigned a failure probability to each service type that was drawn from a beta distribution[20] with parameters $\alpha = f \cdot 10$ and $\beta = 10 - \alpha$ (unless $f = 0$ or $f = 1$, in which case all services had the same failure probability). This process, which was repeated for all 1,000 runs for each value of f, meant that the average failure probability of all service types would approach f, but still allowed considerable variance between the different types of services.

With these experimental settings[21], we again tested the *flexible* strategy against several other approaches (see Figure 4.13). Here, a similar pattern as shown in Figure 4.9 emerges and our *flexible* approach clearly dominates the other approaches when service success is uncertain (i.e.,

[20]The beta distribution was simply chosen because it always ranges between 0 and 1.

[21]These parameters were chosen to exemplify the performance of the strategy. We have experimented with other values and observed the same broad trends.

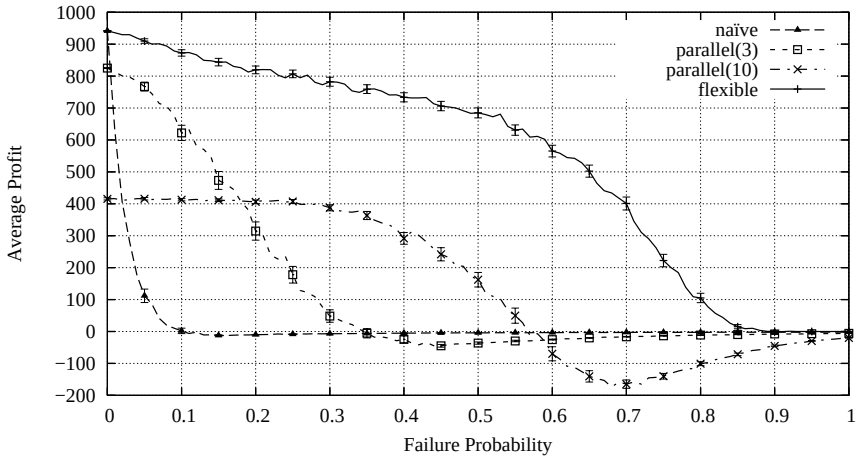


FIGURE 4.13: Average profit for various strategies when faced with complex workflows (data shown with 95% confidence intervals).

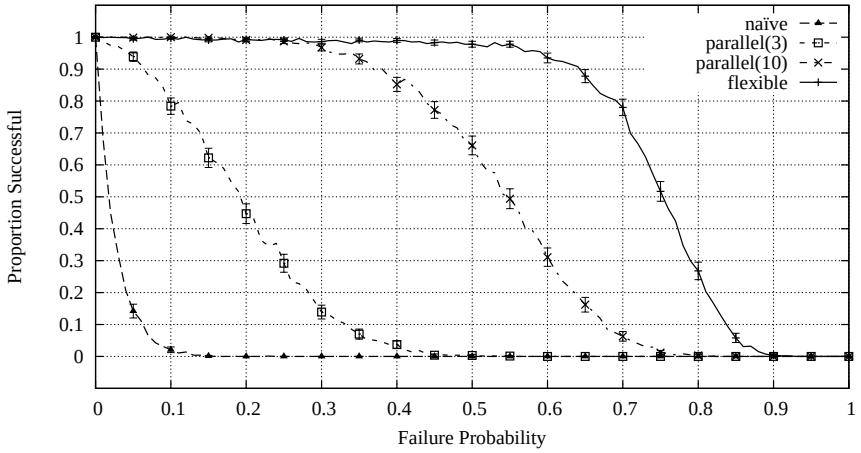
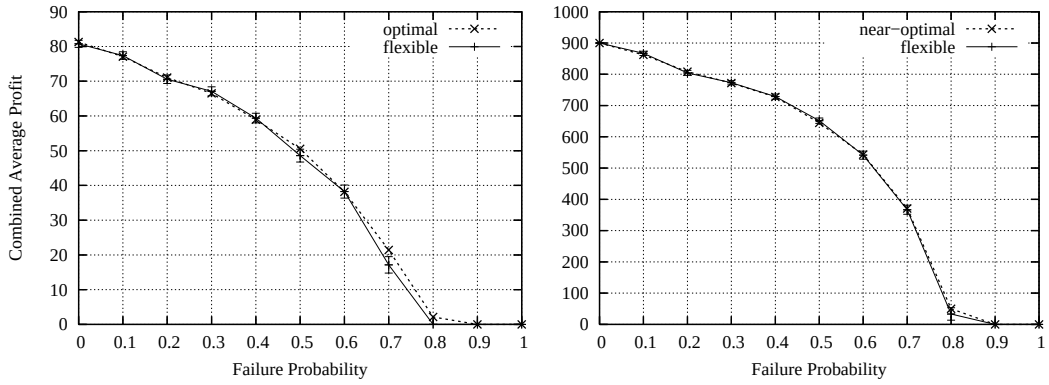


FIGURE 4.14: Success probabilities for various strategies when faced with complex workflows (data shown with 95% confidence intervals).

when the failure probability is greater than 0). When no services fail (failure probability is 0), the flexible strategy does as well as the *naïve* approach and better than any of the others.

To complete the summary of this experiment, Figure 4.14 shows the success probabilities of the strategies we tested. Again, the flexible strategy performs very well compared to the other approaches. It initially completes at least as many workflows as the other strategies, then stays at a high level and only starts to drop below 90% when the failure probability rises to 70%. Overall, the results presented in this section further highlight the promise of flexible provisioning techniques and show that our strategy is applicable to large workflows with heterogeneous service types and parallel workflow tasks. In particular, the results confirm that our hypotheses 3 and 4 hold in these environments, as the same trends as in the previous section are observed.



(a) Workflow with 3 Tasks (b) Workflow with 10 Tasks

FIGURE 4.15: Average profit of *flexible* strategy (with 95% confidence intervals), compared to the optimal strategy for 3 tasks in (a) and to the near-optimal strategy for 10 tasks in (b).

4.5.7 Optimality of Flexible Provisioning

As discussed previously, the *flexible* strategy uses a heuristic utility function and a hill-climbing mechanism that is not optimal in general. However, adopting this heuristic method has made the provisioning of complex workflows tractable. In this section, we compare the performance of our algorithm to the theoretical optimal. More specifically, we first show our results in a simple environment (we consider a workflow with 3 sequential tasks, each of which has a cost of 3, duration distribution Gamma(2,4), 20 providers, and a utility function with deadline 30, maximum utility 100 and penalty 10). This scenario allows us to solve our original optimisation problem (as given by Equation 4.4) analytically. This is then followed by an analysis of the environment used in Sections 4.5.3 – 4.5.5. Because deriving the optimal solution is intractable in this case, we designed a new *analytical flexible* strategy. This is based on our *flexible* strategy, but accurately calculates the expected utility, rather than relying on a heuristic function. It then repeatedly performs a hill-climbing search with random restarts (we restart the algorithm 200 times with random initial allocations). We believe that this is a reasonable approximation to the optimal, and, in fact, there is no significant difference between its performance and the theoretical optimal in the smaller environment.

Figure 4.15 shows the average profit of our strategy in these two environments (here, failure probabilities were varied in steps of 0.1 due to the computational cost of calculating an optimal solution). In both cases, while clearly sub-optimal, our strategy comes close to the expected utility of the optimal or near-optimal strategies. In fact, when averaging over the failure probabilities we examined, for 3-task workflows (Figure 4.15(a)), our *flexible* strategy achieves an average utility of 41.7 ± 0.7, compared to the optimal expected utility of 42.5, which corresponds to achieving $98.2 \pm 1.7\%$ of the optimal. For 10-task workflows (Figure 4.15(b)), we achieve even closer results with an average utility of 512.0 ± 7.0 compared to the near-optimal expected utility of 516.1. In fact, a t-test confirms that this is not a statistically significant difference ($p = 0.764$). This improvement, compared to the smaller workflows, may be due to our reliance on the central limit theorem to estimate the duration distribution. When the workflows become

larger, this tends to give more accurate estimates. Overall, these results are promising, because they show that our strategy achieves a level of performance that is close to the optimal in the environments we tested, using a fast heuristic method that is tractable even for large workflows.

4.6 Summary

In this chapter, we outlined five strategies for provisioning services as part of complex workflows in environments where little information is known about the available services. The first four of these strategies are based on related work in the area. Specifically, the *naïve* approach uses no knowledge about the performance characteristics of services and simply provisions a single random service for each task in the workflow. The following strategies, *parallel(n)*, *serial(w)* and the composite *hybrid(n,w)*, deal with potential services failures by proactively invoking multiple services for each task and by responding to failures by re-provisioning new services. However, these strategies rely on a human decision-maker to choose the parameters n and w (respectively the number of services to invoke in parallel and the waiting time before re-provisioning). This shortcoming is finally addressed by the novel *flexible* strategy, which provisions services flexibly and without human intervention based on the performance characteristics of services and the constraints imposed by the workflow and its reward function.

After introducing these strategies, we described a number of empirical experiments. These highlighted the benefits of the *flexible* approach, which consistently outperformed all other strategies and managed to maintain a high success probability of around 95% even when individual services had an 80% failure probability. This strategy meets our original requirements by dealing with service failures both reactively (Requirement A.2.a) and proactively (A.2.b). It also deals with uncertainty in a principled manner (Requirement A.1) by taking decisions that maximise the consumer's predicted utility, and we have shown that it deals well with larger workflows and environments with many services (Requirement A.3).

However, we have so far only looked at systems where the information about services is highly limited. We argued that there are many realistic application areas for this case, but there are clearly other environments where the consumer may have more detailed information about individual services and where these might be highly heterogeneous. We address this case (and thereby our Requirement M.4) in the following chapter.

Chapter 5

Service Provisioning with Heterogeneous Providers

We now turn our attention to environments where more finely-grained information about service instances is available to the consumer. Specifically, we assume that providers offer their services at varying levels of quality and that the consumer has some information to distinguish these services. For example, the consumer may know that certain data-processing services offered by large companies are more reliable than their cheaper counterparts running on idle desktop computers. Similarly, a given provider may offer tier-based services, whereby the consumer may pay a higher service fee, in order to receive a better quality of service. Such tier-based services might be implemented, for example, by elevating the priority of a consumer in the provider's scheduling algorithm, hence resulting in a shorter and more certain service duration. As outlined in Chapter 3, we again assume that the consumer has obtained such performance information about services either through previous interactions or a suitable trust mechanism.

When such heterogeneity exists in the system, the provisioning problem becomes more difficult, as its dimensionality increases: rather than just considering the number of parallel services and their waiting times, the consumer now needs to consider which service instances should be provisioned. To address this problem (and our original Requirement M.4), we extend our work from the previous chapter. Specifically, in Section 5.1, we describe a system model that includes heterogeneous services. As there is already existing work that proposes a provisioning approach for a similar case, we briefly formalise that approach for our framework in Section 5.2. Next, we return to our flexible provisioning strategy and show how it can be adapted for systems with heterogeneous providers in Section 5.3. Finally, in Section 5.4, we show empirically that our extended approach performs well in practice by benchmarking it against the algorithms in Section 5.2 and others from the state of the art.

5.1 Model Extension

In contrast to the previous chapter, we now assume that the consumer has a cost, failure probability and duration distribution associated with each service instance s_i, rather than for the whole set S_i ($c(s_i)$, $f(s_i)$ and $d(s_i, x)$, as introduced in Section 3.3). Additionally, we introduce the notion of service *populations*, which describe services that display identical behaviour and performance characteristics. Formally, we let $P = \{P_1, P_2, P_3, \ldots, P_{|P|}\}$ be a set partition of $\mathcal{S}$, whose members are disjoint subsets of $\mathcal{S}$ with $\bigcup_i P_i = \mathcal{S}$. 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 service type that is mapped to s_x by μ is also mapped to s_y.

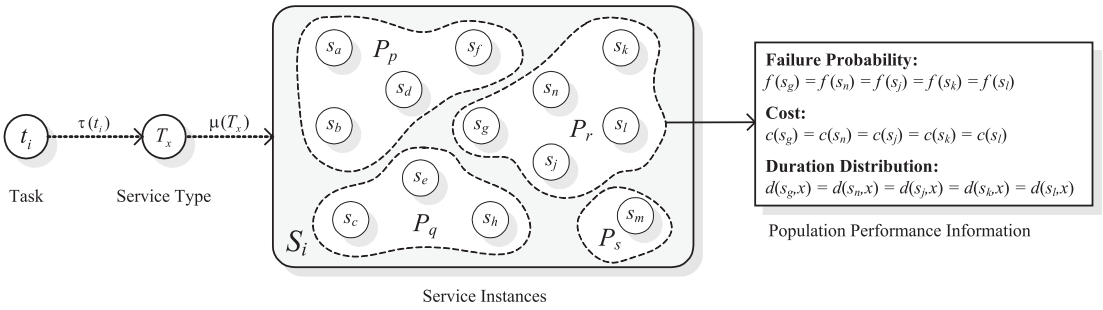
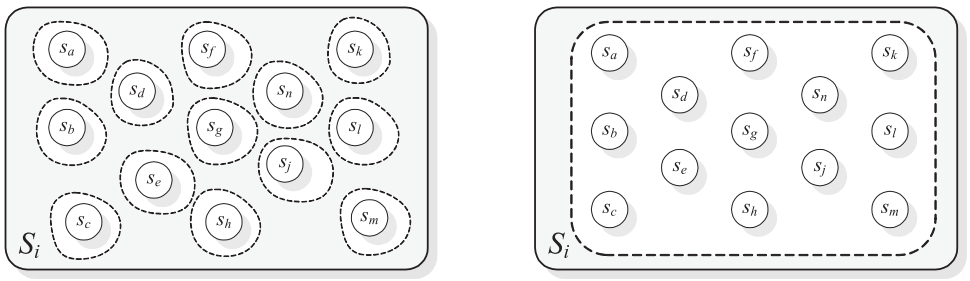


FIGURE 5.1: Information that is available about the service populations for each task.

In more detail, Figure 5.1 illustrates this extended system model. Here, thirteen services are able to solve task t_i, as given by the set S_i. Furthermore, this set is again partitioned into four smaller sets, P_p, P_q, P_r and P_s, each of which represents a group of service instances whose behaviour is identical. For example, s_k and s_l here have the same failure probability, duration distribution and cost, but s_k and s_m will differ in some or all of their performance characteristics.

The model presented here includes both the cases where the consumer has detailed information about each service instance (see Figure 5.2(a)), and where it has only limited information about the whole set S_i, as assumed in the previous chapter (see Figure 5.2(b)). The notion of service populations 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



(a) Information about each service in S_i.

(b) Information limited to overall set S_i.

FIGURE 5.2: Examples where different levels of information about the services in S_i are available (dashed lines denote the partitions of S_i).

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 $P_p = \{s_a, s_b, s_d, s_f\}$ as cheap, unreliable providers, and $P_q = \{s_c, s_e, s_h\}$ as reliable, but expensive providers[1]).

In contrast to the previous chapter, we also no longer assume that previous services are ignored as soon as a task is re-provisioned. Hence, it is now possible that the consumer invokes service s_1 for a given task, waits for some time, then invokes s_2 for the same task, but later receives a response from s_1, thereby completing the task. We make this change to our model here, because we believe that it is a more realistic assumption when services are highly heterogeneous and especially when different service populations follow varying duration distributions. To illustrate this with an example, the consumer might be able to choose from two populations for a particular task: P_1, which contains reliable, slow and expensive services, and P_2, which contains unreliable, fast and cheap services. In this case, its best strategy may be to invoke one service from each population, followed by more services from P_2, invoked in series and with short time intervals to account for the population's short service duration. Such behaviour would be ineffective if we assumed that a service from P_1 is ignored as soon as a second service from P_2 is invoked.

Now, as we discussed in Chapter 2, there is already a significant body of research that has considered the provisioning of services in environments where they are highly heterogeneous. Typically, such research has concentrated on optimising weighted sums of various quality-of-service parameters, but without planning for service uncertainty in a principled manner. In the following section, we formalise this approach in the context of our system model. This gives us an additional benchmark against which to compare our extended flexible approach (discussed in Section 5.3).

5.2 QoS-based Provisioning

This section is based closely on the provisioning strategies described by Zeng et al. (2004), but many similar approaches are widely used in the literature (Aggarwal et al. (2004); Jaeger and Mühl (2007)). We already briefly introduced these strategies in Section 2.4.3.2, and here we simply elaborate them against the background of our system model. We start first by describing a local strategy that makes myopic decisions about the provisioning of each task during execution (Section 5.2.1), and then we discuss a global provisioning strategy that provisions entire workflows before execution (Section 5.2.2).

[1] As in Chapter 4, we assume that the probabilistic performance characteristics in this case express the uncertainty and variability of providers within a population. As before, the cost of providers within a population is assumed homogeneous, but our work similarly applies when there is some uncertainty about the cost and we only know its expected value.

5.2.1 Local Weighted Optimisation

This strategy provisions each task only when it becomes available and without considering the impact of provisioning on the overall workflow (e.g., whether the workflow will likely succeed within the given deadline or whether the overall workflow cost will exceed the maximum utility). More specifically, when task t_i becomes available, it provisions and immediately invokes a matching service s^* that maximises a weighted sum:

$$s^* = \underset{s \in \mu(\tau(t_i))}{\operatorname{argmax}} \sum_{i=1}^{3} \breve{w}_i \cdot Q_i(s) \tag{5.1}$$

$$Q_i(s) = \begin{cases} 0 & \text{if } q_{\max,i} = q_{\min,i} \\ \frac{q_{\max,i} - q_i(s)}{q_{\max,i} - q_{\min,i}} & \text{otherwise} \end{cases} \tag{5.2}$$

where $q_1(s) = c(s)$ is the service cost, $q_2(s) = f(s)$ is its failure probability and $q_3(s) = \frac{1}{D(s,t_{\text{zero}})} \sum_{t=1}^{t_{\text{zero}}} t \cdot (D(s,t) - D(s,t-1))$ is its mean duration (provided it succeeds within t_{zero} time steps). The values for $q_{\max,i}$ and $q_{\min,i}$ are the largest and smallest of these parameters among the services that are considered, and each weight $\breve{w}_i \in [0,1]$ attaches a relative importance to the associated parameter (with $\sum_i \breve{w}_i = 1$).

For the purpose of our experiments, we set $\breve{w}_1 = \breve{w}_2 = \breve{w}_3 = \frac{1}{3}$, which strikes a balance between the various qualities (in most environments, we did not observe a significant difference in performance when adopting other weight distributions). With this, we define the *local* strategy as follows:

Definition 7 (Local Strategy)**.** An agent following a ***local*** strategy provisions a service s^* for each task, so that the weighted sum given in Equation 5.1 is maximised (with $\breve{w}_1 = \breve{w}_2 = \breve{w}_3 = \frac{1}{3}$).

Typically, such a local strategy re-provisions services immediately upon failure. However, as we assume silent failures, it is again necessary to introduce explicit time-out values in order to produce an adaptive strategy. Furthermore, it is possible to increase the robustness of the strategy by including redundancy:

Definition 8 (Local(n,w) Strategy)**.** An agent following a ***local(n,w)*** strategy orders all matching services in descending order of the sum given in Equation 5.1 and then provisions the first n services. If the task has not been successful after w time steps, it repeats this process with the remaining services until the task has been completed.

As such, the strategy makes provisioning decisions about services based on their performance characteristics and reacts to failures when they occur, but considers only single tasks in isolation. The strategy we discuss in the following section addresses this limitation by aggregating performance characteristics over the entire workflow.

5.2.2 Global Weighted Optimisation

This is perhaps the most widely adopted approach for provisioning services in the literature (Aggarwal et al. (2004); Jaeger and Mühl (2007); Zeng et al. (2004)), as we already discussed in Section 2.4.3.2. An agent using this approach considers the whole workflow, provisioning a service for each task, so that a weighted sum similar to Equation 5.1 is maximised. This sum now aggregates the quality parameters over the entire workflow and may contain constraints, such as an overall budget or time limit. More specifically, let $\rho \in \mathcal{S}^{|T|}$ be a vector of $|T|$ services, such that $\rho_i \in \mu(\tau(t_i))$ is the service provisioned for task t_i. The global provisioning approach then finds a vector ρ^* that maximises the weighted sum:

$$\rho^* = \underset{\rho \in \mathcal{S}^{|T|}}{\operatorname{argmax}} \sum_{i=1}^{3} \breve{w}_i \cdot \hat{Q}_i(\rho) \tag{5.3}$$

where $\hat{Q}_i(\rho)$ is again a normalised quality metric, derived from one of the three following workflow qualities:

- $\hat{q}_1(\rho) = \sum_i c(\rho_i)$ is the *workflow cost*,
- $\hat{q}_2(\rho) = \sum_i \ln(1 - f(\rho_i))$ is the natural logarithm of the *workflow success probability*[2],
- $\hat{q}_3(\rho)$ is the *workflow duration*[3].

Furthermore, this optimisation problem may be subject to constraints on these quality parameters. In practice, we derive these directly from our workflow model as follows:

$$\hat{q}_1(\rho) \leq u_{\text{max}} \tag{5.4}$$
$$\hat{q}_3(\rho) < t_{\text{zero}} \tag{5.5}$$

Respectively, these denote that the agent should not spend more than the inherent value of the workflow, and that it should aim to complete the workflow before it receives no more utility from completion. Thus, we define the global strategy as follows:

Definition 9 (Global Strategy)**.** An agent following a ***global*** strategy selects a vector of services ρ^* that maximises Equation 5.3 (with $\breve{w}_1 = \breve{w}_2 = \breve{w}_3 = \frac{1}{3}$), subject to the constraints given by Equations 5.4 and 5.5.

Although some existing global strategies adapt to failures, they typically assume explicit failure messages. Thus, we introduce an explicit time-out parameter, as before:

[2]The logarithm is used here, so that the success probability can be expressed as a sum and thus solved by existing linear integer programming techniques.

[3]This is calculated by aggregating the mean service durations (given by $q_3(s)$) along the critical path, as described in Section 4.4.3.2.

Definition 10 (Adaptive Global(w) Strategy)**.** An agent following an ***adaptive global(w)*** strategy provisions services as the global strategy. However, when a provisioned service has not been successful after w time steps, the agent re-provisions the respective task and all other tasks that have not been invoked yet. In doing so, it adjusts the constraints in Equations 5.4 and 5.5 to take into account the time that has already passed and the total expenditure incurred. Regardless of w, the strategy also re-provisions all uncompleted tasks when $\hat{q}_3(\rho) \geq t_{\text{zero}}$.

As discussed in Section 2.4.3.2, these strategies have a number of shortcomings, and so we describe a more flexible approach based on our previous work in the following section. In Section 4.5, we will then return to the QoS-based approaches and use them as benchmarks against which to compare our proposed strategies.

5.3 Flexible Service Provisioning

In this section, we extend our flexible provisioning strategy to deal with heterogeneous services. Our new strategy builds closely on the techniques introduced in the previous chapter, but we adapt the local task predictions to account for the extended model outlined above. We also consider a more finely-grained decision problem than before by allowing the number of parallel services and time-out values to vary during the execution of a single task, and we propose a modified local search algorithm that takes into account the larger search space.

Our discussion of the flexible strategy for heterogeneous services is divided into five main sections. We begin by formalising the provisioning problem with our modified model (Section 5.3.1), followed by our extended local search and consumer algorithms (Section 5.3.2). In Section 5.3.3, we describe how to adapt the local task predictions for the more complex model of this chapter. As the strategy discussed here considers a large solution space, we outline in Section 5.3.4 how it may be simplified for cases where time is critical. Finally, we illustrate our modified strategy using the bioinformatics workflow from Section 3.5.

5.3.1 Problem Formulation

In the previous chapter, our flexible provisioning strategy optimised two parameters for each task: the number of parallel services (n_i) and a waiting time before provisioning more services (w_i). As we now assume more information to be available about individual service instances, we clearly want to take this into consideration and extend the problem accordingly. A simple approach might be to introduce a third decision variable to indicate from which population services should be selected. Such an approach would require minimal modifications of the work discussed in the previous chapter, but it is likely to be insufficient in most cases, as the consumer will often benefit from provisioning services from different populations for a single task. This is particularly evident when populations are small, as in Figure 5.2(a), or when the consumer can

benefit from provisioning services from several populations concurrently (for example, when relying on services with very different characteristics, as described in Section 5.1, or when there is little difference between the populations).

For these reasons, we decided to extend the consumer's decision space in this chapter and consider a *detailed provisioning allocation* that constitutes a plan of which services to invoke for a given task and at what time (as long as the task is still uncompleted):

Definition 11 (Detailed Provisioning Allocation)**.** A detailed provisioning allocation is a mapping $\alpha : T \rightarrow (\mathcal{S} \rightarrow \mathbb{N})$ that associates each task in T with the provisioned services for that task and their respective invocation time steps.

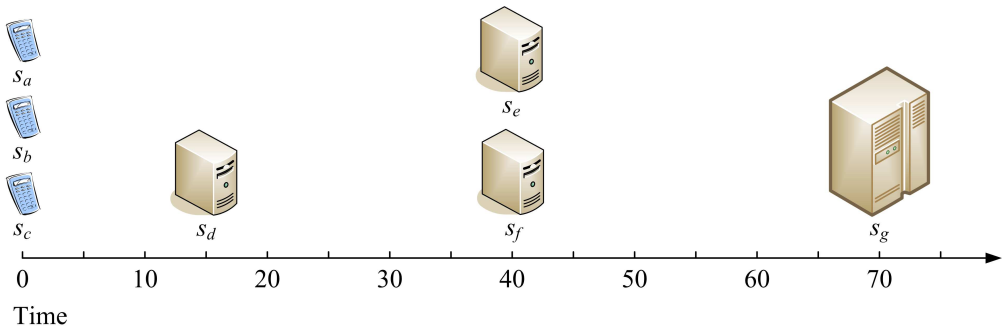


FIGURE 5.3: A detailed provisioning allocation with three service populations.

This allows the consumer to provision services of different populations with varying invocation times. As an example of this, Figure 5.3 shows an allocation for a particular task t_i, $\alpha(t_i) = \{(s_a, 0), (s_b, 0), (s_c, 0), (s_d, 15), (s_e, 40), (s_f, 40), (s_g, 70)\}$. Here, the consumer first provisions a set of cheap and unreliable services to be invoked at time step 0 (s_a, s_b and s_c). If these are not successful by time step 15, the consumer will then proceed to invoke a service from a more reliable and more expensive population (s_d), followed 25 time steps later by two more services of the same population (s_e and s_f). After a longer time-out period, the consumer then invokes the most reliable and expensive service available (s_g). Using this allocation, the consumer initially exploits the cheaper services, as there is a possibility that they complete the task successfully. When this is not the case, the consumer then switches to the more reliable services to ensure that the task is eventually completed successfully.

Having defined this allocation, we now extend our definition of the *flexible* strategy (Definition 6) to cover heterogeneous services and term the new strategy *detailed flexible*, as it considers a more finely-grained decision problem:

Definition 12 (Detailed Flexible Strategy)**.** A consumer following a ***detailed flexible*** strategy makes appropriate decisions to provision services for its workflow. To this end, the agent finds a suitable detailed provisioning allocation, so that the agent's predicted profit is maximised.

Again, we can formulate this as an optimisation problem:

$$\max_{\alpha} \ (\bar{u}_t(\alpha) - \bar{c}(\alpha)) \tag{5.6}$$

where $\bar{u}_t(\alpha)$ is the expected reward of following allocation α and $\bar{c}(\alpha)$ is the associated expected cost.

This problem is computationally hard for the same reasons as described in the previous chapter, and so we decided to take a similar approach in solving it. However, the problem here is more complex, as we now consider a larger decision space than before (this complexity is further investigated in Appendix C). This means that we require a modified local search technique, which we outline below (Section 5.3.2). This is followed in Section 5.3.3 by a discussion of an updated utility estimation approach that takes into account the model extensions described in Section 5.1.

5.3.2 Updated Generic Algorithm

Our modified algorithm, shown in Algorithm 5.7, follows broadly the same structure as that in Section 4.4.2. In more detail, it starts in line 2 by generating a random allocation α, which is then iteratively improved, based on an estimated utility value (lines 5–20). During each iteration, the algorithm picks a random task t_i from the workflow (line 9), and considers each of a set of neighbours of α, which are obtained by randomly applying small changes to the provisioned services for task t_i (line 11). 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 15). 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`[4] times without discovering a better solution, at which point the current α is returned (line 21).

Now, the local search procedure in Algorithm 5.8 depends on two procedures: GENERATE-INITIAL and GENERATE-NEIGHBOURS. Respectively, these create an initial solution and generate neighbour allocations of a given α, as described in the following.

5.3.2.1 Initial Provisioning Allocation Creation

The GENERATE-INITIAL procedure, detailed in lines 23–36, initially provisions a random non-empty subset of S_i for each task t_i (line 27), assigning an invocation time that is sampled from $\mathcal{U}_d(0, t_{\text{zero}} - 1)$ to each provisioned service[5]. Finally, the service times assigned to each task are altered in such a way that there is at least one service with an invocation time of 0 (line 32). This is achieved by the procedure TRUNCATE-ALLOCATION in lines 37–44, which finds the minimum invocation time (m) of a provisioned service for a given task and deducts this from every invocation time. This ensures that there are no unnecessary delays before the first invocation.

[4]This accounts for the fact that we select random neighbours and may miss potentially better solutions. In our work, we set this to 10, in order to balance the quality of the solution with the time taken to find it.

[5]As defined in Section 3.2.2, t_{zero} is the first time step at which the consumer no longer receives any reward for completing a workflow.

Algorithm 5.7 Modified hill-climbing algorithm for finding provisioning allocation α.

procedure FIND-ALLOCATION(W)
  $\alpha \leftarrow$ GENERATE-INITIAL(W) ▷ Generate initial allocation
  $u \leftarrow \tilde{u}(\alpha)$ ▷ Estimate utility
  $n_{\text{failed}} \leftarrow 0$ ▷ Keep track of unsuccessful iterations
  repeat ▷ Main loop
    $n_{\text{failed}} \leftarrow n_{\text{failed}} + 1$ ▷ Increase counter
    $T' \leftarrow T$ ▷ Copy set of tasks
    while $n_{\text{failed}} > 0 \wedge |T'| > 0$ **do** ▷ No better α found and more tasks left?
      $t_i \in T'$ ▷ Random choice
      $T' \leftarrow T' \setminus t_i$ ▷ Remove t_i
      $\mathcal{N} \leftarrow$ GENERATE-NEIGHBOURS(α, t_i) ▷ (see Algorithm 5.8)
      for all $\alpha' \in \mathcal{N}$ **do** ▷ Check all neighbours
        $u' \leftarrow \tilde{u}(\alpha')$ ▷ Utility of neighbour
        if $u' > u$ **then** ▷ If neighbour is more promising...
          $(\alpha, u) \leftarrow (\alpha', u')$ ▷ ...update
          $n_{\text{failed}} \leftarrow 0$ ▷ Reset counter
        end if
      end for
    end while
  until $n_{\text{failed}} \geq$ `maxFailed` ▷ Continue until too many unsuccessful iterations
  return α ▷ Return best allocation found
end procedure

procedure GENERATE-INITIAL(W)
  $\alpha \leftarrow \emptyset$ ▷ Initialise overall allocation
  for all $t_i \in T$ **do** ▷ Consider all tasks
    $A \leftarrow \emptyset$ ▷ Allocation for t_i
    $S_A \in \mathcal{P}(S_i) \setminus \{\emptyset\}$ ▷ Random non-empty subset of S_i
    for all $s_j \in S_A$ **do** ▷ Store each service
      $t \leftarrow$ sample from $\mathcal{U}_d(0, t_{\text{zero}} - 1)$ ▷ Random provisioning time
      $A \leftarrow A \cup \{(s_j, t)\}$ ▷ Store provisioning decision
    end for
    $A \leftarrow$TRUNCATE-ALLOCATION(A) ▷ Truncate
    $\alpha(t_i) \leftarrow A$ ▷ Store task allocation
  end for
  return α
end procedure

procedure TRUNCATE-ALLOCATION(A)
  $A' \leftarrow \emptyset$ ▷ Initialise truncated set
  $m \leftarrow$ minimum provisioning time in A
  for all $(s_j, t) \in A$ **do** ▷ Truncate each mapping
    $A' \leftarrow A' \cup \{(s_j, t - m)\}$
  end for
  return A'
end procedure

Algorithm 5.8 Neighbour generation procedure for a provisioning allocation α.

procedure GENERATE-NEIGHBOURS(α, t_i)
  $A \leftarrow \alpha(t_i)$ ▷ Allocation for t_i
  $A_{\text{prov}} \leftarrow$ set partition of A, partitioned by populations[a]
  $A_x \in A_{\text{prov}}$ ▷ Select random partition
  $(s, t) \in A_x$ ▷ Select random service/time

  $S_{\text{all}} \leftarrow$ set partition of S_i, partitioned as above[b]
  $S_{\text{unprov}} \leftarrow$ as S_{all}, excluding services in A[c]
  $S_{\text{other}} \leftarrow$ as S_{unprov}, excluding services from same population as s[d]
  $S_{\text{same}} \leftarrow$ set containing all unprovisioned services from same population as s[e]

  $\alpha_1, \alpha_2, \ldots, \alpha_8 \leftarrow \alpha$
  $\alpha_1(t_i) \leftarrow \alpha_1(t_i) \setminus \{(s, t)\}$ ▷ Remove (s, ot)

  $S_x \in S_{\text{other}}$ ▷ Select random member
  $s_{n1} \in S_x$ ▷ Select random service
  $\alpha_2(t_i) \leftarrow \alpha_1(t_i) \cup \{(s_{n1}, t)\}$ ▷ Replace s by s_{n1}

  $s_{n2} \in S_{\text{same}}$ ▷ Select random service
  $\alpha_3(t_i) \leftarrow \alpha_3(t_i) \cup \{(s_{n2}, t)\}$ ▷ Add new service s_{n2}

  $\alpha_4(t_i)(s) \leftarrow t - 1$ ▷ Decrease t by 1
  $\alpha_5(t_i)(s) \leftarrow t + 1$ ▷ Increase t by 1
  $\alpha_6(t_i)(s) \leftarrow x$, with x sampled from $\mathcal{U}_d(0, t - 2)$ ▷ Decrease t randomly
  $\alpha_7(t_i)(s) \leftarrow x$, with x sampled from $\mathcal{U}_d(t + 2, t_{\text{zero}} - 1)$ ▷ Increase t randomly

  $s_{n3} \in \bigcup_{S_y \in S_{\text{unprov}}} S_y$ ▷ Pick random unprovisioned service
  $t_{n3} \leftarrow$ sampled from $\mathcal{U}_d(0, t_{\text{zero}} - 1)$ ▷ Random provisioning time
  $\alpha_8(t_i) \leftarrow \alpha_8(t_i) \cup \{(s_{n3}, t_{n3})\}$ ▷ Add new service s_{n3}

  for $j = 1$ to 8 **do**
    $\alpha_j(t_i) \leftarrow$ TRUNCATE-ALLOCATION($\alpha_j(t_i)$) ▷ Truncate new allocation
  end for

  return $\{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6, \alpha_7, \alpha_8\}$ ▷ Return all neighbours
end procedure

[a] $A_{\text{prov}} = \{A_x \mid A_x \subseteq A \wedge A_x \neq \emptyset \wedge \exists P_y \in P \cdot \forall s, t \cdot (((s, t) \in A \wedge s \in P_y) \Leftrightarrow (s, t) \in A_x)\}$
[b] $S_{\text{all}} = \{P_x \mid P_x \in P \wedge P_x \subseteq S_i\}$
[c] $S_{\text{unprov}} = \{P_x \mid P_x \neq \emptyset \wedge \exists P_y \in S_{\text{all}} \cdot \forall s_z \in P_y \cdot (s_z \in P_x \Leftrightarrow \neg \exists t \cdot (s_z, t) \in A)\}$
[d] $S_{\text{other}} = \{P_x \mid P_x \in S_{\text{unprov}} \wedge \neg \exists P_y \in P \cdot (P_x \subseteq P_y \wedge s \in P_y)\}$
[e] $S_{\text{same}} = \{s_x \mid \exists P_y \in P \cdot S_z \in S_{\text{unprov}}, (s \in P_y \wedge S_z \subseteq P_y \wedge s_x \in S_z)\}$

Algorithm 5.9 Overall behaviour of the *detailed flexible* strategy.

```
1:  procedure DETAILED-FLEXIBLE-INITIALISE(W)
2:      α ← FIND-ALLOCATION(M)                          ▷ Find best detailed allocation
3:      T_inv ← ∅                                       ▷ Keeps track of invocation times
4:      T_comp ← ∅                                      ▷ Keeps track of completed tasks
5:      if ũ(α) ≤ 0 then                                ▷ Is utility estimate non-positive?
6:          d_stop ← true                               ▷ ...then abandon workflow
7:      else                                            ▷ ...otherwise continue
8:          d_stop ← false
9:      end if
10: end procedure

11: procedure DETAILED-FLEXIBLE-UPDATE(O)
12:     T_new ← {t_i | ∃s_x · (t_i, s_x) ∈ O}           ▷ Recently completed tasks
13:     T_comp ← T_comp ∪ T_new                         ▷ Add to completed tasks
14: end procedure

15: procedure DETAILED-FLEXIBLE-STOPCONDITION
16:     return d_stop          ▷ Abandon if allocation yields non-positive utility
17: end procedure

18: procedure DETAILED-FLEXIBLE-INVOKESERVICES
19:     for all t_i ∈ T \ T_comp do                     ▷ Consider all uncompleted tasks
20:         if ∀(t_j, t_i) ∈ E · t_j ∈ T_comp then      ▷ Is t_i executable?
21:             if ∃y · (t_i, y) ∈ T_inv then           ▷ Has invocation started?
22:                 t̂_start ← T_inv(t_i)
23:             else
24:                 t̂_start ← t̂                        ▷ Start invoking now
25:                 T_inv(t_i) ← t̂
26:             end if
27:             t̂_i ← t̂ − t̂_start                      ▷ Time steps after t_i was started
28:             for all s_x ∈ {s_x | (s_x, t̂_i) ∈ α(t_i)} do
29:                 INVOKE(s_x, t_i)                    ▷ Invoke service s_x for task t_i
30:             end for
31:         end if
32:     end for
33: end procedure
```

5.3.2.2 Neighbour Generation

The GENERATE-NEIGHBOURS procedure is shown separately in Algorithm 5.8. This procedure creates a set of neighbours of a given allocation α by considering a task t_i. To this end, it first partitions $\alpha(t_i)$ into sets that correspond to particular service populations and selects one of these at random (line 4). The algorithm then picks a random service/time pair, (s, t), from the selected set (line 5). Given this, the following transformations are applied separately to α, in order to generate a set of eight neighbours, $\alpha_1, \alpha_2, \ldots, \alpha_8$:

- α_1 (line 13): Service s is removed.
- α_2 (line 17): Service s is replaced by a random service, s_{n1}, from a different population.
- α_3 (line 20): A new service, s_{n2}, from the same population as s is added to the allocation.
- α_4 (line 22): The invocation time for service s is decreased by a single time step.
- α_5 (line 23): The invocation time for service s is increased by a single time step.
- α_6 (line 24): The invocation time for service s is decreased by a random amount.
- α_7 (line 25): The invocation time for service s is increased by a random amount.
- α_8 (line 29): A random unprovisioned service, s_{n3}, from any suitable population is provisioned at a random time.

In doing this, any impossible transformations are ignored (e.g., when all appropriate service providers are already provisioned, we do not perform the transformations for α_2, α_3 and α_8)[6]. Furthermore, we again alter the provisioning times of all new neighbours to ensure that there are no unnecessary delays (line 32).

We have now described how our algorithm generates candidate solutions and finds a good provisioning allocation by performing a local search. Algorithm 5.9 briefly summarises the overall strategy by showing how these procedures are used in the context of our generic agent framework. In the following section, we outline the utility estimation function, $\tilde{u}$, which is used in lines 3 and 13 of Algorithm 5.7 to estimate the expected utility of a candidate solution.

5.3.3 Utility Prediction

Due to the effectiveness of the heuristic function introduced in the previous chapter, we use the same overall form for $\tilde{u}$ in this chapter (omitting the parameter α for brevity):

$$\tilde{u} = p \int_0^\infty d_W(x) u(x) \, \mathrm{d}x \; - \tilde{c} \tag{5.7}$$

[6]To keep the listing concise, this is not shown in Algorithm 5.8.

where p is the overall success probability of the workflow, d_W is an estimated probability density function for the completion time of the workflow if successful and $\tilde{c}$ is an estimated cost.

Now, p, d_W and $\tilde{c}$ are obtained by aggregating a number of local parameters for each task in the workflow, in the same manner as described in Section 4.4.3.2. However, due to the inclusion of heterogeneous services, it is necessary to adapt the local task calculations to our extended model, and we detail these adaptations in the remainder of this section. As before, we are interested in calculating four key parameters for each workflow task t_i, given an allocation $\alpha(t_i)$:

- The success probability p_i.
- The expected cost $\bar{c}_i$.
- The expected completion time $\bar{t}_i$.
- The variance of the completion time σ_i^2.

To calculate these, we define a number of terms. First, we let $\hat{D}(s_x, t)$ be the probability that a service s_x has completed its service successfully within no more than t time steps of invocation (not conditional on overall success):

$$\hat{D}(s_x, t) = (1 - f(s_x)) \cdot D(s_x, t) \tag{5.8}$$

Furthermore, we let $I_i(\alpha, t) = \{(x, y) \mid (x, y) \in \alpha(t_i) \wedge y \leq t\}$ be the set of provisioned services and associated times that are invoked at most t time steps after task t_i was started. Combining this with Equation 5.8, we can calculate the probability that the task is completed successfully within no more than t time steps, denoted $E_i(\alpha, t)$:

$$E_i(\alpha, t) = 1 - \prod_{(x,y) \in I_i(\alpha,t)} (1 - \hat{D}(x, t - y)) \tag{5.9}$$

To illustrate Equations 5.8 and 5.4, we return to the example allocation shown in Figure 5.3 ($\alpha(t_i) = \{(s_a, 0), (s_b, 0), (s_c, 0), (s_d, 15), (s_e, 40), (s_f, 40), (s_g, 70)\}$), and assume that the provisioned services have failure probabilities, durations and costs as shown in Table 5.1. For example, each of the three initially provisioned services, s_a, s_b and s_c, has a failure probability of $f(s_x) = 0.8$, follows an exponential[7] distribution with mean $\mu = 20$ for its duration and has a cost of 5. In this context, Figure 5.4 shows $\hat{D}(s_x, t)$ for the provisioned services, offset by their respective invocation times, as well as the overall success probability for the task $E_i(\alpha, t)$. This demonstrates how the individual duration distributions influence $E_i(\alpha, t)$ as more services are invoked over time, and how the overall success probability rises quickly by provisioning unreliable services redundantly.

[7] We use Exp(μ) to denote an exponential distribution with pdf $p(x, \mu) = \mu^{-1} e^{-\frac{x}{\mu}}$.

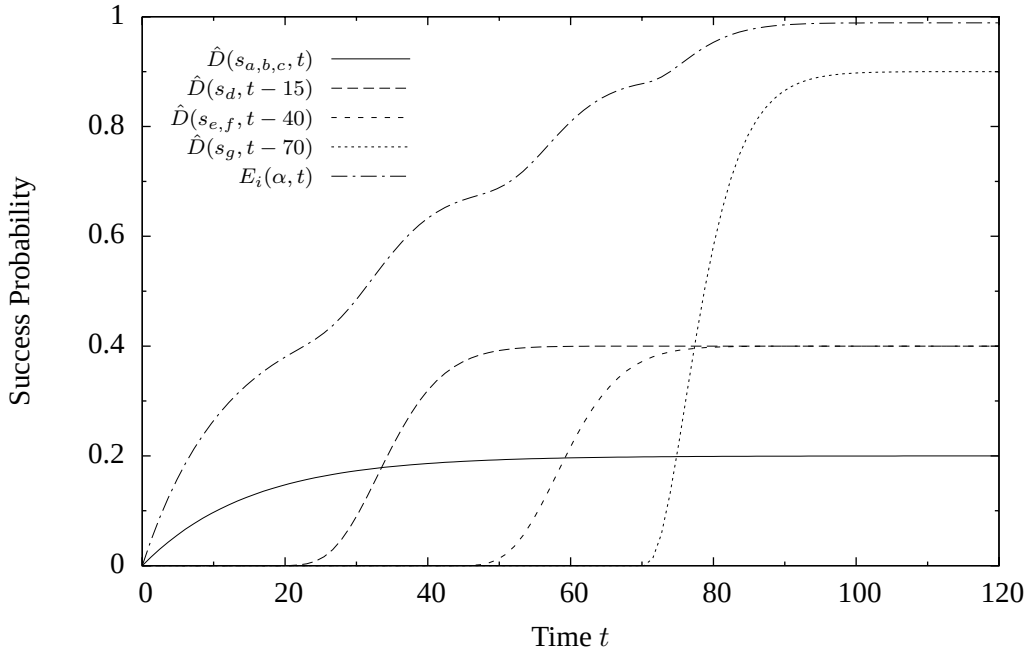


FIGURE 5.4: Cumulative success probabilities for allocation in Figure 5.3.

Service (s_x)	**Fail. Prob.** ($f(s_x)$)	**Duration** ($d(s_x, t)$)	**Cost** ($c(s_x)$)
s_a, s_b, s_c	0.8	Exp(15)	5
s_d, s_e, s_f	0.6	Gamma(10,2)	10
s_g	0.1	Gamma(3,3)	25

TABLE 5.1: Service performance characteristics used in Figure 5.4.

Given $E_i(\alpha, t)$ in Equation 5.4, we can now calculate the four performance parameters given above. To do this, we disregard any service outcomes that occur more than t_{zero} time steps after a task becomes available — this provides us with a limited time horizon to consider, beyond which the consumer is certain to gain no more utility. Hence, the success probability p_i is simply the probability that the task has been successfully completed by any of the invoked services by time t_{zero}:

$$p_i = E_i(\alpha, t_{\text{zero}}) \tag{5.10}$$

For example, if $t_{\text{zero}} = 100$, then the overall success probability of the provisioned task shown in Figure 5.4 is $p_i = E_i(\alpha, 100) = 1 - 0.8^3 \cdot 0.6^3 \cdot 0.1 = 0.99$.

Next, to calculate the expected cost $\bar{c}_i$, we sum the costs of all provisioned services, each multiplied by the probability that the task has not been successfully completed by their respective invocation times:

$$\bar{c}_i = \sum_{(x,y)\in\alpha(t_i)} (1 - E_i(\alpha, y)) \cdot c(x) \tag{5.11}$$

Continuing the example above, the consumer is guaranteed to pay the costs for the first three services, while later costs depend on whether the initial services have been successful: $\bar{c}_i = 3 \cdot 5 + 0.66666 \cdot 10 + 0.36693 \cdot 2 \cdot 10 + 0.12199 \cdot 25 = 32.05$.

In order to calculate the expected completion time $\bar{t}_i$ (again, conditional on overall success), we evaluate all possible outcomes, noting that the consumer receives any service outcomes at discrete time steps:

$$\bar{t}_i = \frac{1}{p_i} \sum_{t=1}^{t_{\text{zero}}} t \cdot (E_i(\alpha, t) - E_i(\alpha, t-1)) \tag{5.12}$$

Using this, it is straight-forward to calculate the associated variance σ_i^2:

$$\sigma_i^2 = -\bar{t}_i^2 + \frac{1}{p_i} \sum_{t=1}^{t_{\text{zero}}} t^2 \cdot (E_i(\alpha, t) - E_i(\alpha, t-1)) \tag{5.13}$$

Applying these to the allocation shown in Figure 5.4 results in an expected completion time of $\bar{t}_i = 33.72$ and a variance of $\sigma_i^2 = 642.44$.

Unfortunately, the calculations described above are less tractable than those presented in Section 4.4.3.1. This is for two reasons. First, we now consider the impact of each individual service on the task performance throughout the duration of the task (while we previously grouped them into multiple service invocations that were independent from each other). Second, Equations 5.12 and 5.13 compute a sum over all time steps to t_{zero}, which is potentially a very large number (depending on the form of the utility function u).

Now, the first issue is an inherent feature of the more complex problem faced in this chapter and means that the time of computing the performance characteristics for each task rises linearly with the number of services provisioned for that task. To address the second issue, we decided to approximate both Equations 5.12 and 5.13 by iteratively dividing the interval $[1, t_{\text{zero}}]$ into smaller segments, each time assuming E_i to be linear on the segments, until a desired minimum error is reached. Specifically, in our work, we approximate Equation 5.12 until it is within 0.1 time steps of the true value, and then we calculate Equation 5.13 over the same segments. This means that our approximations are close to the real values, but require less computational effort.

Given the success probability p_i (Equation 5.10) of each task, the expected cost $\bar{c}_i$ (Equation 5.11), the expected completion time $\bar{t}_i$ (Equation 5.12) and variance σ_i^2 (Equation 5.13), we can now calculate an overall success probability for the workflow, an estimate for its cost and we again use a normal distribution to approximate the workflow duration, as described in Section 4.4.3.2. This allows us to calculate the estimated utility of an allocation α, as shown in Equation 5.7.

This concludes our discussion of the *detailed 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.

5.3.4 Fast Flexible Strategy

A potential drawback of the above strategy is the fact that it explores a large state-space by modifying a single service at a time. This may take a long time to converge to a good solution, especially when there are many services in the system. To address this, we decided to simplify the *full flexible* strategy and consider a coarser decision problem, similar to that discussed in Chapter 4. Hence, rather than considering services 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 services to invoke in parallel (0 means none are invoked).
- $w_{k,i} \in \{1, 2, 3, \ldots, t_{\text{zero}}\}$: the number of time steps to wait before invoking more services from the same population.
- $b_{k,i} \in \{0, 1, 2, \ldots, t_{\text{zero}}\}$: the number of time steps to wait before the first set of services is invoked.

Here, $n_{k,i}$ and $w_{k,i}$ correspond to n_i and w_i used in the previous chapter, while $b_{k,i}$ is introduced to allow the consumer to vary the starting times for different service populations (e.g., to delay the invocation of more expensive services until after cheaper services have been attempted). Again, the provisioning allocation is only followed until the task has been completed successfully. We summarise this allocation as a *simplified provisioning allocation*:

Definition 13 (Simplified Provisioning Allocation)**.** A simplified provisioning allocation is a tuple $\beta = (n, w, b)$, where each component is a function $n, w, b \in (\mathbb{Z} \times \mathbb{Z}) \to \mathbb{Z}$, such that $n(k, i) = n_{k,i}$, $w(k, i) = w_{k,i}$ and $b(k, i) = b_{k,i}$, as defined above.

To give an example, Figure 5.5 shows an allocation for a single task t_i with three possible service populations, P_1, P_2 and P_3. Here, the consumer provisions two services of population P_1 in parallel ($n_{1,i} = 2$), and repeats this invocation every 20 time steps ($w_{1,i} = 20$), starting as soon as the task becomes available ($b_{1,i} = 0$). When the task has not been completed successfully by time step 30, the consumer invokes a single service of the more reliable population P_2 ($b_{2,i} = 30$ and $n_{2,i} = 1$), repeating this every 20 time steps ($w_{2,i} = 20$). Finally, at time step 70, the consumer invokes a service of the most reliable population P_3 ($b_{3,i} = 70$ and $n_{3,i} = 1$), but does this only once (setting $w_{3,i} = t_{\text{zero}}$ ensures that the service will be invoked at most once).

With this simplified allocation, we define the *fast flexible* strategy as follows:

Definition 14 (Fast Flexible Strategy)**.** A consumer following a ***fast flexible*** strategy makes appropriate decisions to provision services for its workflow. To this end, the agent finds a suitable simplified provisioning allocation, so that the agent's predicted profit is maximised.

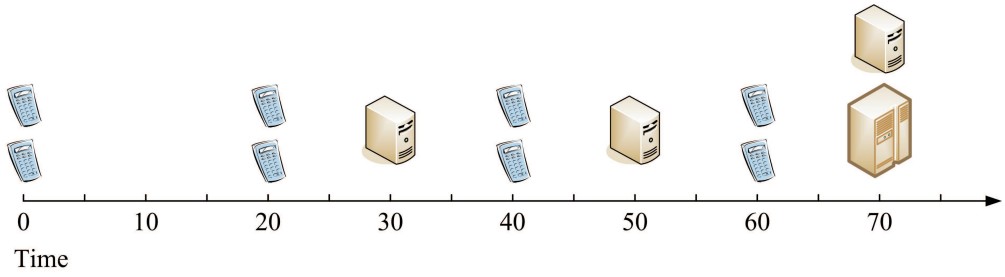


FIGURE 5.5: A simplified provisioning allocation with three service populations.

Algorithm 5.10 Fast algorithm for finding a simplified provisioning allocation β.

```
1:  procedure FAST-FIND-ALLOCATION(W)
2:      g ← number of iterations                          ▷ Pre-defined constant
3:      h ← number of random restarts                     ▷ Pre-defined constant
4:      β_best                                            ▷ Keeps track of best allocation found
5:      u_best ← 0                                        ▷ Keeps track of estimated utility of best allocation found
6:      for c = 1 to h do
7:          β ← FAST-GENERATE-INITIAL(W)                  ▷ Generate initial allocation
8:          u ← ũ(CREATE-DETAILED(β))                     ▷ Estimate utility
9:          n_failed ← 0                                  ▷ Keeps track of contiguous unsuccessful iterations
10:         n_all ← 0                                     ▷ Keeps track of all iterations
11:         repeat                                        ▷ Main loop
12:             (n_failed, n_all) ← (n_failed + 1, n_all + 1)   ▷ Increase counters
13:             T′ ← T                                    ▷ Copy set of tasks
14:             while n_failed > 0 ∧ |T′| > 0 do          ▷ No better β found and tasks left?
15:                 t_i ∈ T′                              ▷ Random choice
16:                 T′ ← T′ \ t_i                         ▷ Remove t_i
17:                 N ← FAST-NEIGHBOURS(β, t_i)           ▷ (see Algorithm 5.8)
18:                 for all β′ ∈ N do                     ▷ Check all neighbours
19:                     u′ ← ũ(CREATE-DETAILED(β′))       ▷ Utility of β′
20:                     if u′ > u then                    ▷ If neighbour is more promising...
21:                         (β, u) ← (β′, u′)             ▷ ...update
22:                         n_failed ← 0                  ▷ Reset counter
23:                     end if
24:                 end for
25:             end while
26:         until n_failed ≥ maxFailed ∨ n_all ≥ g        ▷ Too many iterations?
27:         if u > u_best then
28:             (β_best, u_best) ← (β, u)
29:         end if
30:     end for
31:     return CREATE-DETAILED(β_best)                    ▷ Return best allocation found
32: end procedure
```

Algorithm 5.11 Allocation conversion procedure.

procedure CREATE-DETAILED((n, w, b))
  $\alpha \leftarrow \emptyset$
  for all $t_i \in T$ **do** ▷ Consider each task
    $\alpha(t_i) \leftarrow \emptyset$
    for all $P_k \in \{P_k \in P \mid P_k \subseteq S_i\}$ **do** ▷ ...and population
      $t' \leftarrow b(k, i)$ ▷ First provision time
      if $n(k, i) > 0$ **then**
        $P' \leftarrow P_k$ ▷ Copy population
        while $|P'| > 0 \wedge t' \leq t_{\text{zero}}$ **do** ▷ No services left or time exceeded?
          $c_{\max} \leftarrow \min(|P'|, n(k, i))$ ▷ How many service to provision
          for $c = 1$ to $c_{\max}$ **do**
            $s_x \in P'$ ▷ Random choice
            $P' \leftarrow P' \setminus \{s_x\}$ ▷ Remove s_x from P'
            $\alpha(t_i) \leftarrow \alpha(t_i) \cup \{(s_x, t')\}$ ▷ Provision s_x
          end for
          $t' \leftarrow t' + w(k, i)$ ▷ Advance time
        end while
      end if
    end for
  end for
  return α
end procedure

Algorithm 5.12 Fast initial allocation generation procedure.

procedure FAST-GENERATE-INITIAL(W)
  $(n, w, b) \leftarrow (\emptyset, \emptyset, \emptyset)$ ▷ Initialise β
  for all $t_i \in T$ **do** ▷ Iterate through all tasks
    $P' \leftarrow \{P_k \in P \mid P_k \subseteq S_i \wedge P_k \neq \emptyset\}$ ▷ All suitable populations
    repeat
      for all $P_k \in P'$ **do**
        $n(k, i) \leftarrow$ sample from $\mathcal{U}_d(0, |P_k|)$ ▷ Random number of services
        $w(k, i) \leftarrow$ sample from $\mathcal{U}_d(1, t_{\text{zero}})$ ▷ Random waiting time
        $b(k, i) \leftarrow$ sample from $\mathcal{U}_d(0, t_{\text{zero}})$ ▷ Random initial waiting time
      end for
    until $P' = \emptyset \vee \exists k \cdot n(k, i) \neq 0$ ▷ Ensure a service is provisioned for t_i
  end for
  return FAST-TRUNCATE-ALLOCATION((n, w, b))
end procedure

Algorithm 5.13 Procedure to remove unnecessary waiting times at start of task.

procedure FAST-TRUNCATE-ALLOCATION((n, w, b))
  for all $t_i \in T$ **do**
    $K \leftarrow \{k \mid n(k,i) > 0\}$ ▷ Population indices with provisioned service for t_i
    $b_{\text{min}} \leftarrow \min_{k \in K} b(k,i)$ ▷ Find minimum initial waiting time
    for all $k \in K$ **do**
      $b(k,i) \leftarrow b(k,i) - b_{\text{min}}$ ▷ Deduct b_{min} from all initial waiting times
    end for
  end for
  return (n, w, b)
end procedure

Algorithm 5.14 Neighbour generation procedure for a simplified allocation β.

procedure FAST-NEIGHBOURS(β, t_i)
  $\beta_1, \beta_2, \ldots \beta_{12} \leftarrow \beta$ ▷ Initialise neighbours[a]
  $K \leftarrow \{k \mid P_k \in P \wedge P_k \subseteq S_i\}$ ▷ Indices of all populations for t_i
  $k \in K$ ▷ Pick one at random

  $n_1(k,i) = n_1(k,i) - 1$ ▷ Decrease $n_{k,i}$ by 1
  $n_2(k,i) = n_2(k,i) + 1$ ▷ Increase $n_{k,i}$ by 1
  $n_3(k,i) = \text{sampled from } \mathcal{U}_d(0, n_3(k,i) - 2)$ ▷ Decrease $n_{k,i}$ randomly
  $n_4(k,i) = \text{sampled from } \mathcal{U}_d(n_4(k,i) + 2, |P_k|)$ ▷ Increase $n_{k,i}$ randomly

  $w_5(k,i) = w_5(k,i) - 1$ ▷ Decrease $w_{k,i}$ by 1
  $w_6(k,i) = w_6(k,i) + 1$ ▷ Increase $w_{k,i}$ by 1
  $w_7(k,i) = \text{sampled from } \mathcal{U}_d(1, w_7(k,i) - 2)$ ▷ Decrease $w_{k,i}$ randomly
  $w_8(k,i) = \text{sampled from } \mathcal{U}_d(w_8(k,i) + 2, t_{\text{zero}})$ ▷ Increase $w_{k,i}$ randomly

  $b_9(k,i) = b_9(k,i) - 1$ ▷ Decrease $b_{k,i}$ by 1
  $b_{10}(k,i) = b_{10}(k,i) + 1$ ▷ Increase $b_{k,i}$ by 1
  $b_{11}(k,i) = \text{sampled from } \mathcal{U}_d(0, b_{11}(k,i) - 2)$ ▷ Decrease $b_{k,i}$ randomly
  $b_{12}(k,i) = \text{sampled from } \mathcal{U}_d(b_{12}(k,i) + 2, t_{\text{zero}})$ ▷ Increase $b_{k,i}$ randomly

  for $j = 1$ to 12 **do**
    $\beta_j \leftarrow$ FAST-TRUNCATE-ALLOCATION($\beta_j(t_i)$) ▷ Truncate new allocation
  end for

  return $\{\beta_1, \beta_2, \ldots, \beta_{12}\}$ ▷ Return all neighbours
end procedure

[a] Here, we use (n_x, w_x, b_x) to denote the components of β_x. Hence, $n_x(k,i)$ is the number of parallel services of population P_k, provisioned for task t_i by allocation β_x.

To implement this strategy, we again use a local search, as shown in Algorithm 5.10. This is mostly identical to the *detailed flexible* strategy, but includes a number of minor differences. First, as we are interested in reducing the overall time to find a good solution, we exploit the *anytime* property of our local search and terminate its main loop (lines 11–26) after g iterations. However, as 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. Here, both g and h are constants defined by the user to balance the speed of obtaining a solution with its quality[8].

Furthermore, the *fast flexible* strategy now operates on a simplified provisioning allocation as its candidate solution, but converts this to a detailed provisioning allocation to estimate its utility and to store its final allocation (using the CREATE-DETAILED procedure shown in Algorithm 5.11). The procedure to generate an initial solution has also been adapted and now selects a random allocation for each $n_{k,i}$, $w_{k,i}$ and $b_{k,i}$ (using the FAST-GENERATE-INITIAL procedure shown in Algorithm 5.12).

Similarly, new neighbours are created by randomly varying $n_{k,i}$, $w_{k,i}$ and $b_{k,i}$ for a particular population and task in unit and random steps (as described by the FAST-NEIGHBOURS procedure in Algorithm 5.14). As for the *detailed flexible* strategy, all allocations are altered so that at least one service is invoked immediately when the task becomes available (using the FAST-TRUNCATE-ALLOCATION procedure in Algorithm 5.13). Finally, because the output of the FAST-GENERATE-INITIAL procedure is a detailed provisioning allocation, the basic consumer algorithm for the *fast flexible* is identical to that of the *detailed flexible* (Algorithm 5.9).

Before proceeding to conduct a detailed empirical evaluation of both strategies presented so far, we briefly return to the bioinformatics workflow introduced in Section 3.5 and show how the *detailed flexible* strategy provisions it when there are heterogeneous services.

5.3.5 Illustrative Example

To illustrate the approach developed in this chapter, we again use the bioinformatics workflow shown in Figure 3.6, but now assume that there are several heterogeneous populations that satisfy each service type. To this end, we include populations with the same performance characteristics as those discussed in Section 4.1, as well as two additional populations for each type (see Table 5.2). Generally, we have chosen these to offer certain trade-offs compared to the original population — e.g., services in P_1 are more reliable and faster than those in P_0, but also three times as expensive, while services in P_{22} are slower but also more reliable than those in P_{21}.

As the overall mechanism of the strategies is similar to that in the previous chapter, we only describe the final allocations of the *detailed flexible* strategy[9]. In more detail, Figure 5.6 shows the

[8] We use $g = 10 \cdot |T|$ and $h = 5$, because these values lead to good results in a variety of environments.

[9] We do not treat the *fast flexible* strategy here, as it behaves in a similar manner as the *detailed flexible*.

detailed provisioning allocations[10] that the local search procedure of the *detailed flexible* strategy found during two example runs with the same utility functions as first described in Section 3.5 (one *normal* scenario with a four-hour deadline, and one *urgent* scenario with a 150-minute deadline). The respective cumulative success probabilities of each task, as given by $E_i(\alpha, t)$ in Equation 5.9, are shown in Figure 5.7. Finally, Figure 5.3 summarises the performance parameters of all tasks and the overall workflow, given the two allocations.

Now, the allocations in Figure 5.6 illustrate some general trends of the strategy. First, we notice that all allocations include some redundancy, which is usually achieved by provisioning multiple services in series, but also frequently by provisioning several services in parallel (e.g., for task t_1 in the *normal* case, which is started by invoking three services of the relatively slow and unreliable population P_5 at the same time). Next, the strategy normally relies on several service populations throughout the execution of a task. For example, task t_3 is started in both cases by invoking cheaper services from P_{10} first, but as these run out, the strategy switches over to services from P_9 to continue executing the task. Similarly, for task t_6, the strategy provisions services from P_{18} and P_{20} in parallel, as the latter is very cheap, but still has a small chance of success.

Finally, it is clear that the strategy also adapts appropriately to changing deadlines and utility values. Comparing the allocations for the normal workflow with its urgent counterpart, many tasks in the latter case are provisioned with higher levels of redundancy. For example, rather than the single service provisioned initially for t_7 in the normal case, the strategy immediately provisions two services in parallel, followed soon by a third when the workflow is urgent. This increases the probability of success and also shortens the completion time of the task, as is evident by the cumulative success probability over time, shown in Figure 5.7. For similar reasons, the intervals between successive service invocations for t_3 are shortened in the urgent case. In some cases, the strategy even changes the populations it relies on as the workflow becomes more urgent and valuable. For example, for tasks t_1, t_2 and t_6, it initially provisions services from populations that were not used at all before. These are more expensive and more reliable services that are better suited for the high value and tight deadline of the urgent workflow.

Overall, this flexible adaptation is summarised both by the cumulative success probabilities in Figure 5.7, which rise more quickly in the urgent case, and by the overall characteristics in Table 5.3, where tasks are generally more expensive but also much quicker.

In the following section, we evaluate our strategies over a range of settings.

[10] For brevity, these allocations are only shown to time step 150, but while the strategy provisioned further services at later times, these have little impact on the results for each task.

Service	**Pop.**	**Fail. Prob.**	**Cost ($)**	**Num.**	**Duration**	**Mean (min.)**	**Var.**
BaseCall	P_0	0.2	1	20	Gamma(1.5,2)	3	6
(t_0)	P_1	0.1	3	10	Gamma(1,2)	2	4
	P_2	0.1	1	1	Gamma(1,2)	2	4
GeneAssemble	P_3	0.1	5	25	Gamma(5,2)	10	20
(t_1)	P_4	0	10	1	Gamma(5,2)	10	20
	P_5	0.3	1	10	Gamma(10,2)	20	40
Blast	P_6	0.3	2	50	Gamma(5,3)	15	45
(t_2)	P_7	0.8	0.1	50	Gamma(10,10)	100	1000
	P_8	0.05	10	5	Gamma(2,1)	2	2
LookUp	P_9	0.5	5	10	Gamma(1.5,1.5)	2.25	3.375
(t_3)	P_{10}	0.5	4	2	Gamma(1.5,1.5)	2.25	3.375
	P_{11}	0.75	5	10	Gamma(0.5,0.5)	0.25	0.125
Render	P_{12}	0.1	10	25	Gamma(30,3)	90	270
(t_4, t_7)	P_{13}	0.01	100	5	Gamma(20,2)	40	80
	P_{14}	0.9	1	25	Gamma(30,3)	90	270
Translate	P_{15}	0.7	0.5	50	Gamma(1,1)	1	1
(t_5)	P_{16}	0.7	0.1	50	Gamma(10,2)	20	40
	P_{17}	0	25	10	Gamma(1,1)	1	1
Fold	P_{18}	0.2	10	5	Gamma(3,30)	90	2700
(t_6)	P_{19}	0.05	50	1	Gamma(3,5)	15	75
	P_{20}	0.75	1	1	Gamma(50,2)	100	200
Print	P_{21}	0.2	2	20	Gamma(2,3)	6	18
(t_8)	P_{22}	0.05	2	10	Gamma(5,5)	25	125
	P_{23}	0.9	0.1	30	Gamma(2,3)	6	18

TABLE 5.2: Service types used in the example workflow.

Task	**Success Prob.**	**Cost**	**Mean Duration**	**Variance**	**Utility**
Non-Urgent Workflow ($t_{\max} = 240$, $\delta = 1$, $u_{\max} = 150$)					
t_0	1.00	1.22	3.23	9.16	
t_1	1.00	3.13	17.94	51.76	
t_2	1.00	2.70	41.71	3347.25	
t_3	1.00	8.50	28.93	1745.16	
t_4	1.00	11.24	106.06	2586.45	
t_5	1.00	2.06	3.13	7.46	
t_6	0.99	22.33	72.80	1851.81	
t_7	0.99	11.79	100.77	1326.09	
t_8	1.00	2.41	6.00	25.04	
Overall	0.98	65.26	203.87	3271.32	73.25
Urgent Workflow ($t_{\max} = 150$, $\delta = 20$, $u_{\max} = 1000$)					
t_0	1.00	2.73	2.24	1.94	
t_1	1.00	10.00	10.50	20.70	
t_2	1.00	10.14	4.75	148.28	
t_3	1.00	8.61	11.98	281.14	
t_4	1.00	21.14	83.86	273.00	
t_5	1.00	2.74	1.92	1.46	
t_6	1.00	55.37	18.51	269.73	
t_7	1.00	30.18	80.16	152.04	
t_8	1.00	3.48	4.16	9.26	
Overall	1.00	144.39	117.49	455.14	843.54

TABLE 5.3: Finally provisioned workflows using the *detailed flexible* strategy.

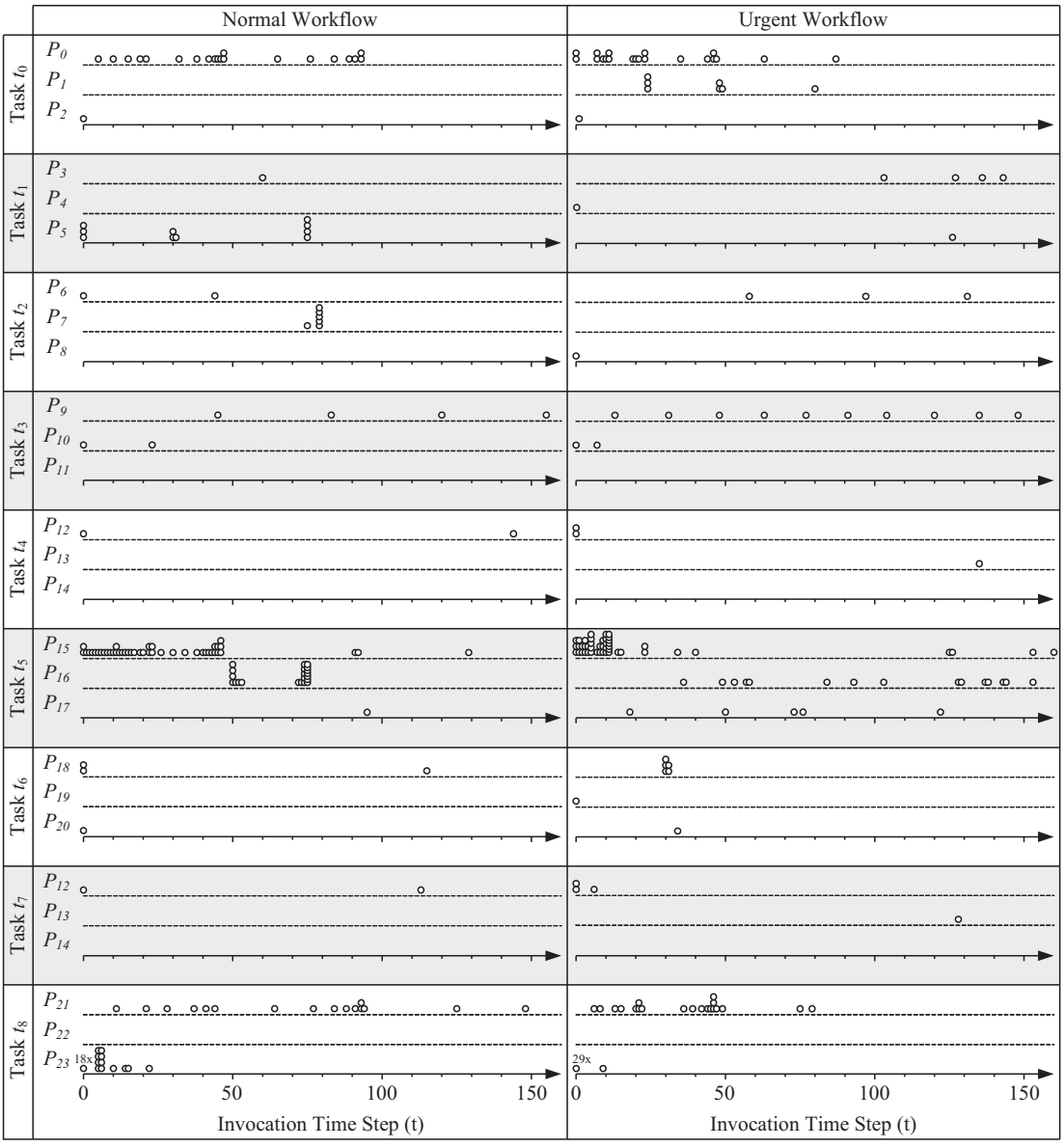


FIGURE 5.6: Detailed provisioning allocations for finally provisioned workflows. Each circle represents one provisioned service instance of a particular population at a certain time.

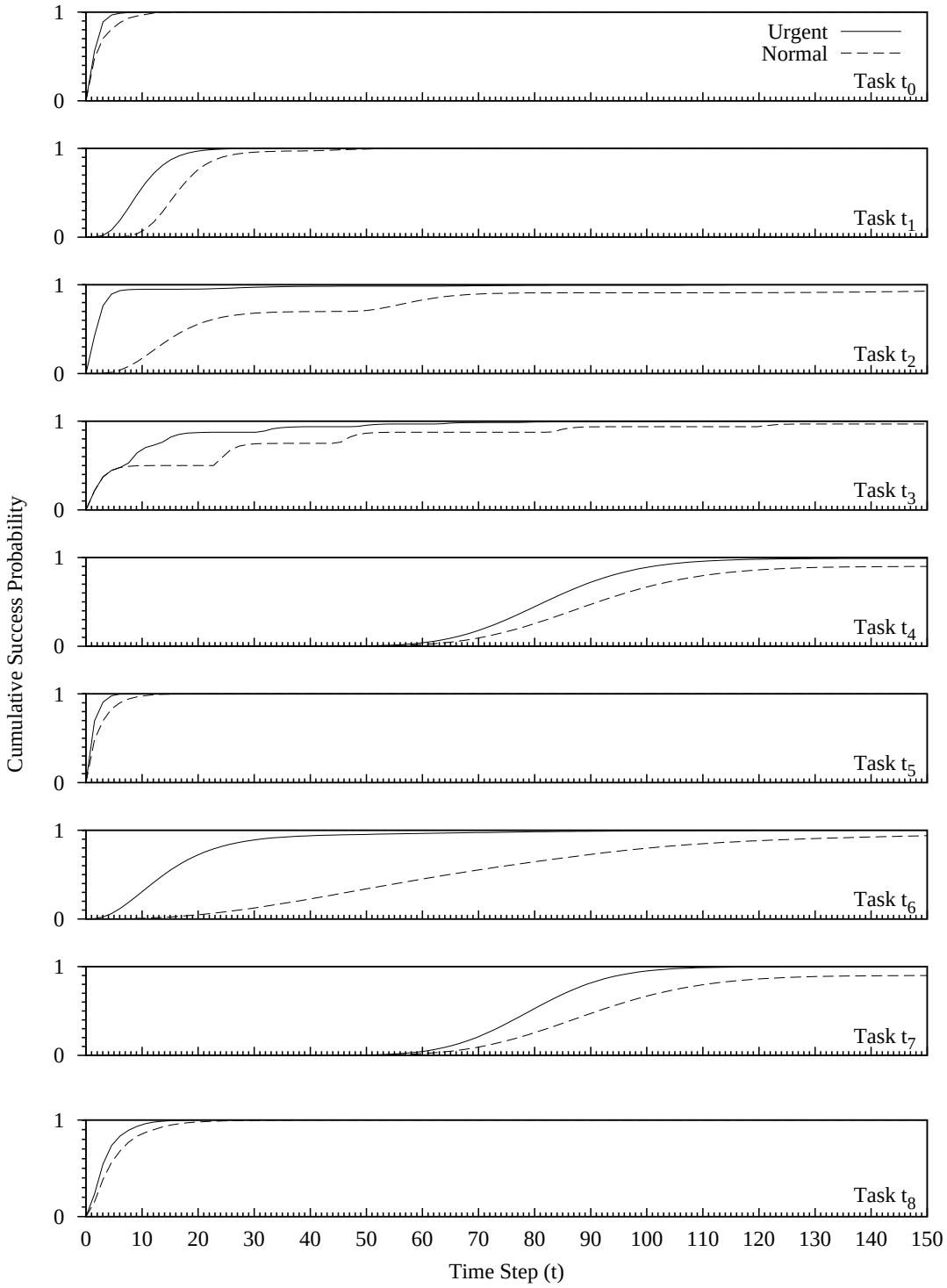


FIGURE 5.7: Cumulative success probabilities of tasks in bioinformatics workflow (*detailed flexible* strategy).

5.4 Empirical Evaluation

In this section, we investigate whether our strategies can achieve significant improvements over currently prevalent approaches in environments where services are heterogeneous and unreliable, and we compare our two strategies (*detailed* and *fast flexible*) to each other. In the following, we first describe our experimental setup and then report our results.

5.4.1 Experimental Setup

To test the performance of our strategies, we simulate a service-oriented system in a similar manner as described in Section 4.5.1, but now assume there are several heterogeneous service populations for the different tasks of a workflow. More specifically, Table 5.4 lists a number of controlled variables and distributions that we use in this section to generate services and workflows. As in the previous chapter, the main variable we vary throughout our experiments is the *average failure probability* of services in the system (denoted as Φ). This allows us to test our strategies in the presence of varying degrees of unreliability. All other variables given in the table are kept static throughout our experiments, both for consistency and to allow a fair comparison between environments with different failure probabilities[11]. As in the previous chapter, we obtain statistical significance by repeating all experiments 1000 times with new randomly generated services and workflows, and carry out appropriate statistical tests and ANOVA at the 95% confidence level (for larger workflows, we occasionally carry out fewer repetitions, as indicated in the text).

In more detail, we first generate five service types, $\mathcal{T} = \{T_1, T_2, T_3, T_4, T_5\}$, and assign an average cost, duration shape and scale to each[12], which are drawn from the respective continuous uniform distributions given in Table 5.4. As in Section 4.5.6 of the previous chapter, when $0 < \Phi < 1$, we also add some variance to the failure probabilities of different service types by assigning a failure probability to each type that is drawn from a beta distribution with a mean equal to Φ and a variance equal to 0.01 (the *inter-type failure variance*). This gives us the broad characteristics of different services types in our simulated environment. To give an example, Table 5.5 shows a set of randomly generated service types with overall $\Phi = 0.5$, demonstrating how this generation process results in service types with highly varying characteristics. For example, T_3 is cheap and slow with a mean duration of 27.32, while T_4 is more than three times as expensive but has a mean duration of only 2.77.

As we are mainly interested in heterogeneity *within* service types in this chapter, we next generate a number of service populations for each of the five service types. More specifically, for each type, we create a number of service populations equal to an integer drawn from the discrete

[11] The values in Table 5.4 were chosen as a plausible workflow scenario. We have carried out a number of experiments with other environments and workflows, and observed the same broad trends as those reported in this chapter.

[12] As before, these represent the parameters k and θ of a gamma distribution.

Variable	**Value**
Environmental Variables	
Average Failure Probability (Φ)	varied
Number of Types	5
Populations Per Type	$\mathcal{U}_d(3, 10)$
Services Per Population	$\mathcal{U}_d(1, 100)$
Average Type Cost	$\mathcal{U}_c(1, 10)$
Average Duration Shape (k)	$\mathcal{U}_c(1, 10)$
Average Duration Scale (θ)	$\mathcal{U}_c(1, 5)$
Inter-Type Failure Variance	0.01
Intra-Type Failure Variance	0.005
Intra-Type Variance	0.05
Workflow Variables	
Workflow Length	10
Workflow Parallelism	0.25
Deadline ($t_{\max}$)	100
Maximum Utility ($u_{\max}$)	1000
Penalty (δ)	50

TABLE 5.4: Controlled variables

Type	**Cost**	**Fail. Prob.**	**Gamma Shape**	**Gamma Scale**
T_1	7.57	0.55	5.45	4.19
T_2	8.06	0.42	3.14	2.47
T_3	2.53	0.40	6.60	4.14
T_4	8.63	0.49	2.59	1.07
T_5	4.12	0.58	4.88	1.78

TABLE 5.5: Example of randomly generated services types (with $\Phi = 0.5$).

uniform distribution $\mathcal{U}_d(3, 10)$, and populate each of these with a number of services drawn from $\mathcal{U}_d(1, 100)$. We then use the type-specific characteristics determined above to further characterise each population. To this end, we draw the failure probability for a population from a beta distribution with a mean equal to the type-specific failure probability and a variance equal to 0.005 (the *intra-type failure variance*). Furthermore, we determine the service cost of each population as the product $2 \cdot y \cdot z$, where y is the type-specific average cost and z is sampled from a beta distribution with mean 0.5 and variance 0.05 (the *intra-type variance*). This is repeated for the duration parameters (resampling z for each). To give an example, Table 5.6 shows some randomly generated populations for T_3 and T_4 from Table 5.5. Here, it is clear that services of type T_3 are generally cheaper and slower than those of type T_4 (as they are based on the overall type characteristics), but there is still considerable heterogeneity between populations of the same type. For example, services in P_i are particularly cheap, but much slower than the more expensive services from P_e.

Finally, workflows always consist of 10 tasks, with a parallelism of 0.25, and we define $u(t)$ by setting $t_{\max} = 100$, $u_{\max} = 1000$ and $\delta = 50$. The matching function τ is created by mapping each task to the services of a randomly chosen service type. This process ensures

Population	**Type**	**Num. of Services**	**Cost**	**Fail. Prob.**	**Gamma Shape**	**Gamma Scale**
P_a	T_3	76	2.38	0.43	8.67	5.79
P_b	T_3	8	3.44	0.33	6.99	2.91
P_c	T_3	87	2.82	0.48	6.86	2.82
P_d	T_3	61	2.90	0.45	4.60	4.05
P_e	T_4	52	11.52	0.50	0.39	0.99
P_f	T_4	63	13.60	0.50	4.03	1.19
P_g	T_4	58	8.12	0.36	4.61	0.66
P_h	T_4	59	5.90	0.47	0.70	0.88
P_i	T_4	18	3.77	0.43	1.98	0.83
P_j	T_4	28	10.72	0.47	4.52	1.08

TABLE 5.6: Example of randomly generated populations (based on the types in Table 5.5).

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. To evaluate our strategies, we compare them to the following seven benchmark strategies:

- **naïve:** As discussed in Section 4.2, this strategy provisions a single service for each task in the workflow (chosen randomly from all matching services).
- **hybrid(n,w):** As discussed in Section 4.3.3, this strategy provisions multiple services for each task in the workflow, but does so in a fixed manner without explicitly considering the service parameters. Specifically, the *hybrid(n,w)* strategy provisions sets of n random service providers in parallel, every w time-steps after a task becomes available.
- **local:** For each task, this strategy selects the service that maximises a weighted sum of its performance characteristics, as described in Section 5.2.1.
- **adaptive local(n,w):** Similar to the above, this strategy selects the n best services and repeats this provisioning every w time steps until the task is completed.
- **global:** This strategy provisions a single service for each workflow task, in order to maximise a weighted sum of aggregated performance characteristics, as discussed in Section 5.2.2. We implemented this strategy using the ILOG CPLEX optimisation package.
- **adaptive global(w):** As above, but this strategy treats services as failed when they take w time steps or longer, and re-provisions them accordingly.
- **best hybrid/local/global:** To approximate the upper bound achievable by any of the parameterised strategies (*hybrid(n,w)*, *adaptive local(n,w)* and *adaptive global(w)*), we test a large range of possible parameters[13] and then select the best performing strategy for a given environment (i.e., for each Φ value).

[13]To limit the time required to compute this upper bound, we test each $(n, w) \in \{1, 2, \ldots, 19, 20\} \times \{1, 2, 3, 4, 5, 10, 15, \ldots, 45, 50, 60, \ldots, 90, 100, 150, \infty\}$. Although this means we that we do not test *all* possible parameters, we observed that the strategies do not generally achieve significantly different results between these intervals.

5.4.2 Hypotheses

During our empirical investigation, we are interested in four hypotheses. The first seeks to establish whether simple, non-flexible redundancy is potentially beneficial in the environments we consider. The last three evaluate our flexible strategies, both by comparing them to the non-flexible approaches and to each other.

Hypothesis 5. Adopting the *hybrid(n,w)* strategy can lead to a significant improvement in the average profit over the *naïve* strategy, when appropriate values for n and w are chosen.

Hypothesis 6. The *detailed flexible* strategy achieves a higher average profit than the *hybrid(n,w)* strategy over all environments considered, and for all n and w.

Hypothesis 7. The *detailed flexible* strategy achieves a higher average profit than any non-adaptive QoS-based strategy over all environments considered.

Hypothesis 8. The *detailed flexible* strategy achieves a higher average profit than any adaptive QoS-based strategy.

Hypothesis 9. The *fast flexible* strategy finds a solution in less time than the *detailed flexible* strategy.

Hypothesis 10. The *fast flexible* strategy does not achieve a significantly different average profit from the *detailed flexible* strategy.

In the following sections, we consider each of the above hypotheses separately.

5.4.3 Hybrid Results (Hypothesis 5)

During our first set of experiments, we were interested in evaluating the performance of the *hybrid(n,w)* strategy, in order to ascertain whether redundant provisioning could be used to deal with uncertain service providers (Hypothesis 5). To examine this, we compared the performance of the *naïve* strategy to the *hybrid(n,w)* strategy with various parameter choices for n and w. Figure 5.8 shows our results in four distinct environments, with varying values of Φ. More specifically, Figure 5.8(a) shows an environment where services never fail ($\Phi = 0.0$), then Figures 5.8(b) and 5.8(c) show environments where services increasingly fail ($\Phi = 0.3$ and $\Phi = 0.6$), while Figure 5.8(d) shows the case where services are guaranteed to fail ($\Phi = 1.0$). In these figures, the *naïve* strategy is marked by a circle (the left-most point with $n = 1$, $w = \infty$).

It is clear that there are choices for n and w that significantly outperform the *naïve* strategy. However, the figures show that the best-performing strategies are different in all environments. For example, when services never fail, the highest average profit (727.83 ± 8.96) is achieved by *hybrid(3,60)*. When the failure probability rises to 0.3, *hybrid(5,45)* obtains the highest profit (594.41 ± 19.20), and at $\Phi = 0.6$, the best performing strategy is *hybrid(8,30)* with a profit of 315.07 ± 25.91. This is because the higher level of redundancy allows the consumer to cope better with uncertainty, but incurs unnecessary costs when services are reliable. Similar to our

results in the previous chapter, this indicates the need for a more flexible way of provisioning services than the static method employed by the hybrid strategy, and it provides us with a basic benchmark to evaluate our work against.

Finally, Figure 5.9 compares some representative *hybrid(n,w)* strategies to the *naïve* strategy over a range of environments with different failure probabilities. It is obvious here that the *naïve* strategy is outperformed[14] by other strategies, thus validating Hypothesis 5.

5.4.4 Flexible Provisioning Results (Hypothesis 6)

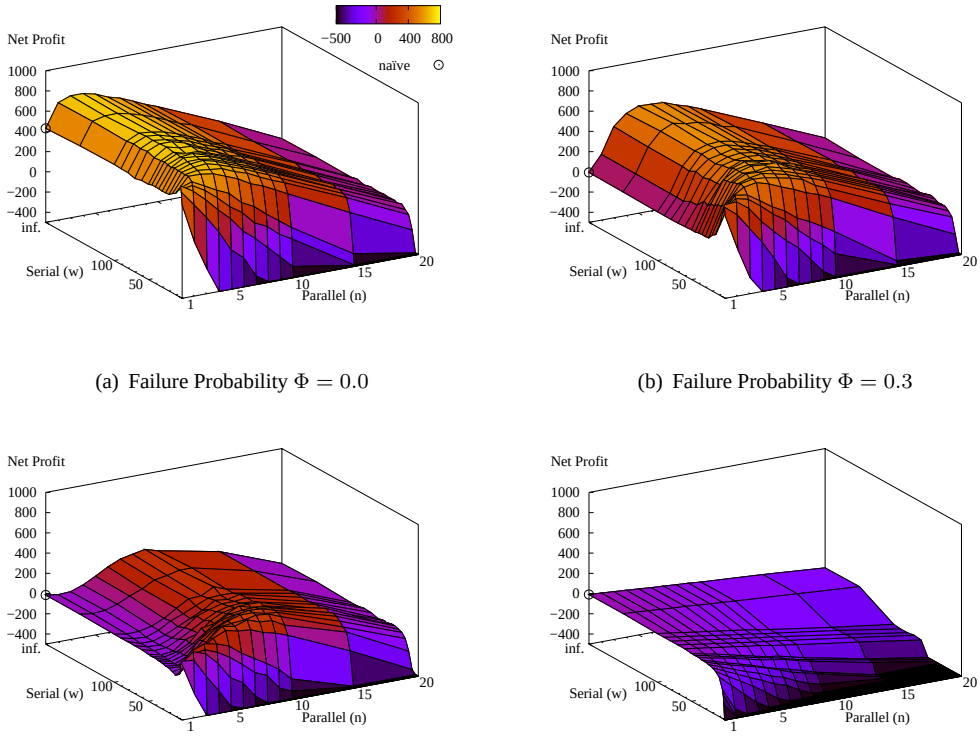
Next, we compared the performance of our flexible approach to the *hybrid(n,w)* strategy over a range of environments (Hypothesis 6). To this end, Figure 5.10 shows the average profit of the *detailed flexible* strategy and the *best hybrid* strategy, which represents an upper bound achievable by any *hybrid(n,w)* strategy. For reference, this graph also includes the *naïve* strategy, which does not address uncertainty or service heterogeneity in any way.

Here we note that the *best hybrid* strategy performs well in most environments, with its profit decreasing gradually as the average failure probability rises. It only starts making a small net loss at $\Phi = 0.8$ and beyond (at which point it is equivalant to the *naïve* strategy, as this invokes the smallest number of services). However, it should be noted that the *best hybrid* strategy is a purely speculative approach that is based on retrospectively selecting the best parameter for n and w, and so it is not a viable option in realistic scenarios.

Now, the *detailed flexible* strategy performs even better than the *best hybrid* strategy, and does so consistently over all values for Φ we tested. When there is no uncertainty in the system, it achieves almost maximum utility as it is able to select the cheapest providers available, and thus obtains an average profit that is around twice as high as the *naïve* approach. Beyond this, the average profit decreases slowly, and continues to make a positive profit even when $\Phi = 0.8$ and $\Phi = 0.9$, at which point all other strategies make an overall loss. At $\Phi = 1.0$, the strategy makes neither a loss nor a profit, as it recognises the workflow as infeasible and thus makes no investments.

The good performance of the *detailed flexible* strategy is due to two reasons. First, the strategy is able to flexibly provision multiple services redundantly for its tasks when there is uncertainty, and it is able to re-provision services when they have apparently failed. In this way it operates in a similar manner as the *best hybrid* strategy, but uses a decision-theoretic framework and knowledge about its environment to pick appropriate levels of redundancy and time intervals between invocations (rather than determining these retrospectively). Second, the *detailed flexible* strategy is able to exploit the heterogeneity of services (and tasks) and pick the most suitable services

[14] An ANOVA *rejects* H_0 that all means are equal for failure probability 0.3 ($F = 694.11$ and $p < 0.001$). A second t-test to compare *naïve* with *hybrid(10,∞)* (for example) *rejects* H_0 that both strategies achieve the same net profit in favour of H_A that *hybrid(10,∞)* achieves a higher net profit ($T = 55.36$ and $p < 0.001$).



(a) Failure Probability $\Phi = 0.0$

(b) Failure Probability $\Phi = 0.3$

(c) Failure Probability $\Phi = 0.6$

(d) Failure Probability $\Phi = 1.0$

FIGURE 5.8: Average profit of *hybrid(n,w)* and *naïve* strategies in environments with varying values of Φ (shading indicates profit).

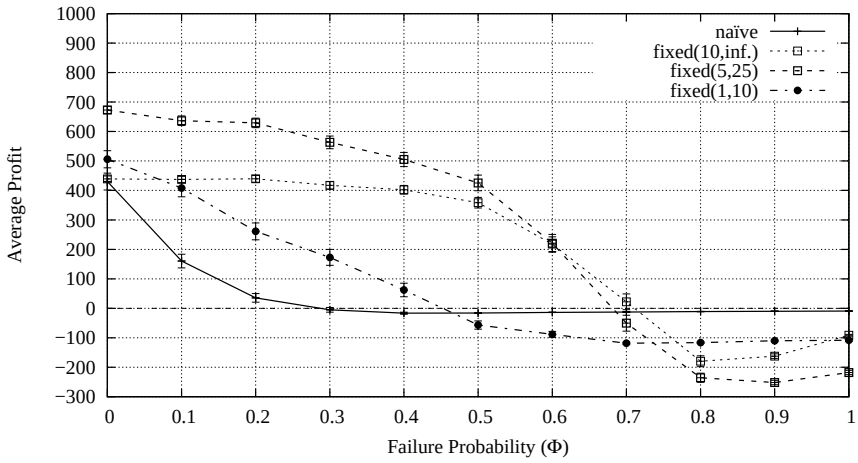


FIGURE 5.9: Some representative *hybrid(n,w)* strategies compared to the *naïve*.

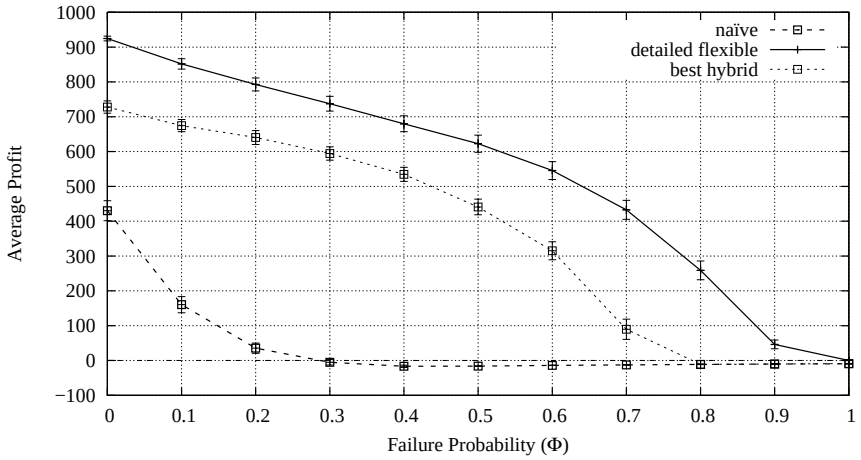


FIGURE 5.10: Performance of the *detailed flexible* strategy.

available, or even rely on multiple services of different heterogeneous populations at the same time.

Averaged over all values for Φ, the *detailed flexible* strategy achieves an average net profit of 535.66 ± 8.29, while the *best hybrid* strategy achieves only 362.49 ± 7.74. This supports[15] Hypothesis 6. For comparison, the best individual hybrid strategy we tested, *fixed(5,25)* (as shown in Figure 5.9), achieves an average profit of 263.4 ± 9.1, while the *naïve* strategy only achieves 48.33 ± 4.24.

5.4.5 Non-adaptive QoS-based Provisioning Results (Hypothesis 7)

In our next set of experiments, we considered the performance of the *local* and *global* strategies. As stated by Hypothesis 7, we expected these to perform worse than our *detailed flexible* strategy, because they do not adapt to failures at all.

Figure 5.11 shows the results of these strategies (along with the *naïve* and *detailed flexible* strategies, for reference). Clearly, they consistently perform better than the *naïve* approach, as they provision providers based on their performance characteristics. Hence, they tend to complete workflows faster, at a lower cost and with a higher success probability. The *global* strategy here performs slightly better than the *local* strategy when the failure probability is low, because the former reasons explicitly about the overall duration with respect to the duration constraint t_{zero} and thus generally finishes workflows earlier. Despite this, when Φ reaches 0.4, both begin to make a net loss as they do not react to failures and complete only around 1% of all workflows successfully.

[15] A t-test *rejects* H_0 that the *detailed flexible* strategy achieves the same average net profit as the *best hybrid* (over all environments) in favour of H_A that *detailed flexible* achieves a higher net profit with $T = 29.94$ and $p < 0.001$. Further t-tests for all individual failure probabilities confirm this result (all with $p < 0.001$).

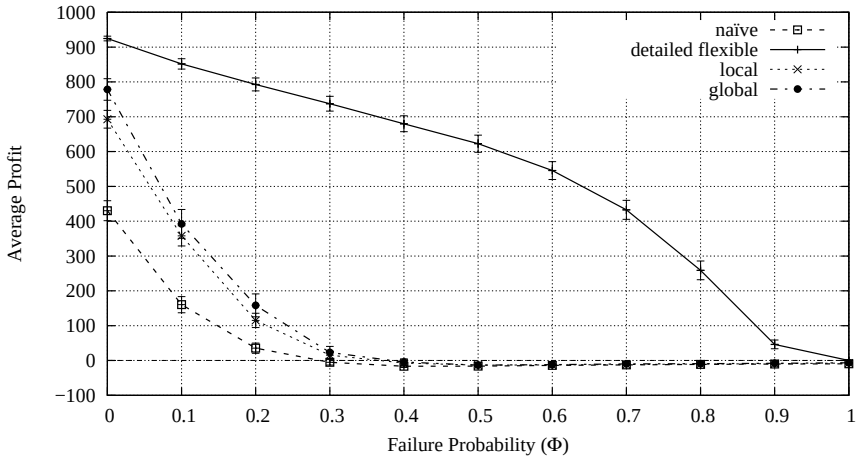


FIGURE 5.11: Performance of the non-adaptive QoS-based strategies.

It is obvious that the *detailed flexible* strategy performs better than the *local* and *global* strategies, which, respectively, achieve 117.18 ± 8.64 and 100.94 ± 5.79. This supports[16] Hypothesis 7.

5.4.6 Adaptive QoS-based Provisioning Results (Hypothesis 8)

Given the poor performance of the non-adaptive QoS-based approaches, we next investigated how their adaptive counterparts perform. We expected these to generally achieve a lower net profit than our *detailed flexible* strategy (Hypothesis 8), as they rely on simple weighted sums to guide the service provisioning.

Figure 5.12 shows a number of example *adaptive local(n,w)* strategies: *adaptive local(2,90)*, *local(4,40)* and *local(11,25)*. We selected these particular parameters, because they are some of best-performing strategies we tested and because they display the general trends of the strategy as higher redundancy and shorter time-outs are introduced. Much as we observed earlier with the *parallel(n)* and *serial(w)* strategies, it is clear that the strategies here are well suited only for particular environments. For example, the *adaptive local(2,90)* performs very well when services always succeed, but as the failure probability rises, more aggressive strategies that rely on higher redundancy quickly begin to outperform it. Even when the failure probability is high, some strategies can achieve a good positive profit, but typically make a large loss when workflows become infeasible (this is most evident with the *adaptive local(11,25)* strategy, as the failure probability rises above 0.8.

The figure also shows the hypothetical *best local* strategy. This performs very well and is clearly better than the *best hybrid* strategy from the preceding section, as it chooses services based on their quality parameters rather than randomly. It also approaches the performance of the *detailed flexible* strategy. In fact, although the *detailed flexible* achieves a higher average profit than the

[16] A t-test *rejects* H_0 that the *detailed flexible* strategy achieves the same average net profit as the *local* strategy in favour of H_A that *detailed flexible* achieves a higher net profit with $T = 84.33$ and $p < 0.001$. The corresponding H_0 comparing the *detailed flexible* strategy and the *global* is rejected with $T = 68.54$ and $p < 0.001$.

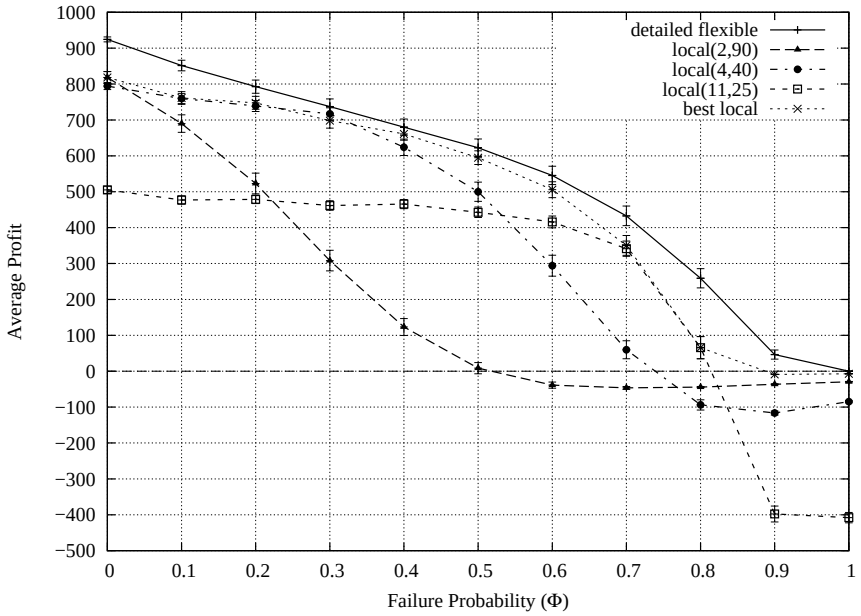


FIGURE 5.12: Performance of the adaptive local QoS-based strategies.

best local over all failure probabilities, the difference is not statistically significant[17] at $\Phi = 0.4$ and $\Phi = 0.5$. However, as the *best local* is a purely speculative strategy, the results confirm that our *detailed flexible* strategy adopts suitable levels of redundancy for its respective environment.

Similarly, the shape of the graph suggests that the *detailed flexible* strategy has a higher advantage over the *best local* strategy towards the extremes of Φ. We believe this is because our strategy increasingly relies on provisioning allocations that cannot be expressed within the parameters of any *adaptive local(n,w)* strategy. In particular, when Φ is high, our strategy mixes different service populations and is generally more sensitive to small differences between the performance characteristics of different service types. On the other hand, when Φ is low, our strategy typically provisions single services, followed, after some long time-out period, by a number of redundant services (to ensure the task is completed).

Next, Figure 5.13 shows various representative *adaptive global(w)* strategies (each experiment was repeated 250 times due to the more time-intensive nature of the strategies). These follow similar trends as the *adaptive local(n,w)* strategies — initially, the strategies with longer time out values perform better, but as the failure probability increases, the more aggressive strategies achieve a higher profit. As before, none of the strategies is particularly well suited for all environments and sometimes they even incur a substantial loss as the failure probability rises. The *best global* again provides an appropriate upper bound, which is here slightly lower than the *best local* described above. This is because the global approaches do not include explicit redundancy and so have to rely on extremely short time out values in order to meet their deadlines.

[17] A t-test *accepts* H_0 that the net profits are equal at $\Phi = 0.4$ and $\Phi = 0.5$ with $T = 1.27$, $p = 0.203$ and $T = 1.75$, $p = 0.080$, respectively. It is rejected at all other failure probabilities with $p = 0.022$ or less.

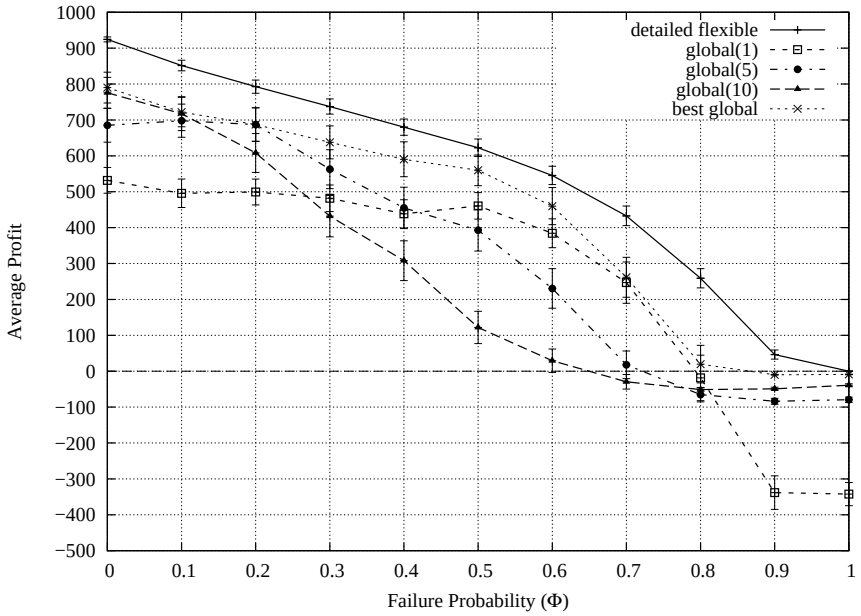


FIGURE 5.13: Performance of the adaptive global QoS-based strategies.

To conclude, when averaging over all environments, the *best local* strategy achieves a profit of 482.98 ± 8.26 and the *best global* strategy achieves a slightly lower profit of 428.13 ± 17.04. Both strategies are outperformed by our *detailed flexible* strategy (with average profit 535.66 ± 8.29), thus supporting[18] our Hypothesis 8.

5.4.7 Fast Flexible Search Time (Hypothesis 9)

Given the promising results of the *detailed flexible* strategy, we were interested in how it compares to the *fast flexible* strategy. Due to the simplified search space, we expected *fast flexible* to reach a solution faster (Hypothesis 9).

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 Φ, the average time of *detailed 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%. This supports[19] Hypothesis 9. 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. The overall better convergence time of the *fast flexible* strategy is not surprising, as we have reduced the search space and introduced an artificial cut-off time for its hill-climbing procedure.

[18] A t-test *rejects* H_0 that $\bar{u}_{\text{detailed}} = \bar{u}_{\text{local}}$ in favour of H_A that $\bar{u}_{\text{detailed}} > \bar{u}_{\text{local}}$ with $T = 8.82$ and $p < 0.001$. Similarly, a t-test *rejects* H_0 that $\bar{u}_{\text{detailed}} = \bar{u}_{\text{global}}$ in favour of H_A that $\bar{u}_{\text{detailed}} > \bar{u}_{\text{global}}$ with $T = 17.73$ and $p < 0.001$.

[19] A t-test *rejects* H_0 that $\bar{t}_{\text{fast}} = \bar{t}_{\text{detailed}}$ in favour of H_A that $\bar{t}_{\text{fast}} < \bar{t}_{\text{detailed}}$ with $T = 126.56$ and $p < 0.001$.

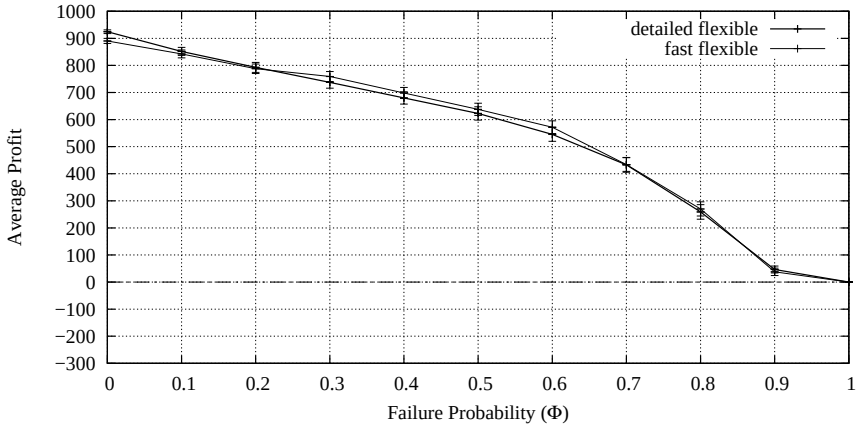


FIGURE 5.14: Performance comparison of *fast*/*detailed* strategies.

5.4.8 Fast Flexible Profit (Hypothesis 10)

While taking less time to converge to a good solution, we were next interested in how the quality of the solution obtained by the *fast flexible* strategy compares to the *detailed flexible* strategy (Hypothesis 10).

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 5.14, 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 (*detailed flexible*) and 539.0 ± 8.03 (*fast flexible*). Hence, their overall performance is not significantly different, supporting[20] Hypothesis 10. This trend continues when comparing the results individually for all values for $\Phi > 0.0$. The only exception is at $\Phi = 0.0$, when the *detailed 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 $\Phi = 0.0$, the *detailed flexible* strategy achieves an average net profit of 924.5 ± 6.8 while the *fast flexible* achieves 890.4 ± 9.4.

This result indicates that it is not necessary to search the full space of all possible provisioning allocations. Rather, it is sufficient to select an appropriate number of providers to provision in parallel as well as a time-out value after which more services are provisioned. This is because a service consumer can generally gain little from altering the number of parallel providers or the frequency of provisions as times passes compared to the overall gain that the introduction of redundant services offers.

[20] A t-test *accepts* H_0 that the net profits, averaged over all environments, are equal ($\bar{u}_{\text{detailed}} = \bar{u}_{\text{fast}}$) with $T = 0.57$ and $p = 0.572$.

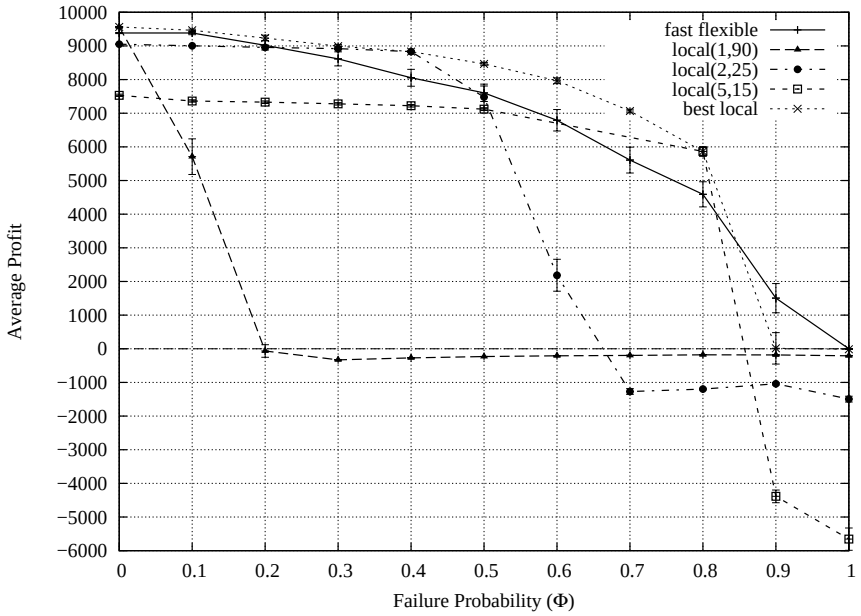


FIGURE 5.15: Performance of the adaptive local QoS-based strategies with large workflows.

5.4.9 Performance in Complex Environments

So far, we examined the performance of our strategy in environments with small workflows. To investigate whether the trends shown in previous sections hold for larger workflows, we repeated the above experiments for workflows consisting of 100 tasks, with a maximum utility $u_{\max} = 10000$, deadline $u_{\max} = 1000$ and penalty $\delta = 50$. All other parameters remain the same except for the inter-type failure variance, which is set to 0 (otherwise, there is a high likelihood that some types have a failure probability close to 1 even when $\Phi < 1$, thus making the entire workflow infeasible). We also now generate 25 distinct service types and repeat all experiments 250 times (due to the more complex nature of these workflows).

Figure 5.15 plots the results of several representative *adaptive local(n,w)* strategies, their upper bound *best local* and our flexible strategy (due to the more complex environments and the small difference between our strategies, we only show the results of the *fast flexible* strategy here). The broad trends are similar as those described in Section 5.4.6. However, we now note that in most environments, there is some *adaptive local(n,w)* that achieves a higher average profit than our flexible approach. The main reason for this is that our strategy usually chooses provisioning allocations with few initial providers and higher redundancy after some time has passed, thus resulting in a high task duration and variance. Although such an allocation results in a high estimated utility (in particular due to a low overall cost), our critical path technique is slightly inaccurate and underestimates the completion time. This inaccuracy is exacerbated in this case by the high task duration variances, which increase the probability that tasks not on the estimated critical path will become critical at run-time. Hence, we observed that our strategy often finishes a short time after $t_{\max}$, thus incurring a penalty on its eventual reward. The *adaptive local(n,w)*

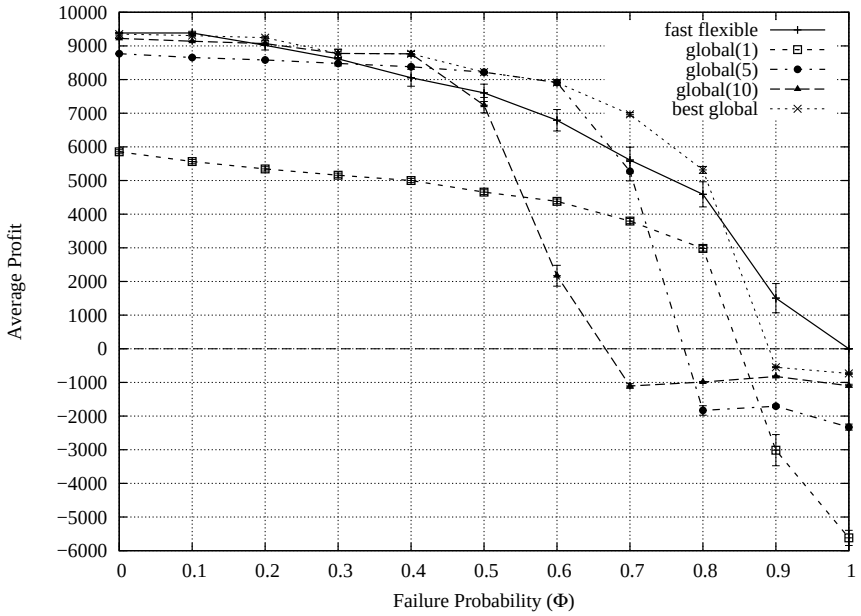


FIGURE 5.16: Performance of the adaptive global QoS-based strategies with large workflows.

strategies, on the other hand, almost always finish some time before $t_{\max}$ and thereby achieve the maximum utility $u_{\max}$.

These results are mirrored by the *adaptive global* strategies, as shown in Figure 5.16. Here, there are again some strategies for most environments that achieve a higher profit than our *fast flexible* strategy. This is mostly for the same reasons as mentioned above, and additionally the *adaptive global* strategies adapt dynamically to new information as it becomes available. More specifically, the strategy frequently replans during execution and thereby takes into account the performance of past services. Thus, it can react to services that take longer than expected (and therefore become part of the critical path), which the *flexible* strategy does not currently do.

In conclusion, these results show that our strategy still manages to obtain a high average profit even when services are highly uncertain. However, we noted that it is often outperformed by hypothetical strategies that retrospectively choose time-out and redundancy parameters. This contradicts our Hypothesis 8 as the *best local* and *best global* achieve a higher average profit than the *fast flexible* strategy.

Nevertheless, the results indicate that our strategy performs well in selecting good provisioning allocations without the need for manually setting parameters and it typically achieves a profit that is close to the *best local* and *best global* strategies. In more detail, the *fast flexible* strategy achieves an average profit of 6413.87 ± 139.35, while the *best local* obtains 6942.23 ± 126.94 and the *best global* achieves 6598.73 ± 135.88.

Furthermore, while setting the best parameters for n and w in some environments results in a higher average profit than the *fast flexible* strategy, the same parameters often perform very badly in other scenarios (e.g., the adaptive *adaptive local(5,15)* strategy performs very well at

Variable	**Value**
Environmental Variables	
Average Failure Probability (Φ)	varied
Number of Types	12
Populations Per Type	$\mathcal{U}_d(3, 10)$
Services Per Population	$\mathcal{U}_d(1, 200)$
Average Type Cost	$\mathcal{U}_c(1, 80)$
Average Duration Shape (k)	$\mathcal{U}_c(1, 40)$
Average Duration Scale (θ)	$\mathcal{U}_c(1, 10)$
Inter-Type Failure Variance	0.0
Intra-Type Failure Variance	0.005
Intra-Type Variance	0.05
Workflow Variables	
Workflow Length	50
Workflow Parallelism	0.25
Deadline ($t_{\max}$)	2000
Maximum Utility ($u_{\max}$)	25000
Penalty (δ)	37.5

TABLE 5.7: Controlled variables for complex environments.

$\Phi = 0.8$, but incurs a severe loss at $\Phi = 0.9$). Thus, our results support a weaker version of Hypothesis 8:

Hypothesis 11. The *full flexible* strategy achieves a higher profit (averaged over all environments considered) than any parameterised adaptive QoS-based strategy.

Finally, as a result of observing that both the local and global strategies typically finish comfortably within the deadline, we believe that the scenario covered so far in this section does not represent a particularly challenging environment, where simple strategies that do not reason specifically about the cost of failures (both the financial cost and the additional time incurred) can perform well. For this reason, we briefly discuss a more challenging case below.

In these experiments, we alter a number of our controlled variables, as shown in Table 5.7 to represent a more challenging environment, where services are potentially more expensive (we increase the maximum cost from 10 to 80) and service times are significantly longer and display a higher variance than in the scenarios considered so far (we quadruple the maximum shape and double the maximum scale parameters). We believe that the *fast flexible* is more suitable for such environments, as it is able to determine how to balance these different, possibly highly variable qualities. Again, we repeat all experiments 250 times to obtain statistical significance.

Figure 5.17 shows the performance of the *fast flexible* and a number of *adaptive local* strategies in these environments. Clearly, the *fast flexible* now outperforms all other strategies. This is due on one hand to the more heterogeneous environment that requires the agent to carefully balance the benefit of redundant provisioning with the associated cost. On the other hand, the more challenging deadline (given the significantly longer service durations) causes the *local* strategies to frequently miss the overall deadline $t_{\max}$ and thus incur a penalty. In fact, the individual local

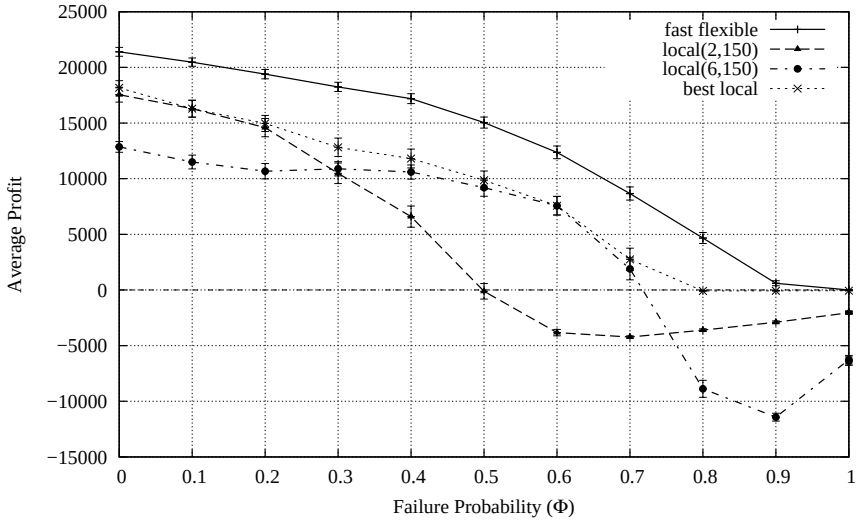


FIGURE 5.17: Performance of the adaptive local QoS-based strategies in highly heterogeneous environments.

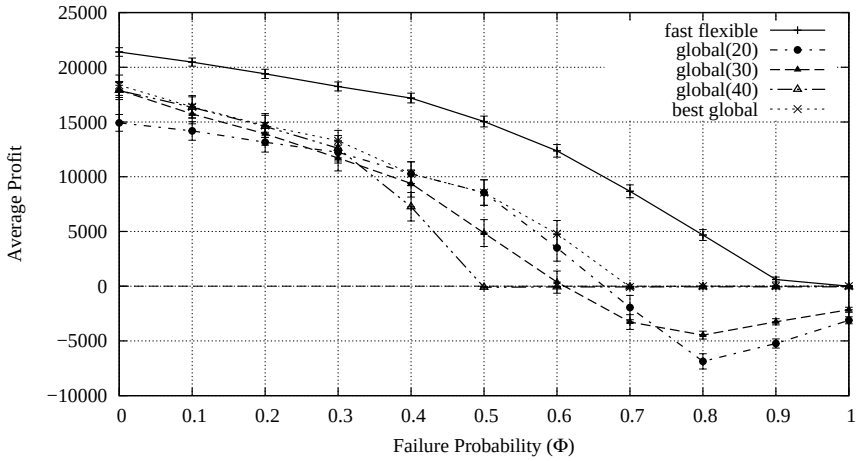


FIGURE 5.18: Performance of the adaptive global QoS-based strategies in highly heterogeneous environments.

strategies often incur a significant negative loss as they provision many services, but do not complete the workflow in time.

These broad trends are mirrored by the *adaptive global* strategies shown in Figure 5.18. The overall profit of these strategies is slightly lower than the local strategies and their overall loss at high failure probabilities is also lower (as they explicitly reason about their budget limit and as they do not provision redundant services). However, they are still consistently outperformed by the *fast flexible* strategy over all failure probabilities.

Concluding this section, we have observed that there are certain environments where simple strategies may outperform the *fast flexible* strategy. However, finding these simple strategies is non-trivial and picking the wrong one may result in a significant loss. Furthermore, when considering more heterogeneous environments with challenging deadlines, we found that the flexible strategy quickly outperforms any other existing strategy. Finally, we noted that the *fast*

flexible strategy achieves good results throughout all environments considered, never incurs a net loss (unlike most other strategies) and often achieves a strictly positive profit even when services are highly unreliable.

Given this, in the following section, we aim to quantify how close the flexible provisioning approach can come to highest possible overall profit.

5.4.10 Optimality of Flexible Provisioning

To examine how well the *fast flexible* strategy performs compared to the optimal, we carried out another set of experiments. However, due to the inherent difficulty of computing an optimal solution, we decided to benchmark the strategy against an upper bound of the optimal that can be efficiently calculated. More specifically, we simplify the provisioning problem in two ways that significantly reduce the complexity of finding an optimal solution. First, we assume that there is no time constraint on the completion of a workflow (i.e., $u(t) = u_{\max}$, regardless of the eventual completion time t). Second, we consider only linear workflows with parallelism 0. These two assumptions allow us to concentrate only on balancing service failure probabilities and their costs, and to disregard the complexities of interleaving parallel tasks. Clearly, an optimal strategy for this simplified problem represents an upper bound for the corresponding optimal strategy in an environment with some deadline and penalty.

Now, given these assumptions, it is straight-forward to determine an optimal strategy and calculate its expected utility. First, we exploit the simple workflow structure and start with the last task in the workflow ($t_{|T|}$). Clearly, the optimal strategy will invoke services that maximise the expected utility of that task, regardless of the services invoked for earlier tasks. We also note that a service s_i will eventually be invoked by the optimal strategy if and only if the task has not been completed yet and the expected utility of invoking s_i is positive (i.e., $(1 - f(s_i)) \cdot u_{\max} - c(s_i) > 0$). Finally, if several services are invoked by the optimal strategy, it will always invoke them in descending order of the ratio $\eta(s_i) = \frac{1-f(s_i)}{c(s_i)}$ (it is easy to show that doing otherwise would result in a lower expected utility). This means that we can quickly determine the optimal strategy for the last workflow task by ordering all services in descending order of $\eta(s_i)$ and discarding those where $(1 - f(s_i)) \cdot u_{\max} - c(s_i) \leq 0$. By constructing a simple decision tree from this, we can calculate the expected utility for that task. Given this utility, we can then repeat the process iteratively for all preceding workflow tasks until we have an overall workflow strategy and an associated expected utility.

This procedure is summarised in Algorithm 5.15, which iterates over the workflow tasks, as described above, in the main loop in lines 4–16. For each task, it computes a task-specific expected utility value, $\bar{u}'$, by considering the contribution of each service that will be invoked, from last to first (as calculated in line 10, this depends on the cost, the probability of success and the utility the agent expects to gain from completing the task, $\bar{u}$). This task-specific utility

Algorithm 5.15 Upper bound on the expected utility of the optimal strategy.

```
1:  procedure COMPUTE-UPPER-BOUND(W)
2:      ū ← u_max                                  ▷ Start with maximum utility
3:      i ← |T|                                    ▷ Begin from last task
4:      while i > 0 do
5:          ū' ← 0                                 ▷ Utility so far for this task
6:          S'_i ← S_i                             ▷ Set of suitable service instances
7:          while |S'_i| > 0 do                    ▷ Consider one service at a time
8:              s ← argmax_{s_j ∈ S'_i} −η(s_i)    ▷ Pick next service to consider
9:              if η(s_i) · ū > 1 then             ▷ If feasible to invoke this...
10:                 ū' ← (1 − f(s)) · ū − c(s) + f(s) · ū'   ▷ ...include its impact on ū'
11:             end if
12:             S'_i ← S'_i \ {s}                  ▷ Remove it from S'_i
13:         end while
14:         ū ← ū'
15:         i ← i − 1                              ▷ Continue with predecessor
16:     end while return ū                         ▷ Return final utility
17: end procedure
```

is then used as $\bar{u}$ for the preceding task and the process is repeated until a final expected utility value is returned in line 16.

Now, using this procedure, we can calculate an upper bound for the average profit of the optimal provisioning strategy (on a sequential workflow). Generally, we observed that this upper bound is very high compared to the performance of any of the other strategies we have tested. This is most likely due to the deadline, which presents a significant constraint and necessitates the more expensive parallel provisioning of services. To evaluate the effect of this constraint in more detail and to compare the performance of our strategy to the optimal as its environment becomes increasingly similar to the simplifying assumptions we made above, we tested our strategy on workflows with varying deadlines.

In more detail, we adopt mostly the same experimental conditions as described in Section 5.4.1, but this time we consider a workflow consisting of 25 sequential tasks with $u_{\max} = 25000$ and carry out 500 repetitions of all experiments. We also draw the average type cost from a new distribution, $\mathcal{U}_c(5, 100)$, in order to place slightly greater emphasis on service costs (rather than the duration). As in the previous section, we set the inter-type variance to 0 to avoid infeasible workflows.

Figure 5.19 shows the results of our experiments as we gradually increase the workflow deadline (the *upper bound* plotted on the graph is the average expected utility obtained by Algorithm 5.15, rather than the result of any experimental run). Here, we see clearly that the deadline has a considerable influence on the performance of the strategy. When $t_{\max} = 125$, the *fast flexible* strategy achieves barely any positive utility. This is because it has to rely on expensive parallel redundancy to complete its workflows within the deadline and this is often infeasible in the environment considered here (in fact, even when $\Phi = 0.0$, the strategy ignores over 90% of its workflows).

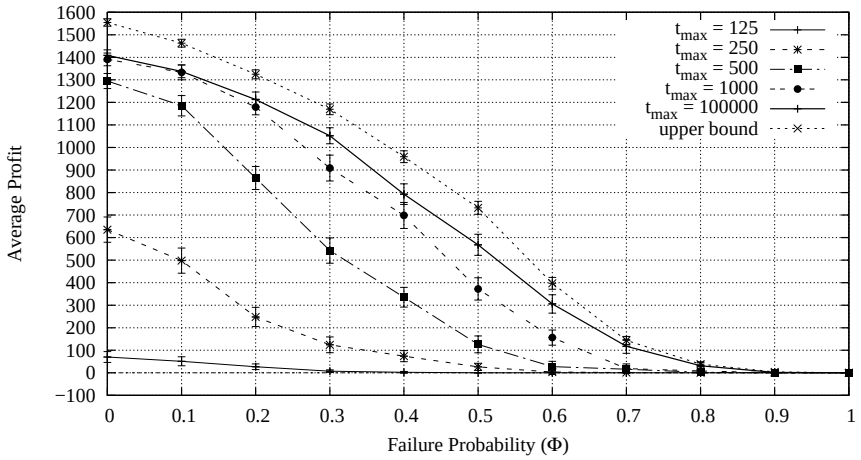


FIGURE 5.19: Performance comparison of the *fast flexible* strategy with upper bound on optimal strategy.

Strategy	Deadline	Average Profit	% of Upper Bound
upper bound	∞	705.63 ± 18.20	100.00
fast flexible	125	14.32 ± 3.21	2.03 ± 0.46
fast flexible	250	146.39 ± 10.84	20.75 ± 1.63
fast flexible	500	399.53 ± 16.37	56.62 ± 2.74
fast flexible	1000	551.47 ± 17.93	78.15 ± 3.24
fast flexible	100000	620.69 ± 16.98	87.96 ± 3.31

TABLE 5.8: Results of the *fast flexible* strategy compared to an upper bound of the optimal.

As we increase the deadline, the *fast flexible* strategy begins to gradually perform better, as it can rely more on the cheaper serial provisioning. In fact, the larger $t_{\max}$, the more similar the performance of the *fast flexible* strategy becomes to the upper bound of the optimal. When $t_{\max} = 100000$, the deadline no longer presents a significant constraint on the consumer and so that environment is most similar to the assumptions we made in calculating the upper bound. Here, the *fast flexible* strategy achieves an overall average profit that is $87.96 \pm 3.31\%$ of the upper bound (the complete results are shown in Table 5.8). This is a promising result that shows the *fast flexible* strategy comes close to the optimal despite relying on a simple local search. We believe the remaining discrepancy in average profit is caused by our early termination of the hill-climbing procedure and by encountering local maxima. Hence, in the following chapter, we will consider a more stochastic search technique that is able to escape such local maxima.

5.5 Summary

In this chapter, we looked at environments where a single workflow task may be satisfied by a large number of highly heterogeneous services. Within this context, we first extended our system model and then proposed a number of modifications to the *flexible* strategy from Chapter 4 to deal with heterogeneous services. These modifications allowed us to address a more complex

problem, but also resulted in a strategy (the *detailed flexible* strategy) that explores a larger decision space than our previous strategy. To speed up the search for a good solution, we then proposed a *fast flexible* strategy that considers a smaller solution space and also terminates its search after a fixed number of iterations.

In empirical experiments, we showed that both our modified flexible approaches outperfom currently prevalent provisioning approaches in most environments, and that they achieve a positive profit even when services fail in 80-90% of cases (at which point all other strategies makes a loss). Thus, in addition to meeting the same requirements as in the previous chapter, the work presented in this chapter additionally meets our research requirement to deal with heterogeneous services (Requirement M.4).

Together, Chapters 4 and 5 present a set of tools that the designer of a service-consuming agent can use in environments where services are invoked on demand. The work in the former chapter is particularly suitable for cases where there is either little difference in the services that satisfy a given task, or when there is considerable uncertainty about this difference. In such systems, our approach determines *how many* services to provision and *when* to re-provision services to deal with failures, using fast calculations. The work presented in the latter chapter expands on this and also answers the question of *which* services to provision when there are many competing candidates offering different levels of quality.

However, so far we have considered only a simple market mechanism, where services are always available on demand and at fixed prices. In the following chapter, we will extend this and look into systems where service availability changes over time (Requirement M.5), where prices are uncertain (Requirement M.2.b), and where the consumer may reach advance agreements with providers, possibly in return for discounted or more reliable services (Requirement M.3.b).

Chapter 6

Service Provisioning with Advance Agreements

So far, we have looked at service-oriented systems where providers offer services without the need for explicit contract negotiations, and where the population of available services is static throughout the execution of a workflow. As discussed in Chapter 2, this applies to many current service-based systems, in which services are advertised on a registry and accessed by consumers on demand. However, there is increasing interest in building systems where the provision of services is negotiated and an explicit service-level agreement, or contract, is agreed upon in advance (as motivated in Section 2.1.5).

To this end, in this chapter, we address such systems by first extending our system model to include service negotiations using a market-based mechanism (Section 6.1). This is followed, in Section 6.2, by a discussion of a novel provisioning strategy. In contrast to the approaches presented in previous chapters, this strategy does not initially provision specific services for all workflow tasks, but rather takes high-level decisions about how and when to provision workflow tasks during execution. These are then constantly adapted and refined during execution as the agent interacts with the dynamic market. Finally, we evaluate this strategy in Section 6.3. Hence, we deal with our last outstanding model requirements to address systems where prices are not fixed (Requirement M.2.b), where advance agreements are entered into with providers (Requirement M.3.b) and where service populations are dynamic (Requirement M.5). In so doing, we also show how the agent can adapt its decisions throughout execution as new information becomes available (Requirement A.4).

6.1 Model Extension

To address more dynamic systems with flexible pricing and advance agreements, we substantially modify our model in this section. Most importantly, we now include a negotiation process,

Term	Description
$s(o) : \mathcal{T}$	The *service type* offered (equal to the requested type).
$t(o) : \mathbb{N}$	The *starting time* at which the service can be invoked (equal to the requested time).
$c_r(o) : \mathbb{R}$	The *reservation cost*, which must be paid immediately by the consumer when entering the contract.
$c_e(o) : \mathbb{R}$	The *execution cost*, which is the remaining cost (after the reservation cost) that the consumer must pay when invoking the service.
$d(o) : \mathbb{Z}^+$	The *duration*, i.e., the number of time steps it will take for the service to complete.
$\delta_f(o) : \mathbb{R}$	The *failure penalty*, which is paid to the consumer when the service fails to complete successfully within the agreed duration.

TABLE 6.1: Service contract terms.

whereby the consumer and provider agree on the terms of a provided service before it is invoked. To this end, we use the contract-net protocol, as it is simple and has been widely used in distributed multi-agent systems (see Section 2.2.2).

Hence, rather than having access to static performance information about service instances, the consumer interacts with a service market to discover the current availability and quality of services. To this end, at a given time step, it may send a *call for proposals*, $\varphi : \mathcal{T} \times \mathbb{N}$, to the service market to request a particular type of service at a certain time step. For example, $\varphi = (T_1, 2)$ indicates that the consumer requires a service of type T_1 to start at time step 2.

In response to each call, the market returns a set of *offers*, $C_\varphi \subseteq \mathbb{C}$. These are potential contracts that the service providers participating in the system are willing to offer to the consumer ($\mathbb{C}$ is the set of all offers). Each offer $o \in C_\varphi$ contains a number of terms, as given in Table 6.1. Although based on the performance characteristics introduced in Section 3.3, there are some differences to our previous model. Specifically, a service instance is now offered at a specific time step only ($t(o)$) and we use a more expressive cost model, which splits the investment of the consumer into two parts — an initial reservation cost ($c_r(o)$) and an execution cost ($c_e(o)$). This cost model is more realistic in the contracting scenario we consider in this chapter, because it requires the consumer to pay the provider for its commitment to execute the service at a later time, but it does not necessarily require the full cost of the service if the consumer later changes its mind[1]. Furthermore, we also extend our model to include a penalty, $\delta_f(o) : \mathbb{R}$, that is payable by the provider upon service failure and that constitutes a compensation to the consumer (or simply a refund of the service costs if $\delta_f(o) = c_r(o) + c_e(o)$).

This process of requesting services and receiving responses may be repeated arbitrarily often during a given time step for different time steps or service types (we assume that the offers returned for two requests with the same service types and times are always identical during a particular time step). Furthermore, we assume that the consumer has some information about the probabilities of the possible outcomes of each offer, as shown in Table 6.2 (in practice, these

[1] As outlined in Section 2.2.2, such leveled payments are common in related work (Sandholm and Lesser (1996); Collins et al. (2001)).

Probability	**Description**
$P_s(o) : [0, 1]$	The *success probability* is the probability that the service will be completed successfully within the agreed duration.
$P_f(o) : [0, 1]$	The *failure probability* is the probability that the service will not be completed successfully within the agreed duration and that the provider will pay the failure penalty $\delta_f(o)$.
$P_d(o) : [0, 1]$	The *defection probability* is the probability that the service will not be completed successfully and that the provider will also fail to pay the agreed penalty $\delta_f(o)$ (e.g., if the provider leaves the market, maliciously disregards the market rules or if the service simply crashes).

TABLE 6.2: Performance information (outcome probabilities).

may be obtained through a trust and reputation mechanism, or through previous interactions). Together, these probabilities describe all possible, mutually exclusive outcomes of an offer, such that $P_s(o) + P_f(o) + P_d(o) = 1$. As in previous chapters, we assume that the outcomes of any two distinct offers are independent.

During the same time step as receiving offers from the market, the consumer may provision[2] any number (or none) of these offers for the tasks of its workflow. To do this, it sends a single acknowledgement to the market, $a : C_{\hat{t}} \rightarrow T$, that maps offers to the corresponding tasks of the workflow, where $C_{\hat{t}}$ is the set of all offers received during the time step. At this point, the consumer must pay the reservation costs of all provisioned offers, and any offers not in the domain of a are implicitly assumed to be rejected. We also assume that the consumer may provision several offers for a single task (as we did previously).

At the end of each time step, the consumer may invoke its provisioned offers (including those provisioned during previous time steps), provided that all relevant precedence constraints given by E have been satisfied and that the agreed starting time matches the current time. The outcome of the invocation is one of the outcomes listed in Table 6.2, but we assume that it is not known until the beginning of the time step at which the service is scheduled to end (e.g., if invoking offer o with $t(o) = 15$ and $d(o) = 10$, the consumer will only be notified of the outcome at the beginning of time step $t = 25$).

The extended system model described in this section meets the remaining model requirements outlined in Section 1.4.1. In particular, we now consider flexible service pricing (Requirement M.2.b), as the cost of a service is not fixed or publicly known, but rather determined through a negotiation process. Furthermore, services are provisioned explicitly in advance (Requirement M.3.b) and the number of offers and their characteristics may vary dynamically (Requirement M.5). Given this extended model, we continue in the following section by describing a flexible provisioning strategy that provisions services in advance.

[2]To avoid any confusion, it is important here to note that our use of the word "provisioning" is more specific in this chapter than in the remainder of the thesis. While we used it earlier to denote any implicit decision by the agent to invoke a particular service, we now use it only when the agent has decided to accept, and pay for, a particular service offer.

6.2 Flexible Provisioning

In designing a flexible provisioning strategy in environments with advance agreements, we follow the same basic approach as discussed in the previous chapters. However, we have had to make a number of significant changes to account for the more complex provisioning scenario. We briefly summarise these here before outlining the details of the strategy in the following sections.

First, the flexible pricing and more dynamic environment means that the consumer no longer has complete information about the exact availability and performance characteristics of services before requesting offers and provisioning them. This could be addressed by provisioning the entire workflow in advance (i.e., requesting and provisioning offers for every task in the workflow at time step $\hat{t} = 0$). However, doing so is not practical for large workflows or when services have a high probability of failure, as some tasks may not be completed as planned, thereby resulting in missed starting times of later tasks in the workflow.

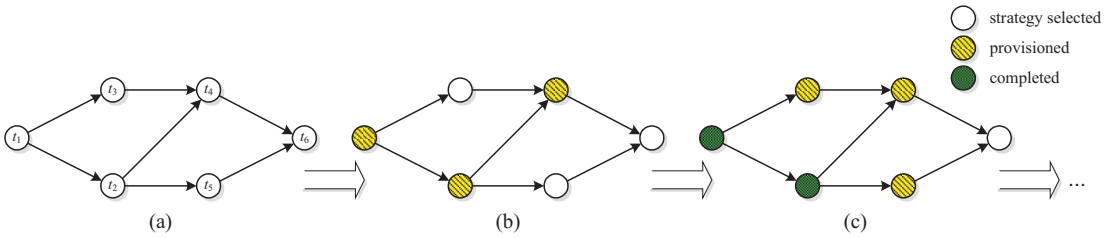


FIGURE 6.1: Progressive provisioning of a workflow over time.

For this reason, we decided to design a strategy that provisions workflows progressively throughout execution, as outlined by Figure 6.1. In more detail, the consumer agent first makes simple high-level decisions about how and when to provision each task in the workflow, but without requesting or committing to any service offers yet (Figure 6.1 (a)). These include decisions about how long in advance they should be provisioned, how to select among competing offers and how much time to leave between successive tasks (considering that some services may fail and thus jeopardise the successful execution of later offers). Using statistical information based on past observations of the market, these high-level decisions allow the agent to estimate the various task parameters used in Chapters 4 and 5, and thus obtain an overall estimated utility.

Based on these initial decisions, the consumer then gradually requests and selects concrete offers for the workflow, as shown in Figure 6.1 (b). Here, it has provisioned some tasks in advance, e.g., because they tend to be of better quality when a longer advance notice is given to the providers, or because they are scarce services that are difficult to procure at short notice. However, it has also left some tasks unprovisioned, e.g., because they are plentiful and can easily be provisioned exactly when required, or because the completion time of the preceding tasks is too uncertain. Then, as tasks are completed successfully, further tasks are provisioned as required (Figure 6.1 (c)).

The second significant difference to our previous work is that we now perform the provisioning in a more adaptive manner, thereby addressing Requirement A.4. That is, rather than following the initial decisions blindly throughout workflow execution, the consumer now adapts its decisions as new information becomes available. We make this change, because the initial high-level decisions use statistical performance information that may turn out to be inaccurate when the actual offers become known. Similarly, making the strategy more adaptive allows it to react appropriately to unexpected outcomes of tasks. For example, when an initially reliable offer turns out to be unsuccessful, the consumer may need to adapt its provisioning strategies for later tasks of the workflow, in order to meet the overall deadline.

Before we start to discuss the strategy in detail, we briefly summarise it below as an optimisation problem.

6.2.1 Problem Formulation

As in previous chapters, we are interested in building a rational agent that acts to maximise its expected utility. Hence, we want our agent to adopt a *provisioning strategy* Ψ that maximises the difference between the reward and cost of following it. Here, we use *provisioning strategy* to denote a set of decisions for each workflow task about how the agent intends to complete it. This may either be a high-level decision about how to provision it in the future, or a concrete set of offers that have already been provisioned for it. In both cases, the agent may also associate further decisions for contingencies with a task, but we will discuss and formalise this later. In this context, we define the overall agent strategy we develop in this chapter as follows:

Definition 15 (Dynamic Flexible Strategy)**.** A consumer following a ***dynamic flexible*** strategy makes appropriate decisions to provision services for its workflow. To this end, the agent finds a suitable provisioning strategy Ψ, so that the agent's predicted profit is maximised. Furthermore, the agent continuously incorporates new service outcomes into its predictions and adapts its provisioning strategy accordingly.

Following the notation of Chapters 4 and 5, we formulate this as an optimisation problem:

$$\max_{\Psi} \ (\bar{u}_t(\Psi) - \bar{c}(\Psi)) \tag{6.1}$$

where $\bar{u}_t(\Psi)$ is the expected reward of following the provisioning strategy Ψ and $\bar{c}(\Psi)$ is the associated expected cost.

In the following, we discuss our *dynamic flexible* strategy in more detail. We start by showing how the basic task parameters used in previous chapters can be calculated from a given set of offers (Section 6.2.2). Then we discuss how we use high-level task strategies to estimate the outcomes of tasks before provisioning, and how these strategies can be combined into simple contingency plans for each task (Section 6.2.3). In Section 6.2.4, we show how the consumer decides when to begin provisioning each task and in Section 6.2.5, we briefly discuss how the

overall utility of a complete workflow is estimated. Similar to the work in previous chapters, this is used in Section 6.2.6 as a basis for a local search alrogithm. Finally, we discuss how our agorithm adaptively improves its decisions at run-time as more information about available offers and invocation outcomes become known (Section 6.2.7), and we summarise the strategy based on our generic agent algorithm (Section 6.2.8).

6.2.2 Task Provisioning

In this section, we outline some basic calculations to predict the outcome of provisioning a certain set of offers for a workflow task. These calculations are central to the remainder of this discussion, as we use them both when the consumer has decided what offers to provision for the task, and also to derive the performance characteristics of high-level task strategies.

In this context, we refer to a chosen set of offers for a particular task as a *provisioning decision*:

Definition 16 (Provisioning Decision)**.** A provisioning decision $\gamma_i \subseteq \mathbb{C}$ is a set of offers (of the correct type) that the consumer has provisioned for a task t_i.

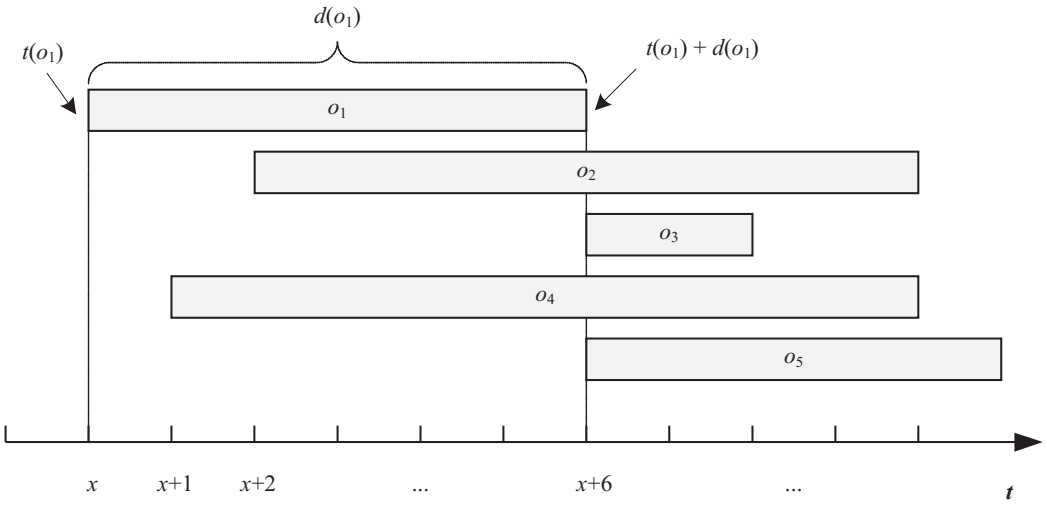


FIGURE 6.2: An example provisioning decision for a single task.

As an example, Figure 6.2 shows the provisioning decision $\gamma_i = \{o_1, o_2, o_3, o_4, o_5\}$, where five distinct offers have been provisioned for a single task.

We also assume that we have a cumulative probability density function, $\mathcal{E}_i(t)$, that describes the probability that any predecessors of t_i will have completed successfully by time t. Here, we assume that $\mathcal{E}_i(t)$ is conditional on the successful completion of all predecessors.

Now, first we are interested in calculating the expected cost of a provisioning decision. This is simply the sum of all offer reservation costs and expected execution costs[3]:

$$\bar{c}_t = \bar{c}_r + \bar{c}_e \tag{6.2}$$

where $\bar{c}_r$ is the sum of all reservation costs:

$$\bar{c}_r = \sum_{o \in \gamma(t_i)} c_r(o) \tag{6.3}$$

and $\bar{c}_e$ is the overall expected execution cost for all offers. To calculate this, we define a number of auxiliary terms:

- $\hat{s} : \mathbb{Z} \to \mathbb{Z}$ is a sequence of all unique start times of the offers in γ_i in ascending order (i.e., $\hat{s}(1)$ is the earliest unique starting time of any offer in γ_i, $\hat{s}(2)$ the second earliest, and so on). For the offers in Figure 6.2, $\hat{s} = \{(1, x), (2, x+1), (3, x+2), (4, x+6)\}$.

- $\hat{p}_s : \mathbb{Z} \to \mathbb{R}$ is a sequence of real numbers, each of which represents the probability that the corresponding element in $\hat{s}$ is the first time step at which offers in γ_i can be invoked (depending on the completion time of the task's predecessors, as given by $\mathcal{E}_i$). More formally, for all $n \in \mathbb{N}$, such that $1 \leq n \leq |\hat{s}|$:

$$\hat{p}_s(n) = \begin{cases} \mathcal{E}_i(\hat{s}(n)) & \text{if } n = 1 \\ \mathcal{E}_i(\hat{s}(n)) - \mathcal{E}_i(\hat{s}(n-1)) & \text{otherwise} \end{cases} \tag{6.4}$$

- $o_{\text{pre}} : (\mathbb{Z} \times \mathbb{Z}) \to \mathcal{P}(\mathbb{C})$ is a function that maps two time slots, $\hat{t}_1$ and $\hat{t}_2$, to the set of offers that start on or after $\hat{t}_1$ and end on or before $\hat{t}_2$ (i.e., $o_{\text{pre}}(\hat{t}_1, \hat{t}_2) = \{c \in \gamma_i \mid t(o) \geq \hat{t}_1 \wedge t(o) + d(o) \leq \hat{t}_2\}$). For example, in Figure 6.2, $o_{\text{pre}}(x+1, x+10) = \{o_2, o_3, o_4\}$.

- $o_{\text{after}} : \mathbb{Z} \to \mathcal{P}(\mathbb{C})$ is a function that maps a time slot, $\hat{t}$, to the set of offers that start on or after $\hat{t}$ (i.e., $o_{\text{after}}(\hat{t}) = \{c \in \gamma_i \mid t(o) \geq \hat{t}\}$).

- $c_{\text{inv}} : \mathbb{C} \to \mathbb{R}$ maps an offer to the expected cost of invoking it:

$$c_{\text{inv}}(o) = c_e(o) - P_f(o)\delta_f(o) \tag{6.5}$$

- $p_{\text{inv}} : (\mathbb{C} \times \mathbb{Z}) \to \mathbb{R}$ maps an offer and a time step, $\hat{t}$, to the probability that the offer will eventually be invoked, given that execution of the task starts at time step $\hat{t}$:

$$p_{\text{inv}}(o, \hat{t}) = \prod_{o' \in o_{\text{pre}}(\hat{t}, t(o))} \big(1 - P_s(o')\big) \tag{6.6}$$

[3]Here, and in the following, we assume that it is never rational for the consumer to invoke a service that is no longer needed (i.e., that the expected execution cost is never negative). This assumption keeps the calculations more concise, but can be easily relaxed.

Given these, we calculate the overall expected execution cost, $\bar{c}_e$, by considering each possible starting time for the task (given by $\hat{s}$), and then the respective offers that may be executed given the starting time:

$$\bar{c}_e = \sum_{a=1}^{|\hat{s}|} \left(\hat{p}_s(a) \cdot \sum_{o \in o_{\text{after}}(\hat{s}(a))} (p_{\text{inv}}(o, \hat{s}(a)) \cdot c_{\text{inv}}(o)) \right) \tag{6.7}$$

Next, we consider three different overall outcomes of following a provisioning decision γ_i for task t_i:

- **Late:** The predecessors of t_i complete late, such that no offer in γ_i is ever executed.
- **Failed:** At least one offer in γ_i is executed, but none succeeds.
- **Successful:** At least one offer in γ_i is executed and successfully completes the task.

For each of these, we calculate a probability that this outcome occurs (p_l, p_f, p_s, respectively), an expected end time when the outcome is known (d_l, d_f, d_s), and the variance of this time (v_l, v_f, v_s). We consider these outcomes separately, as we will later construct contingency plans that the agent may take when a task has been unsuccessful (see Section 6.2.3.2).

Now, treating each of the parameters separately, we can calculate the probability that the predecessors complete *late* as follows:

$$p_l = 1 - \mathcal{E}_i(\hat{s}(|\hat{s}|)) \tag{6.8}$$

The associated expected end time is simply the highest starting time of any offer, and the variance of this is 0:

$$d_l = \hat{s}(|\hat{s}|) \tag{6.9}$$

$$v_l = 0 \tag{6.10}$$

When the provisioning decision has *failed*, we again examine each possible task starting time separately (as in Equation 6.7). Hence, the probability of this event is:

$$p_f = \sum_{a=1}^{|\hat{s}|} \left(\hat{p}_s(a) \cdot \prod_{o \in o_{\text{after}}(\hat{s}(a))} (1 - P_s(o)) \right) \tag{6.11}$$

The expected end time of this outcome now depends on the latest end time of a group of failed offers, again evaluated for different starting times:

$$d_f = \frac{1}{p_f} \cdot \sum_{a=1}^{|\hat{s}|} \left(\hat{p}_s(a) \cdot t_{\text{end}}(o_{\text{after}}(\hat{s}(a))) \cdot \prod_{o \in o_{\text{after}}(\hat{s}(a))} (1 - P_s(o)) \right) \tag{6.12}$$

where $t_{\text{end}} : \mathcal{P}(\mathbb{C}) \rightarrow \mathbb{R}$ is a function that maps a set of offers to the highest end time within that set:

$$t_{\text{end}}(C) = \max_{o \in C}(t(o) + d(o)) \tag{6.13}$$

The variance of this can be calculated in a similar manner, using the expected squared end time:

$$v_f = -d_f^2 + \frac{1}{p_f} \cdot \sum_{a=1}^{|\hat{s}|} \left(\hat{p}_s(a) \cdot t_{\text{end}}(o_{\text{after}}(\hat{s}(a)))^2 \cdot \prod_{o \in o_{\text{after}}(\hat{s}(a))} (1 - P_s(o)) \right) \tag{6.14}$$

Finally, the probability that the decision will result in a *successful* execution of the task is then:

$$p_s = 1 - p_f - p_l \tag{6.15}$$

The expected end time in this case depends on the end time of the first successfully executed offer. To calculate this, we use an auxiliary function, $c_{\text{end}} : \mathbb{Z} \rightarrow \mathcal{P}(\mathcal{P}(\mathbb{C}) \times \mathbb{Z})$, that maps a time step, $\hat{t}$, to a set of tuples, each of which consists of a set of offers that start on or after $\hat{t}$ and that end on a common time step, as well as the respective end time. Formally:

$$\begin{aligned} c_{\text{end}}(\hat{t}) = \{(C, e) \quad | \quad & C \neq \emptyset \wedge e \in \mathbb{Z} \wedge C \subseteq o_{\text{after}}(\hat{t}) \\ & \wedge \forall c \in o_{\text{after}}(\hat{t}) \cdot c \in C \Leftrightarrow t(o) + d(o) = e\} \end{aligned} \tag{6.16}$$

To give an example, for the provisioning decision in Figure 6.2, $c_{\text{end}}(x+1) = \{(\{o_3\}, x + 8), (\{o_2, o_4\}, x + 10), (o_5, x + 11)\}$.

Then we use this to calculate the probability of each possible end time and thus the expected end time:

$$\begin{aligned} d_s = \frac{1}{p_s} \cdot \sum_{a=1}^{|\hat{s}|} \quad & \left(\hat{p}_s(a) \cdot \sum_{(C,e) \in c_{\text{end}}(\hat{s}(a))} \left(e \cdot \left(1 - \prod_{o \in C} (1 - P_s(o)) \right) \cdot \right. \right. \\ & \left. \left. \prod_{o' \in o_{\text{pre}}(\hat{s}(a), e-1)} (1 - P_s(o')) \right) \right) \end{aligned} \tag{6.17}$$

The variance is calculated in a similar manner as in Equation 6.14:

$$\begin{aligned} v_s = -d_s^2 + \frac{1}{p_s} \cdot \sum_{a=1}^{|\hat{s}|} \quad & \left(\hat{p}_s(a) \cdot \sum_{(C,e) \in c_{\text{end}}(\hat{s}(a))} \left(e^2 \cdot \left(1 - \prod_{o \in C} (1 - P_s(o)) \right) \cdot \right. \right. \\ & \left. \left. \prod_{o' \in o_{\text{pre}}(\hat{s}(a), e-1)} (1 - P_s(o')) \right) \right) \end{aligned} \tag{6.18}$$

The calculations outlined above now allow us to determine similar performance parameters as we have used in previous chapters, including the success probability of a provisioned task, its expected cost, end time and variance of the end time. However, as we discussed at the beginning

of Section 6.2 and as shown in Figure 6.1, the consumer agent will typically not provision an entire workflow at once, and therefore it will not know the exact offers available for each task until later during execution. For this reason, we rely on averages for these figures, which have been learnt over time by observing offers on the market. We describe this information in more detail in the next section.

6.2.3 High-Level Task Strategies

In order to make predictions about unprovisioned tasks, our flexible provisioning approach first selects simple high-level task strategies for each task. These are decision rules that the agent will later use to submit a call for proposals for the task and to select from the returned offers, and that have some associated statistical information about their performance. For example, such a high-level strategy might be to submit a call for proposals some time before the task actually becomes executable, and then to provision the most reliable offer available, or it might be to provision the five cheapest offers at the last moment, when the task is already executable. Depending on the market conditions, such strategies may have very different performance characteristics — the former might result in a cheaper and more reliable execution of the task than the latter, but also carry a risk that the arranged starting time cannot be met if the preceding tasks finish later than expected.

In this section, we formalise these high-level strategies, outline how the agent learns statistical information about them by observing the market and describe how to construct simple contingency plans to deal with failures.

6.2.3.1 Strategy Library

High-level provisioning strategies are available to the consumer as a library of strategies, $l : \mathcal{T} \rightarrow \mathcal{P}(\Omega)$, that maps each service type to a set of strategies (Ω is the set of all strategies). Each strategy $\omega \in \Omega$ is described by a number of parameters, as shown in Table 6.3. The first two of these prescribe how the consumer will formulate its call for proposals, e.g., if $t_a(\omega) = 100$ and $t_w(\omega) = 3$, it will request services 100 time steps in advance and for three consecutive time steps. The latter two describe how it will select from the returned offers. Here, we consider four simple selection strategies for parameter $\vartheta(\omega)$: {`cost`, `unreliability`, `end_time`, `balanced`}. The first three indicate that the consumer will always choose the offers with, respectively, the lowest expected cost ($c_r(o)+c_e(o)-P_f(o)\delta_f(o)$), the lowest probability of not succeeding ($1 - P_s(o)$) or the lowest end time ($t(o) + d(o)$). The selection strategy `balanced` will pick the offers that minimise a sum of these parameters, each normalised to the interval $[0, 1]$, so that 0 corresponds to the offer with the lowest parameter and 1 to the highest. We also assume that there is a high-level strategy not to do anything, ω_{null} (i.e., the agent will stop executing the task).

Parameter	Description
$t_a(\omega) : \mathbb{N}$	Number of time steps to provision offers in advance.
$t_w(\omega) : \mathbb{Z}^+$	Time interval to request services for.
$n(\omega) : \mathbb{Z}^+$	Maximum number of offers to provision.
$\vartheta(\omega)$	Strategy for choosing offers to provision when more than $n(\omega)$ offers are available.

TABLE 6.3: Task strategy parameters.

Furthermore, we assume that the consumer has some performance information about each of the strategies, which it previously learnt by observing the response of the market to various calls for proposals. Specifically, we assume that the consumer has repeatedly submitted calls of proposals corresponding to its known strategies to the market, calculated the probabilities and expected values described in Section 6.2.2 and built up statistical averages for these, without necessarily provisioning or invoking any services. In doing so, we assume that tasks are invoked in isolation, i.e., that there are no predecessors and so $\forall \hat{t} \cdot \mathcal{E}_i(\hat{t}) = 1$. These statistics are summarised in Table 6.4. Here, ϵ denotes the overall outcome of the strategy, with $\epsilon \in \{\texttt{success}, \texttt{unavailable}, \texttt{failed}\}$ (which refers to the same outcomes as described in Section 6.2.2 with the addition of `unavailable`, which we introduce to denote the case where no offers were found). We also do not include the *late* outcome here, because we examine tasks in isolation.

In more detail, these statistics are derived directly from those discussed in Section 6.2.2. The first three, $\check{c}_r(\omega)$, $\check{c}_e(\omega)$ and $\check{c}(\omega)$ are based on Equations 6.3 and 6.7. The next, $\check{p}(\omega, \epsilon)$, is based on Equations 6.11 and 6.15, as well as on the frequency with which the consumer fails to find any offers. Finally, the duration, squared duration and derived variance are obtained using similar calculations as in Equations 6.12, 6.14, 6.17 and 6.18 (with small modifications to calculate the duration from the first time step the original request was submitted for, and to record only the squared duration rather than the variance).

Statistic	Description
$\check{c}_r(\omega) : \mathbb{R}$	Average of the reservation cost.
$\check{c}_e(\omega) : \mathbb{R}$	Average of the expected execution cost.
$\check{c}(\omega) : \mathbb{R}$	Overall expected cost ($\bar{c}_r(\omega) + \bar{c}_e(\omega)$).
$\check{p}(\omega, \epsilon) : [0, 1]$	Average of the probability of outcome ϵ.
$\check{d}(\omega, \epsilon) : \mathbb{R}$	Average of the expected time until outcome ϵ is known (measured from first time step that call for proposals was submitted for).
$\check{d^2}(\omega, \epsilon) : \mathbb{R}$	Average of the expected squared time until ϵ is known.
$\tilde{v}(\omega, \epsilon) : \mathbb{R}$	Variance of time ($\tilde{v}(\omega, \epsilon) = \check{d^2}(\omega, \epsilon) - \check{d}(\omega, \epsilon)^2$).

TABLE 6.4: Average performance statistics when following strategy ω ($\epsilon \in \{\texttt{success}, \texttt{unavailable}, \texttt{failed}\}$).

These strategies now allow the consumer agent to make some predictions about the likely outcomes, the cost and duration for completing a task, given that it adopts a certain strategy (see

Figure 6.5 in Section 6.2.9 for some example strategies and the performance statistics). However, assigning a single strategy to each task is unlikely to be sufficient in uncertain environments as the consumer needs some capabilities to plan for contingencies and predict their impact on the cost and feasibility of the workflow. Hence, we decided to include several contingent strategies that the consumer will use if its primary strategy was not successful. We describe these in the following section.

6.2.3.2 Planning for Contingencies

The contingent strategies we consider are shown in Figure 6.3. Here, s_p is the main strategy the consumer will use to provision the task, but it also has a number of strategies to fall back on if the initial offers were not successful:

- s_l is used to re-provision offers when the preceding tasks in the workflow have not been completed by the time the initial offers are available for invocation. In this case, the consumer will wait until the preceding tasks have completed and then provision new offers using s_l.
- s_u is used when either all initial offers were cancelled, or when the initial strategy did not result in any provisioned offers at all. In the latter case, the agent waits until all preceding tasks have been completed and then adopts s_u.
- s_f is adopted when the initial offers were started, but did not complete successfully. It is carried out as soon as the last offer completes unsuccessfully.

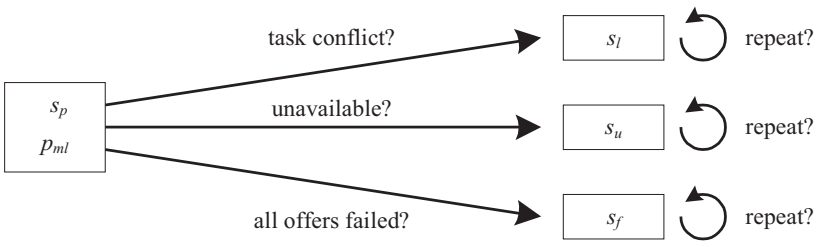


FIGURE 6.3: Task contingencies.

To further extend the number of strategies we consider, we note that the consumer might continue to repeat certain strategies until a task is completed (e.g., when the consumer does not have a tight deadline, it may decide to select the cheapest offer on the market, attempt it, and, in case of failure, simply try another cheap offer until the task is eventually completed). Hence, we extend the space of possible strategies for s_l, s_u and s_f by adding a repeated strategy, ω_r for each $\omega \in \Omega$. Generally, these repeated strategies will be carried out as soon as the previous attempt is known to have failed, except when no suitable offers were found at all — in this case, the agent waits a time step before attempting the strategy again.

Now, for the repeated strategies, we derive their performance statistics from their non-repeated counterparts as follows[4]. First, we assume that, given an infinite number of attempts, the task will eventually be successful (provided $\check{p}_s(\omega) > 0$):

$$\check{p}_s(\omega_r) = \begin{cases} 1 & \text{if } \check{p}_s(\omega) > 0 \\ 0 & \text{if } \check{p}_s(\omega) = 0 \end{cases} \tag{6.19}$$

Next, the expected cost will rise if there is a chance of failure:

$$\check{c}(\omega_r) = (\check{c}_r(\omega) + \check{c}_e(\omega)) \cdot \sum_{n=0}^{\infty} (1 - \check{p}_s(\omega))^n = \frac{\check{c}_r(\omega) + \check{c}_e(\omega)}{\check{p}_s(\omega)} \tag{6.20}$$

The duration will also rise. To calculate this, we use $\check{d}_{\bar{s}}(\omega)$, $\check{d}^2_{\bar{s}}(\omega)$ and $\tilde{v}_{\bar{s}}(\omega)$ to denote the expected value, expected squared value and variance of the duration when following strategy ω, given that it either fails or is not started:

$$\check{d}_{\bar{s}}(\omega) = \frac{\check{p}_f(\omega)\check{d}_f(\omega) + \check{p}_u(\omega)}{\check{p}_f(\omega) + \check{p}_u(\omega)} \tag{6.21}$$

$$\check{d}^2_{\bar{s}}(\omega) = \frac{\check{p}_f(\omega)\left(\check{d}^2_f(\omega) + \tilde{v}_f(\omega)\right) + \check{p}_u(\omega)}{\check{p}_f(\omega) + \check{p}_u(\omega)} \tag{6.22}$$

$$\tilde{v}_{\bar{s}}(\omega) = \check{d}^2_{\bar{s}}(\omega) - \check{d}_{\bar{s}}(\omega)^2 \tag{6.23}$$

Now, we let $\hat{d}_{\bar{s}}(\omega) = \check{d}_{\bar{s}}(\omega) + t_a(\omega)$, which accounts for the extra time that is needed to re-provision, and we calculate the repeated duration, its expected square and variance as follows:

$$\begin{aligned} \check{d}_s(\omega_r) &= \check{p}_s(\omega) \cdot \sum_{n=0}^{\infty} (1 - \check{p}_s(\omega))^n \left(\check{d}_s(\omega) + n \cdot \hat{d}_{\bar{s}}(\omega)\right) \\ &= \check{d}_s(\omega) + \hat{d}_{\bar{s}}(\omega)\frac{1 - \check{p}_s(\omega)}{\check{p}_s(\omega)} \end{aligned} \tag{6.24}$$

$$\begin{aligned} \check{d}^2_s(\omega_r) &= \check{p}_s(\omega) \sum_{n=0}^{\infty} \Bigg((1 - \check{p}_s(\omega))^n \cdot \\ &\qquad \left(\left(\check{d}_s(\omega) + n\hat{d}_{\bar{s}}(\omega)\right)^2 + \tilde{v}_s(\omega) + n\tilde{v}_{\bar{s}}(\omega)\right) \Bigg) \\ &= \check{d}_s(\omega)^2 + \tilde{v}_s(\omega) + \left(2\hat{d}_{\bar{s}}(\omega)\check{d}_s(\omega) + \tilde{v}_{\bar{s}}(\omega)\right) \cdot \\ &\qquad \frac{1 - \check{p}_s(\omega)}{\check{p}_s(\omega)} + \hat{d}_{\bar{s}}(\omega)^2 \left(\frac{2 - 3\check{p}_s(\omega)}{\check{p}_s(\omega)^2} + 1\right) \end{aligned} \tag{6.25}$$

$$\tilde{v}_{\bar{s}}(\omega_r) = \check{d}^2_{\bar{s}}(\omega_r) - \check{d}_{\bar{s}}(\omega_r)^2 \tag{6.26}$$

Finally, the task strategy is annotated with a *maximum late probability*, $p_{ml} : [0, 1)$. This is the largest acceptable probability that the task will not be executable when the offers provisioned by

[4]For conciseness, we use subscripts s, u and f to refer to various outcomes. For example, $\check{p}_s(\omega_r) = \check{p}(\omega_r, \text{success})$.

s_p can be invoked (i.e., that some of its predecessors will not be completed yet). Later we will also use this parameter to decide exactly when to provision each task (see Section 6.2.4), but for now it allows us to calculate some probabilities and expected values related to the task.

Specifically, the overall success probability can be obtained by simply considering all branches of Figure 6.3 that result in success:

$$p_i = p_{ml}\check{p}_s(s_l) + (1 - p_{ml})\left(\check{p}_s(s_p) + \check{p}_n(s_p)\check{p}_s(s_u) + \check{p}_f(s_p)\check{p}_s(s_f)\right) \tag{6.27}$$

The expected reservation cost is the average reservation cost of the primary strategy:

$$c_{ri} = \check{c}_r(s_p) \tag{6.28}$$

The expected execution cost is again calculated by considering the probabilities of all contingencies:

$$c_{ei} = p_{ml}\check{c}(s_l) + (1 - p_{ml})\left(\check{c}_e(s_p) + \check{p}_n(s_p)\check{c}(s_u) + \check{p}_f(s_p)\check{c}(s_f)\right) \tag{6.29}$$

We use similar calculations for the expected time (denoted $\bar{t}_i$) and its expected square (denoted $\bar{t}_{s,i}$), both conditional on overall success as we are not interested in the durations of tasks that have not been completed:

$$\begin{aligned}\bar{t}_i = p_i^{-1}\Big(&p_{ml}\check{p}_s(s_l)\left(t_a(s_l) + \check{d}_s(s_l)\right) + \\ &(1 - p_{ml})\left(\check{p}_s(s_p)\check{d}_s(s_p) + \right. \\ &\check{p}_n(s_p)\check{p}_s(s_u)\left(\check{d}_n(s_p) + t_a(s_u) + \check{d}_s(s_u)\right) + \\ &\left.\check{p}_f(s_p)\check{p}_s(s_f)\left(\check{d}_f(s_p) + t_a(s_f) + \check{d}_s(s_f)\right)\right)\Big)\end{aligned} \tag{6.30}$$

$$\begin{aligned}\bar{t}_{s,i} = p_i^{-1}\Big(&p_{ml}\check{p}_s(s_l)\left(\tilde{v}_s(s_l) + \left(t_a(s_l) + \check{d}_s(s_l)\right)^2\right) + \\ &(1 - p_{ml})\left(\check{p}_s(s_p)\check{d}_s^2(s) + \check{p}_n(s_p)\check{p}_s(s_u)\cdot\right. \\ &\left(\tilde{v}_n(s_p) + \tilde{v}_s(s_u) + \left(\check{d}_n(s_p) + \check{d}_s(s_u) + t_a(s_u)\right)^2\right) + \\ &\check{p}_f(s_p)\check{p}_s(s_f)\left(\tilde{v}_f(s_p) + \tilde{v}_s(s_f) + \right. \\ &\left.\left.\left.\left(\check{d}_f(s_p) + \check{d}_s(s_f) + t_a(s_f)\right)^2\right)\right)\right)\end{aligned} \tag{6.31}$$

The five parameters described above — the success probability of a task, p_i, the expected reservation cost, c_{ri}, the expected execution cost, c_{ei}, the expected duration, $\bar{t}_i$, and the expected squared duration, $\bar{t}_{s,i}$ — as well as the variance, v_i, which can be calculated as in Equation 6.23, give some general performance metrics for each task, given a set of strategies. Our agent uses them to estimate the overall expected utility of an execution strategy, which we will elaborate in Section 6.2.5.

However, so far we have looked at each task in isolation, calculated task durations without taking into account the initial provisioning time ($t_a(s_p)$) and we have used an artificial late probability.

In the following section, we address these issues by adding an initial waiting time to the task duration, and we elaborate on our use of the maximum late probability, showing how it is used to determine exactly when to start provisioning a task.

6.2.4 Provision Timing

In some environments, it may be beneficial for the service consumer to give a longer notice period to the service provider (indicated by a large $t_a(s_p)$). However, in these cases, the consumer either has to wait longer (if it provisions services only when the respective tasks become available), or it has to accept an additional risk (if it provisions services before the outcomes and completion times of any preceding tasks are certain). To express the amount of risk a consumer is willing to take when provisioning a particular task, we use the maximum late probability p_{ml} introduced in the previous section. This is the largest acceptable probability when provisioning task t_i that one of the predecessors of t_i will still not have been completed successfully by the time step t_i was provisioned for. More formally, the consumer will provision task t_i with primary strategy s_p at the earliest possible time step $\hat{t}$ where $p_{ml} \geq 1 - \mathcal{E}_i(\hat{t} + t_a(s_p))$. Expressing the starting time of a task in such a way allows us to succinctly express when to start provisioning relative to other tasks in the workflow.

Generally, as p_{ml} becomes smaller, the gap between the starting time of t_i and the end times of preceding tasks becomes larger. This means that the consumer may take longer to execute the workflow, but it also reduces the risk of expensive re-provisioning. To estimate this delay (denoted $\hat{w}_i$), we examine the predecessors of t_i and determine the task during which provisioning will take place so that the above condition for p_{ml} is satisfied. To this end, as in previous chapters, we again consider only the critical path to task t_i. We then proceed backwards along the critical path to identify the task during which to provision t_i, as shown in Algorithm 6.16. Here, the input $\mathcal{C}$ is a set of tasks on the critical path to task t_i, which we define as the longest path to the task considering the complete duration of each preceding task (the sum of the expected duration $\bar{t}_i$ and the waiting time $\hat{w}_i$). The functions d, w and v map each of the elements of $\mathcal{C}$ to their respective durations, waiting times and variances (as w of a given task is established by the algorithm, we run it iteratively in topological order over all workflow tasks).

The algorithm returns a tuple $r = (t_x, t, w, \hat{p}_l) : ((T \cup \{\mathsf{none}\}) \times \mathbb{N} \times \mathbb{R} \times [0, 1])$. Here, t_x is the task during which services for t_i should be provisioned (or the special case none if provisioning should start immediately) and t is the time of provisioning, relative to the starting time of task t_x (specifically, the first time step for which t_x will be provisioned). The returned value w is the expected amount of time between the last completion time of any of the predecessors of t_i and the first time step for which t_i was provisioned — this is effectively the expected time that the agent will waste due to provisioning services in advance. Finally, $\hat{p}_l$ is a revised late probability that is used by the consumer to update its calculations for the task, as described in the previous section ($\hat{p}_l \leq p_{ml}$).

Algorithm 6.16 Algorithm to determine the provisioning time.

1: **procedure** DETERMINEPROVISIONTIME($\mathcal{C}, d, w, v, p_{ml}, s_p, t_i$)
2: **if** $p_{ml} = 0 \ \vee \ |\mathcal{C}| = 0 \ \vee \ t_a(s_p) = 0$ **then**
3: **return** $(t_i, 0, t_a(s_p), 0)$ ▷ No advance provisioning required
4: **end if**
5: $d_{pre} \leftarrow 0$ ▷ Total duration of tasks preceding t_i
6: $v_{pre} \leftarrow 0$ ▷ Total variance of tasks preceding t_i
7: $t \leftarrow -1$ ▷ Provisioning time
8: **while** $\mathcal{C} \neq \emptyset \wedge t < 0$ **do** ▷ Step backwards along critical path
9: $t_x \leftarrow$ element of $\mathcal{C}$ that is nearest to t_i
10: $\mathcal{C} \leftarrow \mathcal{C} \setminus t_x$
11: $d_{pre} \leftarrow d_{pre} + d(t_x)$
12: $v_{pre} \leftarrow v_{pre} + v(t_x)$
13: $t \leftarrow \left\lceil \Phi^{-1}_{m_{pre}, v_{pre}}(1 - p_{ml}) \right\rceil - t_a(s_p)$ ▷ Determine target provisioning time
14: **if** $t < 0$ **then** ▷ Negative t indicates earlier provisioning
15: $d_{pre} \leftarrow d_{pre} + w(t_x)$ ▷ Add waiting time before t_x
16: **end if**
17: **end while**
18: **if** $t < 0$ **then**
19: $t_x \leftarrow$ none ▷ Provision immediately
20: $t \leftarrow 0$
21: **end if**
22: $\hat{p}_l \leftarrow 1 - \Phi_{m_{pre}, v_{pre}}(t + t_a(s_p))$ ▷ Calculate actual late probability
23: $w \leftarrow \int_t^{t+t_a(s_p)} \phi_{m_{pre}, v_{pre}}(x)(t + t_a(s_p) - x)\mathrm{d}x$ ▷ Calculate waiting time
24: **if** $t > 0$ **then**
25: $w \leftarrow w + t_a(s_p)\Phi_{m_{pre}, v_{pre}}(t)$ ▷ Add time if tasks complete early
26: **end if**
27: **return** $(t_x, t, w, \hat{p}_l)$
28: **end procedure**

Briefly, the algorithm begins in line 2 by considering the trivial case where $p_{ml} = 0$, where the task has no predecessors, or where the provisioning strategy contains no advance notice time. In these cases, the consumer will always start provisioning only when the task itself becomes available ($t_x = t_i$ and $t = 0$), it will always need to wait the advance provisioning period ($w = t_a(s_p)$) and there will never be any conflicts with preceding tasks ($\hat{p}_l = 0$).

In all other cases, the algorithm will work backwards from task t_i along the critical path to find a suitable task t_x for commencing the provisioning. At each step, it estimates the time it will take from that task until t_i becomes executable by using a normal distribution with mean and variance equal to the sum of all duration means and variances along the path so far. Using the late probability p_{ml}, the algorithm then determines the earliest acceptable provisioning time, relative to the start time of t_x (line 13). If this is negative, it continues to consider further predecessors of t_i. If no suitable task is found in the set of predecessors, the consumer will provision the task immediately (i.e., $t_x = \mathsf{none}$, line 19). Finally, the algorithm calculates the expected waiting time, considering both the case that the predecessors finish after provisioning but before t_i is started (line 23) and that they finish before t_i is even provisioned (line 25).

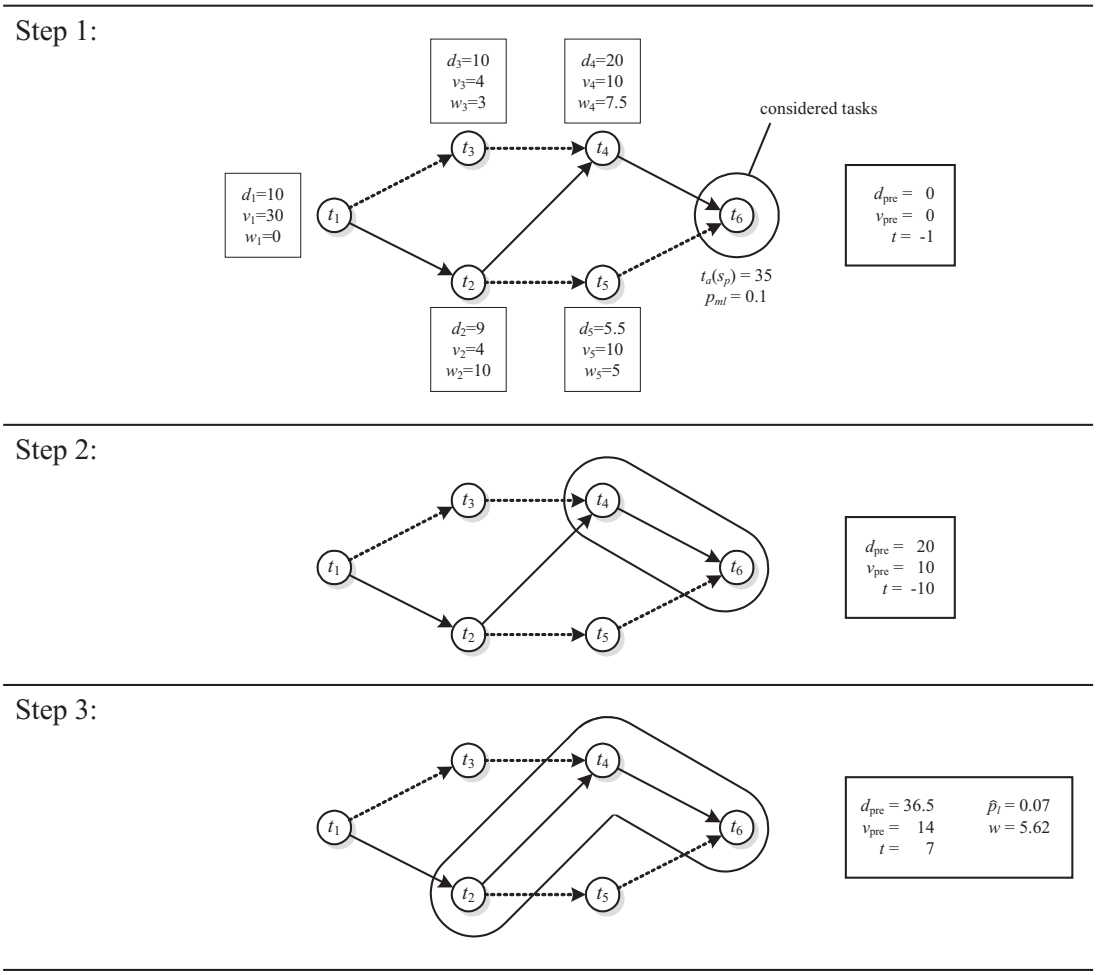


FIGURE 6.4: Algorithm 6.16 operating on an example workflow.

To illustrate this algorithm, Figure 6.4 shows how it determines the waiting time for a single task in an example workflow. Here, we assume that it has already been executed on tasks t_1 – t_5, which now have associated waiting times, and is about to examine task t_6. For this task, the agent has chosen an advance provisioning time of 35 time steps ($t_a(s_p) = 35$) and a maximum late probability $p_{ml} = 0.1$.

The algorithm starts from the task in question, t_6, and initialises the duration and variance of the predecessors it considers, as well as the current provisioning time (d_{pre}, d_{pre} and t). This is shown in step 1, which corresponds to the end of line 7 in the algorithm.

Following this, the algorithm enters its main loop and begins to traverse the critical path to task t_6 backwards (the critical path is shown by uninterrupted arrows). Step 2 shows the state of the algorithm at the end of line 13 during its first iteration. Here, it considers provisioning the task during the execution of t_4, but as the duration of the task is too short ($d_{\text{pre}} = 20$) compared to the required advance provisioning time of 35, the algorithm determines a negative provisioning time and so continues to consider tasks along the critical path.

Next, it examines task t_2, which is shown in step 3 (corresponding to the end of line 13 during the second iteration of the main loop). The algorithm estimates that tasks t_2 and t_4 will have a mean duration $d_{\text{pre}} = 36.5$ and variance $v_{\text{pre}} = 14$. This means that the target provisioning time in line 13 is positive with $t = 7$ (i.e., task t_6 should be provisioned 7 time steps after the starting time step of t_2). Before terminating, the algorithm calculates that the actual late probability of task t_6 is now only 7% ($\hat{p}_l = 0.07$) and that the expected waiting time associated with t_6 is 5.62.

Before we discuss the overall utility calculations of the workflow in the following section, it is important to note that the algorithm presented here is simply a heuristic approach for estimating the durations of tasks and for determining appropriate provisioning times. It relies on several simplifying assumptions that do not generally hold. Specifically, in contrast to Chapters 4 and 5, task durations are no longer independent of each other when the agent has provisioned offers in advance (i.e., when one task is taking longer, then this may have an impact on the duration of following tasks). Furthermore, our treatment of task waiting times simplifies the real problem, as they are not independent from task durations and may also lead to a reduction in variance along the workflow, which we do not consider here. Finally, the algorithm uses a normal distribution even when considering a small number of tasks, and this can lead to inaccurate results.

Despite these simplifications, we chose to adopt the algorithm to make fast predictions about waiting and provisioning times, where an accurate analytical solution is infeasible (for similar reasons that led us to adopt the critical path method in previous chapters). As we use an adaptive provisioning approach, these possibly inaccurate estimates are continuously revised during execution and eventually replaced by concrete offers, as we discuss in more detail in Section 6.2.7. Furthermore, our empirical experiments in Section 6.3 show that our approach works well in practice.

6.2.5 Utility Estimation

As discussed in the previous sections, we can now calculate a number of performance parameters for every task of the workflow, given a set of strategies and a maximum late probability for each task. This allows us to estimate the overall utility of the workflow. These calculations are similar to those employed in previous chapters, but we outline them briefly below for completeness.

First, the overall success probability of the workflow is simply the product of all task success probabilities:

$$p = \prod_{i \in \mathcal{I}} p_i \tag{6.32}$$

where $\mathcal{I}$ is the set of all task indices.

Next, the overall expected workflow cost can be estimated by taking the sum of all task execution costs, each multiplied by the probability that they are reached, and all reservation costs, each

multiplied by the probability that they are paid for:

$$\tilde{c} = \sum_{i \in \mathcal{I}} \left(c_{ei} \prod_{j \in B_i} s_j \; + \; c_{ri} \prod_{j \in \mathcal{P}_{r(i)}} s_j \right) \tag{6.33}$$

where B_i is the set of the indices of all tasks that precede t_i, and $r(i)$ is a function that returns the index of the task during which t_i will be provisioned (i.e., the index of t_x in Algorithm 6.16, or, if $t_x = \mathsf{none}$, we assume $\mathcal{P}_{r(i)} = \emptyset$ and $\prod_{j \in \emptyset} s_j = 1$).

We approximate the duration of the workflow again using the critical path and a normal distribution. To this end, we first attach a predicted completion time and variance to each task:

$$d_{i,\text{end}} = \hat{w}_i + \bar{t}_i + d_{i,\text{pre}} \tag{6.34}$$

$$v_{i,\text{end}} = v_i + v_{i,\text{pre}} \tag{6.35}$$

$$d_{i,\text{pre}} = \begin{cases} 0 & \text{if } B_i = \emptyset \\ \max_{j \in B_i} d_{j,\text{end}} & \text{otherwise} \end{cases} \tag{6.36}$$

$$v_{i,\text{pre}} = \begin{cases} 0 & \text{if } B_i = \emptyset \\ v_{\operatorname{argmax}_{j \in B_i} d_{j,\text{end}}} & \text{otherwise} \end{cases} \tag{6.37}$$

Next, we estimate the overall workflow duration and variance using the task that is expected to finish last:

$$\lambda_W = d_{l,\text{end}} \tag{6.38}$$

$$v_W = v_{l,\text{end}} \tag{6.39}$$

$$\text{where } l = \operatorname*{argmax}_{i \in \mathcal{I}} d_{i,\text{end}} \tag{6.40}$$

Given these, we estimate the final expected reward, conditional on overall success, using a normal approximation:

$$\tilde{r} = \int_0^\infty d_W(t) \cdot u(t) \mathrm{d}t \tag{6.41}$$

which can be written in closed form and quickly calculated as shown in Equations 4.28 and 4.29 in Section 4.4.3.2. Finally, we combine the parameters to derive an estimate for the overall expected utility:

$$\tilde{u} = p \cdot \tilde{r} - \tilde{c} \tag{6.42}$$

In the following section, we describe how we use this utility estimation technique to find a good provisioning strategy.

Algorithm 6.17 Local Search Algorithm

```
1:  procedure OPTIMISE(Ψ, n_max, n_fail, n_exp,Θ,α)
2:      i, f ← 0
3:      repeat
4:          Ψ′ ← GENERATERANDOMNEIGHBOUR(Ψ)
5:          Δũ ← PREDICTUTILITY(Ψ′) − PREDICTUTILITY(Ψ)
6:          if Δũ > 0 then
7:              Ψ, f ← Ψ′, 0
8:          else
9:              x ← drawn uniformly at random from [0, 1]
10:             if x ≤ e^{Δũ/(Θα^i)} then
11:                 Ψ ← Ψ′
12:             end if
13:         end if
14:         i, f ← i + 1, f + 1
15:     until i = n_max ∨ (f > n_fail ∧ i > n_exp)
16:     return Ψ
17: end procedure
```

6.2.6 Optimisation Algorithm

We again perform a local search to find a set of high-level strategies for each task. However, when employing the hill-climbing algorithms used in previous chapters, we noticed that the agent frequently ended its search in a local maximum, where it attempts to provision a single, cheap service for the first task and then gives up, thus obtaining a small negative profit. To avoid such behaviour, we decided to adopt simulated annealing, which is less prone to suffer from local maxima than deterministic local search techniques (Kirkpatrick et al. (1983)). The optimisation algorithm is shown in Algorithm 6.17 and follows the general structure of our previous algorithms. In particular, it is provided with an initial candidate solution, Ψ, which we here informally[5] assume to be a function that maps each workflow task t_i to a tuple consisting of the task's high-level strategies and its maximum late probability: $(s_{pi}, s_{ui}, s_{fi}, s_{li}, p_{mli})$. Given this, the algorithm then repeatedly generates a random neighbour of Ψ (line 4), accepting it as the new candidate solution if it yields a higher utility than the original (line 7), or, with a certain probability, if its utility is less. As is common in simulated annealing, this probability depends on the utility difference, an initial temperature Θ, a decay factor α and the number of steps so far. The algorithm terminates after n_{max} steps or if a better solution has not been found after n_{fail} consecutive attempts (this applies only after the first n_{exp} steps, to allow the algorithm an initial exploration phase).

For the neighbour generation in line 4, we first choose uniformly at random[6] whether to change the strategy associated with a particular task or the structure of the workflow. In the former case, we pick a random task t_i and randomly apply one of the following changes:

[5] We will expand on this in Section 6.2.8.

[6] All random choices in this section assign equal probabilities to all outcomes.

- All strategies (s_{pi}, s_{ui}, s_{fi}, s_{li}) and the late probability, p_{mli}, are re-assigned randomly from the available options.
- One of the task strategies, ω, is picked and changed to ω' so that exactly one of its parameters ($t_a(\omega')$, $t_w(\omega')$, $n(\omega')$, $\vartheta(\omega')$) is different from the original. This is done in one of four ways: either by increasing or decreasing the parameter by a single step, or by randomly choosing one of the remaining higher or lower values.
- One of the task strategies, ω, is picked and changed in one of the three following ways: to a random ω', to its repeated or non-repeated equivalent, or to ω_{null}.
- The late probability, p_{mli}, is changed to p'_{mli} in one of three ways: by randomly choosing a value from $(p_{mli}, 1)$, from $(0, p_{mli})$, or by setting $p'_{mli} = 0$.

When altering the structure of the workflow, we change the precedence constraints E to E' by either introducing or removing temporary edges. This allows us to represent the fact that the consumer may prefer to delay the provisioning or invocation of certain tasks until the outcome of other tasks is known. For example, the consumer might decide to delay a particularly expensive task until it knows the outcome of another, highly unreliable task. Clearly, we never remove the original edges in E, pick only from new edges that do not introduce cycles and we update transitive dependencies, so that E' remains a strict partial order.

In testing our optimisation algorithm, we noticed that we could consistently improve its performance by making small adjustments, which are, for brevity, not shown in Algorithm 6.17. First, we apply an additional penalty to solutions that result in a negative expected utility, to generate a new expected utility value, $\tilde{u}'$, as follows:

$$\delta_{\text{fail}} = (1-p)u_{\max} \tag{6.43}$$

$$\delta_{\text{late}} = (\lambda_W / t_{\text{d}} - 1)u_{\max} \tag{6.44}$$

$$\tilde{u}' = \begin{cases} \tilde{u} & \text{if } \tilde{u} > 0 \\ \tilde{u} - \delta_{\text{fail}} & \text{if } \tilde{u} \leq 0 \wedge \lambda_W \leq t_{\text{d}} \\ \tilde{u} - \delta_{\text{fail}} - \delta_{\text{late}} & \text{otherwise} \end{cases} \tag{6.45}$$

This further encourages the algorithm to avoid the local maximum described above. Second, we found that we could generally decrease the time to find a good solution by immediately reconsidering the same neighbour generation strategy in line 4 if the previously generated neighbour yielded a higher utility.

So far, we have discussed how the consumer can make high-level decisions about the provisioning of its workflow. In the next section, we describe how our mechanism is extended to deal with new information as it becomes available during execution.

6.2.7 Dynamic Adaptation

As we use a local search approach, our provisioning strategy is easily extended to incorporate information at run-time and act on it if necessary. For example, if seemingly reliable services suddenly fail, the agent may need to re-provision the task and possibly even change its strategies for later tasks in the workflow, in order to meet its deadline. Similarly, the agent may come across new opportunities; for example, if it discovers a particularly attractive offer on the market and is able to immediately provision it for a current task.

From the discussions above, it should be clear that it is straight-forward to incorporate information about the performance of services into our calculations. First, when the consumer provisions services for a particular task (according to s_p and at the time determined by the procedure in Section 6.2.4), we use the calculations in Section 6.2.2 for a provisioning decision γ_i to immediately replace those for s_p. This gives us a more accurate estimate of the probabilities of various outcomes, the late probability, the completion time and the cost for the task. Similarly, as services fail, we remove them from their respective tasks, and when reservation or invocation payments are made, we remove the respective costs from the calculations, as we aim to maximise the expected utility of the remaining workflow.

Next, we also refine the overall completion time of the task. Specifically, we consider two cases: the preceding tasks finish in time for at least one of the provisioned offers to be invoked or they finish too late for any provisioned offer to be invoked. In the former case, we can use the equations from Sections 6.2.2 and 6.2.3 with minor modifications to derive a probability distribution for the completion time that assigns probabilities to the various end times of the provisioned offers and uses a normal approximation if the provisioned offers fail. In the latter case, when there is a conflict with the previous task, we use a normal approximation with mean and variance as follows:

$$m_{i,\text{late}} = t_a(s_l) + \check{d}_s(s_l) + (1 - \mathcal{E}_i(\hat{t}_s))^{-1} \int_{\hat{t}}^{\infty} \mathcal{E}_i'(x) x \mathrm{d}x \tag{6.46}$$

$$v_{i,\text{late}} = \tilde{v}_s(s_l) + (1 - \mathcal{E}_i(\hat{t}))^{-1} \int_{\hat{t}}^{\infty} \mathcal{E}_i'(x) x^2 \mathrm{d}x - m_{i,\text{late}}^2 \tag{6.47}$$

Combining these two cases into a single distribution (each occurring with probability $1 - \hat{p}_l$ and $\hat{p}_l$, respectively) gives us a more accurate estimate of the completion time for the task, as we now take into account the provisioned offers. We use this distribution instead of the simpler normal approximation as $\mathcal{E}_i$ in the calculations above and in those presented in Section 6.2.2.

Furthermore, we modify the neighbour generation procedure described in Section 6.2.6 to consider adding to or removing offers from an already provisioned task. These are chosen randomly from all available offers or from the set of offers that the agent plans to provision during that time step (as we will discuss in the next section, offers are not provisioned until the end of a time step). More specifically, in addition to changing the structure and high-level provisioning strategies during the neighbour generation procedure, we include the possibility of changing a

provisioned task. When this occurs, we select a random task that has a concrete provisioning decision γ_i, and randomly carry out one of the following changes:

- **Add offer:** We first sample a value t_r from an exponential distribution with mean $\lambda^{-1} = \frac{1}{|\gamma_i|}\sum_{o\in\gamma_i} t(o) - t_{\text{min}}$, where t_{min} is the lowest starting time in γ_i (when $\sum_{o\in\gamma_i} t(o) - t_{\text{min}} = 0$, we use $\lambda^{-1} = 1$). Then we submit a request for offers for time step $t = t_r + t_{\text{min}} - 20$, and add a random returned offer to γ_i. This process allows us to select a random offer, but with a bias towards offers at a similar time as those already in γ_i.
- **Remove offer:** If $|\gamma_i| > 1$, select a random offer that has been added to γ_i during the same time step and remove it again.

Finally, we also modify Algorithm 6.16 to terminate its main loop when it examines a task that has already been provisioned. This is because the agent has already decided when to start invoking that task and, as a result, the normal approximation will be far less accurate. If t is still negative at this stage, we use the starting time of the provisioned task as an anchor and infer the absolute provisioning time from there (e.g., if $t = -10$ and the earliest provisioned service is to start at time step $t(o) = 120$, the algorithm returns the time $t = 110$ and specifies the target task $t_x = \mathsf{none}$ to signal that the task should be provisioned at an absolute time step).

To conclude our discussion of the dynamic flexible strategy, we now summarise it in the context of our generic agent algorithm from Section 3.4.

6.2.8 Updated Generic Algorithm

In this section, we provide a final overview of the strategy that addresses the optimisation problem outlined in Section 6.2.1. In particular, building on the work described in previous sections, we now define the provisioning strategy Ψ more formally as a tuple:

$$\Psi = (\alpha, \beta, \gamma, d_\beta, d_\gamma, E') \tag{6.48}$$

where α, β and γ are a set partition of T, describing the current state of each workflow task. Here, α contains the tasks that have been completed successfully, β contains the tasks for which some offers have been negotiated, and γ contains the tasks for which no offers are currently provisioned. The functions d_β and d_γ provide further information about the agent's high-level decisions for the members of β and γ, respectively. Based on previous sections, $d_\beta(t_i)$ of a provisioned task $t_i \in \beta$ is:

$$d_\beta(t_i) = (\gamma_i, s_{li}, s_{ui}, s_{fi}) \tag{6.49}$$

where γ_i is the set of offers already provisioned for t_i, while the other objects refer to the contingent strategies. Similarly, $d_\gamma(t_j)$ of a task $t_j \in \beta$ is:

$$d_\gamma(t_j) = (s_{pj}, s_{lj}, s_{uj}, s_{fj}, p_{mlj}) \tag{6.50}$$

Algorithm 6.18 Summary of Flexible Provisioning Strategy

```
1:  t̂ ← 0
2:  Ψ ← create initial strategy
3:  abandoned ← false
4:  repeat
5:      Ψ ← update strategy with recent service outcomes
6:      repeat
7:          Ψ ← local search for better strategy
8:          Ψ ← use high-level strategies to provision services
9:      until Ψ was not altered in line 8
10:     if PREDICTUTILITY(Ψ) > 0 then
11:         provision new services
12:         invoke services that are due
13:     else
14:         abandoned ← true
15:     end if
16:     t ← t + 1
17: until abandoned = true or workflow completed
```

where s_{pj} is the primary provisioning decision and p_{mlj} is the late probability. Finally, E' : $\mathcal{P}(T \times T)$, is the current set of edges.

Given this, Algorithm 6.18 contains a high-level overview of the *dynamic flexible* strategy. At time $\hat{t} = 0$, the consumer creates an initial execution strategy Ψ to form the basis of its local search[7] (line 2). Then, at each time step, the consumer first updates its current plan with any service outcomes (line 5), followed by an optimisation process that refines the plan by changing its high-level task strategies and by altering already provisioned offers (line 7), as described in the previous two sections. In line 8, the agent considers the provisioning of due tasks, as determined by the algorithm described in Section 6.2.4. It does this by carrying out the associated primary strategy, but only temporarily associates the chosen offers with the workflow for now (they are not yet explicitly provisioned). If any such provisions are added to the workflow, the consumer then repeats the optimisation stage, so that the initially chosen offers can be improved (and possibly replaced by better ones), and this continues until no more new tasks are provisioned.

Following that, if the consumer expects to receive a positive utility from continuing the plan, it provisions any new offers that have been added to the workflow during that time step and invokes due services (lines 11 and 12). This procedure continues until the consumer either does not expect to gain any utility from its current plan or the workflow is completed.

Clearly, it is time-consuming for the service-consumer to carry out a long optimisation stage during every time step of the simulation — especially as the expected utility of the workflow

[7] In our work, we start with a simple allocation that uses ω_r with $t_a(\omega_r) = 0$, $t_w(\omega_r) = 10$, $n(\omega_r) = 1$ and $\vartheta(\omega_r) =$ `unreliability` as the primary and contingent strategies (all repeated) for every task and set $p_{ml} = 0.01$. We believe that this already constitutes a feasible strategy in most environments, as it includes repeated provisioning to deal with failures but without relying on expensive redundancy. We have empirically verified this and noted a quicker convergence than a completely random initial strategy.

will not change at each step. Hence, we have found it sufficient to carry out further optimisation of the current allocation only when its expected utility changes significantly from an earlier estimate, and also to vary the amount of time spent during the optimisation depending on the magnitude of the change in utility.

More specifically, we have experimented with various optimisation strategies and found the following approach to work quickly and effectively in a variety of environments. First, we always carry out an extensive initial simulated annealing run with the parameters given in the first row of Table 6.5. This is repeated up to 3 times if the resulting allocation does not yield a positive expected utility. Then, at each time step, we calculate the difference between the current expected utility and the total costs incurred so far. We carry out a "long" optimisation run (see Table 6.5) if this value is at least 40% higher or lower than the same value when this was last run. Otherwise, if it is at least 20% higher or lower than after the last optimisation run, we run a "quick" optimisation procedure. Clearly, these parameters can be easily adjusted for particular problems. For example, when time is critical, n_{max} can be set to a fixed cut-off time.

	n_{max}	n_{fail}	n_{exp}	Θ	α	Threshold
initial	-1	5000	2000	100	0.999	-
short	-1	75	200	50	0.99	0.2
long	-1	1000	500	50	0.99	0.4

TABLE 6.5: Simulated annealing parameters.

For completeness, Algorithms 6.19 and 6.20 contain more detailed descriptions of the strategy, based on the generic agent algorithm from Section 3.4. To fit our extended model, we now assume that the parameter to the UPDATE procedure is a set of tuples $\mathcal{O} : \mathcal{P}(\mathbb{C} \times T \times \{\texttt{succeeded}, \texttt{failed}\})$, each of which indicates that an offer for a particular task has either been successful or failed (this includes both a defection and failure with compensation). For the sake of readibility, we have left a number of procedures undefined, as these are straight-forward, but would require a number of additional data-structures and housekeeping procedures. Instead, we outline them only briefly below:

- REALISESTRATEGIES(Ψ): This procedure iterates through all tasks in γ and identifies those that are due to be provisioned, based on t_x and t returned by the DETERMINEPROVISIONTIME procedure (this is the case either if $t_x = \texttt{none}$ and $t \leq \hat{t}$, or if the earliest offer for t_x was invoked t or more time steps ago). It then requests and provisions offers for those tasks based on the associated primary strategy. If none are found, it adopts s_{ui} and ignores the task for the remainder of the time step.
- PROVISIONSERVICES(Ψ): Any offers that the agent has decided to provision in this time step (during the REALISESTRATEGIES and OPTIMISE procedures) are now actually provisioned.

Before we outline the empirical results of the strategy, we now briefly discuss an illustrative example of how it provisions workflows in practice.

Algorithm 6.19 Main procedures of the dynamic flexible provisioning strategy.

procedure ADVANCE-FLEXIBLE-INITIALISE(W)
  $u_{\text{long}} \leftarrow 0$ ▷ To store utility of last long optimisation
  $u_{\text{short}} \leftarrow 0$ ▷ To store utility of last medium optimisation
  $i \leftarrow 0$
  repeat
    $\Psi \leftarrow$ GENERATE-INITIAL(W) ▷ Generate initial strategy
    $\Psi \leftarrow$ OPTIMISE($\Psi, -1, 5000, 2000, 100, 0.999$) ▷ Optimise strategy
    $i \leftarrow i + 1$
  until PREDICTUTILITY(Ψ)$> 0 \vee i = 3$
end procedure

procedure GENERATE-INITIAL(W)
  $d_\gamma \leftarrow \{(t_i, (\omega_r, \omega_r, \omega_r, \omega_r, 0.01)) \mid t_i \in T\}$ ▷ Initial decision[a]
  $\Psi \leftarrow (\emptyset, \emptyset, T, \emptyset, d_\gamma, E)$
  return Ψ
end procedure

procedure ADVANCE-FLEXIBLE-UPDATE($\mathcal{O}$)
  for all $(o_x, t_i, \texttt{succeeded}) \in \mathcal{O}$ **do** ▷ Iterate through successful offers
    $\alpha \leftarrow \alpha \cup \{t_i\}$ ▷ Add to successful tasks
    $\beta \leftarrow \alpha \setminus \{t_i\}$
    remove mapping of t_i from d_β
  end for
  for all $(o_x, t_i, \texttt{failed}) \in \mathcal{O}$ **do** ▷ Iterate through failed offers
    if $t_i \in \beta$ **then**
      $\gamma_i \leftarrow \gamma_i \setminus \{o_x\}$ ▷ Remove offer
      if $\gamma_i \neq \emptyset$ **then**
        $d_\beta(t_i) \leftarrow (\gamma_i, s_{li}, s_{ui}, s_{fi})$ ▷ Update offers for task
      else
        $\beta \leftarrow \beta \setminus \{t_i\}$ ▷ No longer provisioned
        $\gamma \leftarrow \gamma \cup \{t_i\}$
        $s_{pi} \leftarrow s_{fi}$ ▷ Adopt failure strategy
        if s_{fi} is repeating **then**
          $s_{li}, s_{ui}, s_{fi} \leftarrow s_{fi}$ ▷ Repeat failure strategy in future
        else
          $s_{li}, s_{ui}, s_{fi} \leftarrow \omega_{\text{null}}$ ▷ Do not repeat in future
        end if
        $d_\gamma(t_i) \leftarrow (s_{pi}, s_{li}, s_{ui}, s_{fi}, p_{li})$ ▷ Store strategy
        remove mapping of t_i from d_β
      end if
    end if
  end for
  repeat similarly for late[b] offers (adopting p_{li} if necessary)
end procedure

[a] Here, ω_r is chosen such that $t_a(\omega_r) = 0$, $t_w(\omega_r) = 10$, $n(\omega_r) = 1$ and $\vartheta(\omega_r) =$ `unreliability`.

[b] These are offers that are due this turn ($t(o_x) = \hat{t}$), but cannot be invoked as their predecessors have not been completed yet.

Algorithm 6.20 Implementation of the NEGOTIATESERVICES procedure.

```
1:  procedure ADVANCE-FLEXIBLE-STOPCONDITION
2:      return PREDICTUTILITY(Ψ) ≤ 0
3:  end procedure

4:  procedure ADVANCE-FLEXIBLE-INVOKESERVICES
5:      for all t_i ∈ β do                          ▷ Iterate through provisioned tasks
6:          for all o_x ∈ γ_i do                    ▷ Iterate through all offers for task t_i
7:              if t(o_x) = t̂ then                  ▷ Check time
8:                  INVOKE(o_x, t_i)                ▷ Invoke offer
9:              end if
10:         end for
11:     end for
12: end procedure

13: procedure ADVANCE-FLEXIBLE-NEGOTIATESERVICES
14:     long ← false
15:     short ← false
16:     repeat
17:         u_new ← PREDICTUTILITY(Ψ) + p̂
18:         if |u_long/u_new − 1| ≥ 0.4 then
19:             long ← true
20:             Ψ ← OPTIMISE(Ψ, −1, 1000, 500, 50, 0.99)      ▷ Long optimisation
21:         else if |u_short/u_new − 1| ≥ 0.2 then
22:             short ← true
23:             Ψ ← OPTIMISE(Ψ, −1, 75, 200, 50, 0.99)        ▷ Short optimisation
24:         end if
25:         Ψ_pre ← Ψ
26:         Ψ ← REALISESTRATEGIES(Ψ)                  ▷ Request offers and add to Ψ
27:     until Ψ = Ψ_pre                               ▷ ...until no new offers are added to Ψ
28:     u_new ← PREDICTUTILITY(Ψ) + p̂
29:     if long = true then                           ▷ Store utility values for future
30:         u_long ← u_new
31:         u_short ← u_new
32:     else if short = true then
33:         u_short ← u_new
34:     end if
35:     PROVISIONSERVICES(Ψ)                          ▷ Provision all new offers
36: end procedure
```

6.2.9 Illustrative Example

To show how the *dynamic flexible* strategy works in practice, we again consider the bioinformatics workflows from Section 3.5. In order to simulate a dynamic service market in this scenario, we keep a list of currently available offers associated with each time step, from the current step $\hat{t}_i$ to $\hat{t}_{i+60}$ (hence, the consumer may provision services up to 60 time steps in advance). During the simulation, at the beginning of each time step, we first generate new offers that become available in the market by drawing the number of new offers and their parameters from random distributions. These distributions are detailed in Table 6.6 and depend on the number of time steps the offer is generated in advance. This time dependency allows us to include performance differences in offers when they are provisioned with varying advance notice periods. It is expressed here by including two rows of distributions for each service type — the first indicates the performance of offers when provisioned at short notice (as given by the *advance time* column), and the second gives the performance when offers are provisioned with a long advance notice period.

Service Type	Adv. Time (min.)	Fail. Prob	Reserv. Cost ($)	Exec. Cost ($)	Time (min.)	Birth/ Death	Repay
Base C.	≤ 0	$\mathcal{U}_c(0.2, 0.5)$	$\mathcal{U}_c(0.5, 1.0)$	$\mathcal{U}_c(0, 0)$	$\mathcal{U}_c(6, 10)$	1/3	no
	≥ 10	$\mathcal{U}_c(0.1, 0.2)$	$\mathcal{U}_c(0.5, 1.0)$	$\mathcal{U}_c(0, 0)$	$\mathcal{U}_c(1, 2)$	0.1/0	no
Gene A.	≤ 5	$\mathcal{U}_c(0.1, 0.2)$	$\mathcal{U}_c(2, 5)$	$\mathcal{U}_c(2, 5)$	$\mathcal{U}_c(10, 30)$	1/0.5	yes
	≥ 10	$\mathcal{U}_c(0, 0.1)$	$\mathcal{U}_c(1, 2)$	$\mathcal{U}_c(0, 0)$	$\mathcal{U}_c(5, 10)$	0.5/1	yes
Blast	≤ 5	$\mathcal{U}_c(0.5, 1)$	$\mathcal{U}_c(2, 3)$	$\mathcal{U}_c(1, 5)$	$\mathcal{U}_c(20, 40)$	1/2	no
	≥ 15	$\mathcal{U}_c(0, 0.1)$	$\mathcal{U}_c(1, 5)$	$\mathcal{U}_c(2, 3)$	$\mathcal{U}_c(5, 10)$	0.5/0.5	no
LookUp	≤ 0	$\mathcal{U}_c(0.5, 0.7)$	$\mathcal{U}_c(0, 0)$	$\mathcal{U}_c(4, 10)$	$\mathcal{U}_c(2, 8)$	0.5/0.25	yes
	≥ 1	$\mathcal{U}_c(0.5, 0.7)$	$\mathcal{U}_c(0, 0)$	$\mathcal{U}_c(4, 10)$	$\mathcal{U}_c(2, 8)$	0.5/0.25	yes
Render	≤ 15	$\mathcal{U}_c(0.2, 0.3)$	$\mathcal{U}_c(5, 10)$	$\mathcal{U}_c(10, 15)$	$\mathcal{U}_c(150, 240)$	0.5/1	no
	≥ 30	$\mathcal{U}_c(0, 0)$	$\mathcal{U}_c(5, 10)$	$\mathcal{U}_c(5, 10)$	$\mathcal{U}_c(80, 120)$	1/1	no
Transl.	≤ 10	$\mathcal{U}_c(0.7, 1.0)$	$\mathcal{U}_c(2, 5)$	$\mathcal{U}_c(1, 2)$	$\mathcal{U}_c(10, 40)$	0.5/1	no
	≥ 30	$\mathcal{U}_c(0.3, 0.4)$	$\mathcal{U}_c(0.1, 0.5)$	$\mathcal{U}_c(0.25, 0.5)$	$\mathcal{U}_c(5, 10)$	0.5/0.5	no
Fold	≤ 15	$\mathcal{U}_c(0.25, 0.75)$	$\mathcal{U}_c(5, 20)$	$\mathcal{U}_c(5, 20)$	$\mathcal{U}_c(80, 400)$	3/5	yes
	≥ 45	$\mathcal{U}_c(0, 0.05)$	$\mathcal{U}_c(10, 20)$	$\mathcal{U}_c(20, 30)$	$\mathcal{U}_c(20, 30)$	1/1	yes
Print	≤ 0	$\mathcal{U}_c(0.4, 0.8)$	$\mathcal{U}_c(1, 2)$	$\mathcal{U}_c(0, 0)$	$\mathcal{U}_c(8, 12)$	2/2	yes
	≥ 1	$\mathcal{U}_c(0.4, 0.8)$	$\mathcal{U}_c(1, 2)$	$\mathcal{U}_c(0, 0)$	$\mathcal{U}_c(8, 12)$	2/2	yes

TABLE 6.6: Distributions used to generate random offers for bioinformatics services.

In more detail, for each service type and for each possible time step from $\hat{t}_i$ to $\hat{t}_{i+60}$, we first generate the number of new offers by drawing a sample from a Poisson distribution with a mean given by the respective *birth* rate[8]. Then, for each such generated offer, we draw its failure probability, reservation cost, execution cost and service duration from the relevant distributions (depending on how far the offer is generated in advance). When the offer time lies between the two extremes corresponding to the two rows for each service type, we interpolate linearly between the distribution parameters. For example, when generating an offer for the *Base Call* service type for time step $\hat{t}_{i+2}$, we draw its failure probability from $\mathcal{U}_c(0.18, 0.44)$. If the service

[8]We chose the Poisson distribution here because it is a common distribution for modelling random arrival events (DeGroot and Shervish (2002))

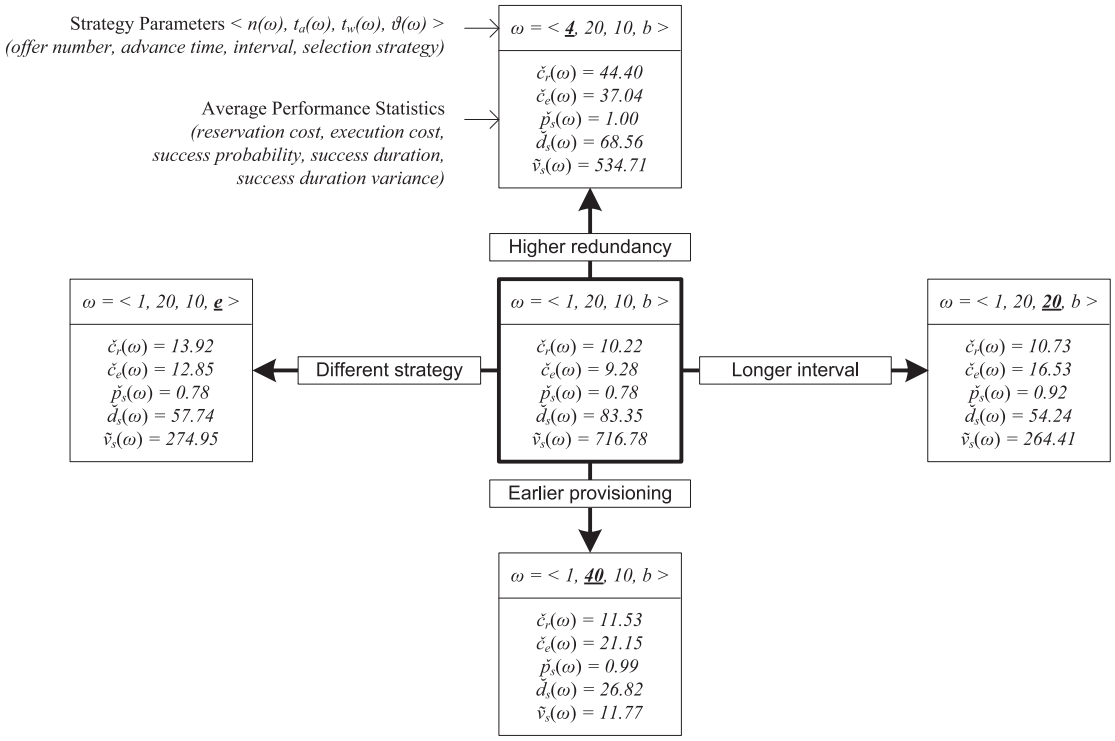


FIGURE 6.5: Example high-level strategies for the *Fold* task.

type is marked as repaying in the table, we set the failure penalty to the combined execution and reservation costs (this means that providers for these services always refund a consumer in case of failures). Otherwise, it is set to 0. Finally, at the end of each time step, we remove offers in a similar way as above by drawing a random sample from a Poisson distribution with its mean given by the *death* rate. This models the demand for such services and we randomly remove the generated number of offers from that time step (or all offers if the number exceeds the current supply).

The service parameters in Table 6.6 are broadly similar to those used in previous chapters, but we now include significant performance differences depending on the time of provisioning. For example, some services are now generally more reliable and faster when provisioned in advance (*Base Call* and *Gene Assemble*), some offer a far better service overall but are also significantly more expensive and need a long advance notice period (*Fold*), and a few display no difference in quality over time (*Look Up* and *Print*).

As described in Section 6.2.3, we assume that the service consumer has already obtained a library of atomic high-level strategies by observing the market over some time. To illustrate this, Figure 6.5 shows a number of example strategies and their statistics for the *Fold* service type. In the centre, we use the notation $\omega = \langle 1, 20, 10, b \rangle$ to represents a strategy ω that provisions a single offer ($n(\omega) = 1$) 20 time steps before it is required ($t_a(\omega) = 20$), with the consumer considering 10 consecutive time steps ($t_w(\omega) = 10$) and selecting the offer that best balances

all performance characteristics ($\vartheta(\omega) = \texttt{balanced}$)[9]. We note that this strategy is relatively cheap, has a success probability of only 78% and takes a long and highly uncertain time to complete. The remaining strategies shown in Figure 6.5 demonstrate the impact of slightly altering this provisioning strategy — for example, when increasing the advance notice period from 20 to 40 time steps, the cost rises, but the success probability also increases to 99%, while the duration and its variance drop significantly (these trends all emerge from the distributions given in Table 6.6).

Given these strategies, Figure 6.6 shows the initial high-level decisions that our agent takes for the bioinformatics workflow (as in previous chapters, we first consider a low-value, non-urgent workflow with maximum utility $u_{\max} = 150$, deadline $t_{\max} = 240$ and penalty $\delta = 1$). Here, the agent generally attempts to spend as little on services as possible, preferring to wait longer for completion. Thus, the algorithm decides to provision only single offers for most tasks and relies on cautious contingency plans, where more single offers are provisioned gradually (and repeatedly) in case of failure. The only parallel redundancy in the plan is used for tasks t_5 (*Translate*) and t_8 (*Print*), which are relatively cheap. Due to the longer deadline in this case, the consumer also decides to include few task overlaps in the workflow and instead prefers to leave the provisioning of each task until all predecessors are complete. The only exception to this is t_5 (*Translate*), which the consumer chooses to provision while its predecessors are still executing. Using Algorithm 6.4, the strategy here determines that the task should be provisioned immediately when the workflow is started (for time step 30) and that this will result in a 5% probability of a losing the provisioned offers later on and an additional delay of 9.02 time steps. These figures are based on the uncertain duration of its predecessors (*BaseCall* and *GeneAssemble*), which are expected to complete by $\hat{t} = 21.08$. Overall, the consumer expects the workflow to finish just before the deadline, after 224.75 time steps (but with considerable variance) and expects to spend \$66.60, thus achieving an expected utility of $\tilde{u} = 81.73$.

Next, Figure 6.7 shows the same workflow after the first offers have been provisioned (during the inital time step). Here, the agent has consulted the market and followed its high-level strategies in reserving offers for some of the workflow tasks. In particular, the consumer has now provisioned three offers for t_0 (*BaseCall*). In this case, it is different from the initial decision of provisioning a single offer, as the agent immediately revises and improves its decisions as it observes the actual offers available on the market. Given these three offers, the task parameters are updated to reflect their terms (hence, the task end time is now almost certain). As is evident in the remainder of the workflow, the consumer has also now adapted its high-level strategies based on the new information. In particular, knowing that t_0 is almost certain to complete by time step $\hat{t} = 11$, it has decided to provision task t_1 (*GeneAssemble*) earlier than originally planned. On the other hand, it also delayed the provisioning of task t_5 (*Translate*) to a later time. Finally, the consumer has introduced a number of additional edges into the workflow. Some of these have no impact on the estimated workflow utility, but the additional edge between t_3 (*LookUp*) and t_7

[9] Here, and in the remainder of the section, we abbreviate each selection strategy in {`cost`, `unreliability`, `end_time`, `balanced`} with its first letter.

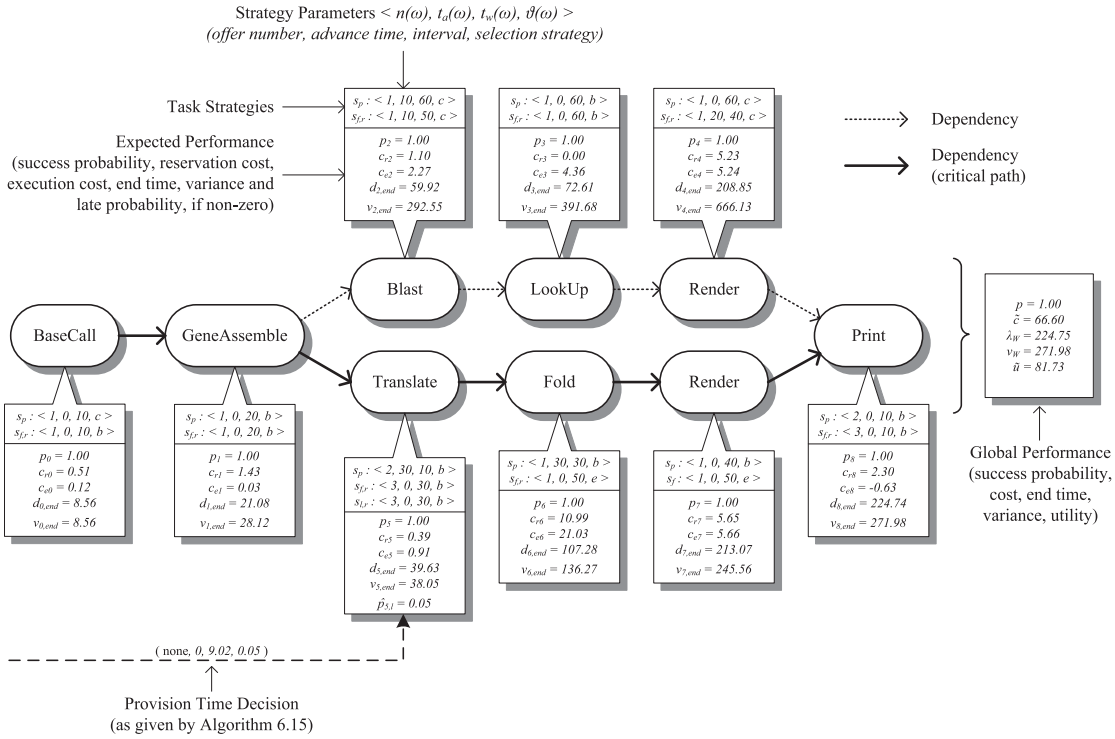


FIGURE 6.6: Workflow with initial high-level decisions.

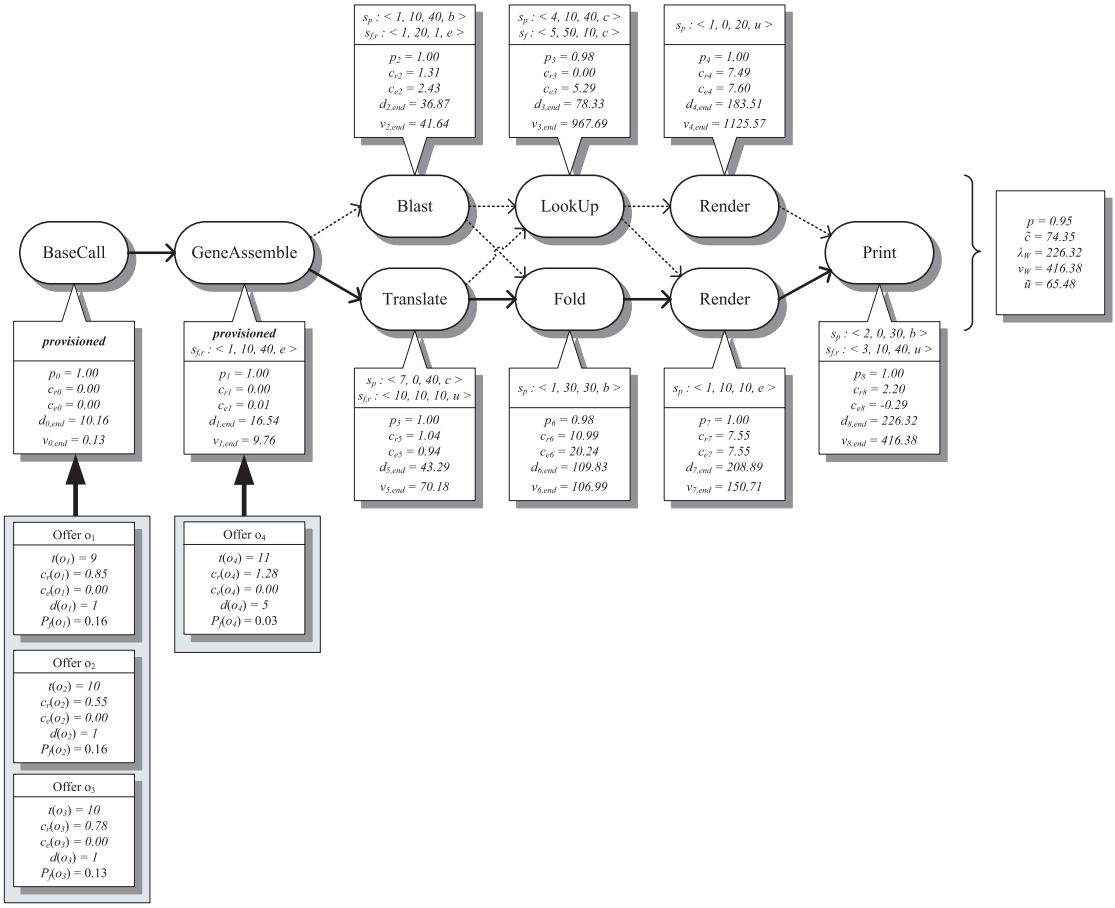


FIGURE 6.7: Workflow after first provisioning.

(*Render*) ensures that the former, slightly uncertain task completes before the expensive *Render* task is started.

In order to investigate how the agent's provisioning decisions change as the workflow becomes more urgent and valuable, we now consider a second scenario. For this, as in previous chapters, we assume a maximum utility $u_{\max} = 1000$, a deadline $t_{\max} = 150$ and penalty $\delta = 20$. Figure 6.8 illustrates the initial high-level provisioning decisions, clearly highlighting an increasing reliance on redundant provisioning and also on advance provisioning, which here allows the agent to obtain better services and decrease the overall execution time (as services are provisioned before their predecessors are completed). In more detail, the agent here decides to provision some tasks immediately (such as *Fold*), but leaves the provisioning of others until later (such as the lower *Render* task), according to the advance notice periods required and the expected completion times of their predecessors.

Next, Figure 6.9 shows the workflow after the first offers have been provisioned. Here, the strategy has mostly followed its initial decisions. However, based on the offers provisioned for task t_0 (*BaseCall*), it also immediately provisioned t_5 (*Translate*), rather than waiting an additional two time steps. As the offers were generally as expected, most remaining high-level decisions are unchanged and the overall expected utility has risen slightly, due to an earlier estimated completion time.

In the following section, we discuss a number of experiments we carried out to investigate some more general trends of our *dynamic flexible* strategy and to compare its performance to other current strategies.

6.3 Empirical Evaluation

As in previous chapters, we have conducted a thorough empirical study of our algorithm in a simulated environment and compared it to a number of current approaches. The primary focus of this section is to investigate the feasibility of our approach in environments of varying uncertainty (i.e., where services are more or less likely to fail) and also in environments where the market favours certain provisioning approaches (e.g., where early provisioning is rewarded by more reliable services). In the following, we first describe how we simulate the market (Section 6.3.1), then we detail the strategies we test (Section 6.3.2), draw up a number of hypotheses (Section 6.3.3) and finally describe our results in Sections 6.3.4–6.3.6.

6.3.1 Market Setup

In our experiments, we assume that there are five different types of services ($\mathcal{T} = \{T_1, T_2, T_3, T_4, T_5\}$). We simulate the dynamic market as described in Section 6.2.9, but now increase the advance provisioning time to 250 time steps. Furthermore, this time we generate offers using

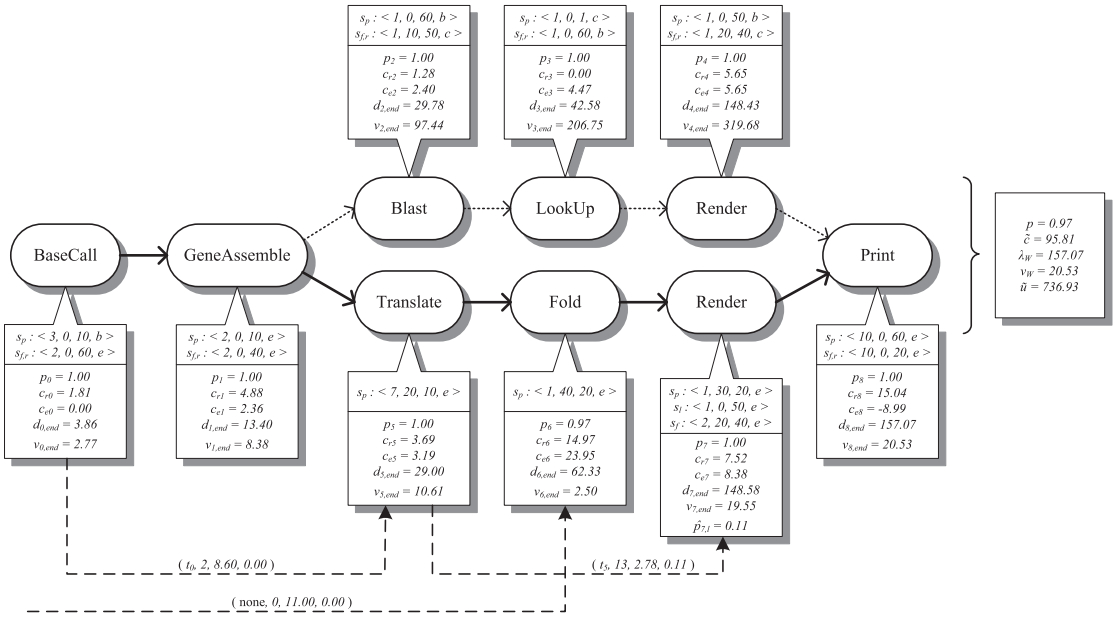


FIGURE 6.8: Urgent workflow with initial high-level decisions.

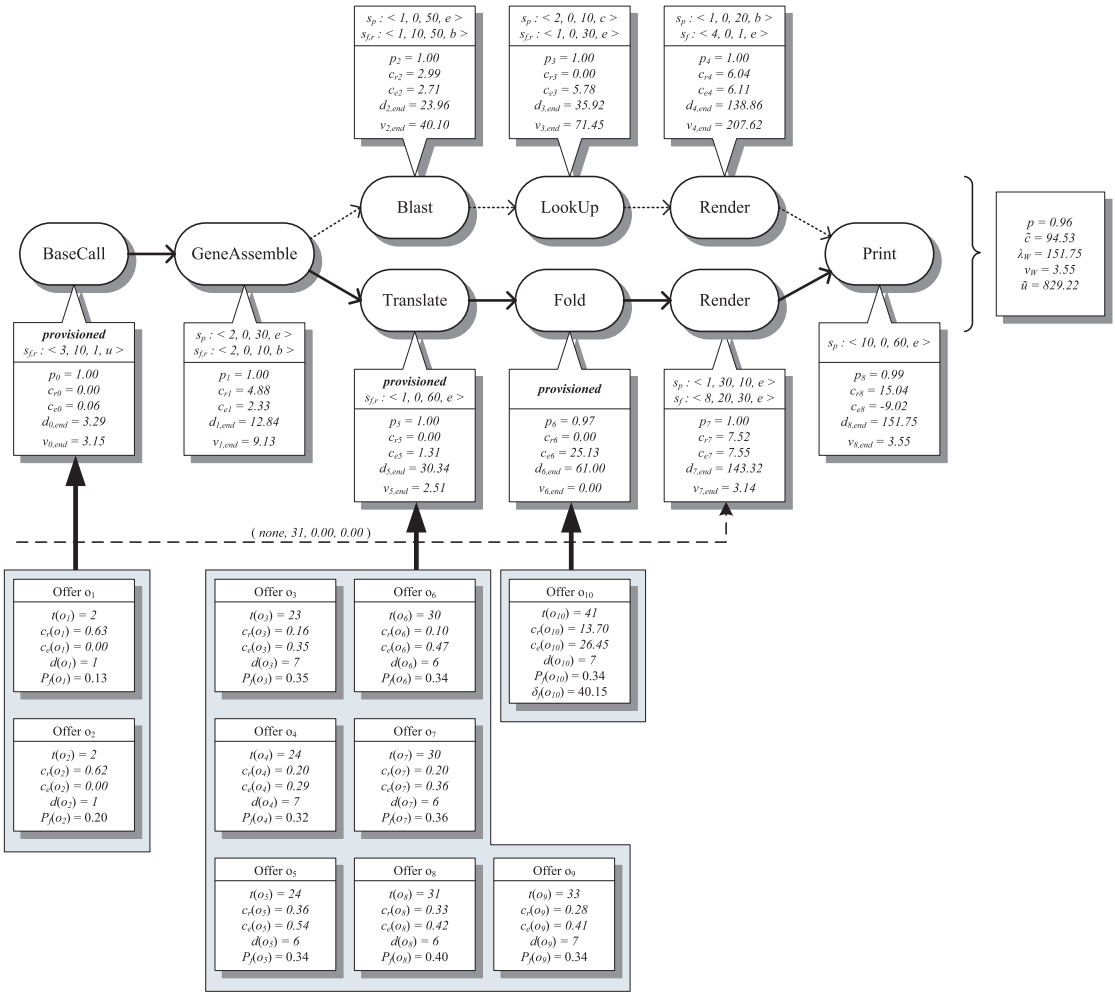


FIGURE 6.9: Urgent workflow after first provisioning.

the distributions in each row of Table 6.7. First, we generate the number of offers by drawing a sample from a Poisson distribution with a mean given by the birth rate in that row (we use $b = d = 0.005$, unless noted otherwise). For each such offer, we then assign it the service type given in the table and draw a value for the reservation cost, execution cost and service duration from the specified distributions[10]. All other offer parameters, such as the failure probability and penalties, are determined according to our experimental parameters detailed in later sections.

	Type	Reserv. Cost	Exec. Cost	Time	Birth Rate	Death Rate
1	T_1	$\mathcal{U}_h(25)$	$\mathcal{U}_h(25)$	$\mathcal{U}_h(5)$	$b/2$	$d/2$
2	T_1	$\mathcal{U}_h(5)$	$\mathcal{U}_h(5)$	$\mathcal{U}_h(40)$	$b/2$	$d/2$
3	T_2	$\mathcal{U}_h(1)$	$\mathcal{U}_h(5)$	$\mathcal{U}_h(50)$	b	d
4	T_3	$\mathcal{U}_h(10)$	$\mathcal{U}_h(10)$	$\mathcal{U}_h(35)$	b	d
5	T_4	$\mathcal{U}_h(50)$	$\mathcal{U}_h(1)$	$\mathcal{U}_h(25)$	b	d
6	T_5	$\mathcal{U}_h(1)$	$\mathcal{U}_h(50)$	$\mathcal{U}_h(25)$	b	d

TABLE 6.7: Service type parameters.

In our simulations, a consumer is rewarded a maximum utility of $u_{\max} = 2000$ for completing a workflow, with penalty $\delta = 40$ and deadline $t_{\mathrm{d}} = 200$. Each workflow consists of 8 tasks (with types chosen randomly from $\mathcal{T}$) and we generate them randomly as in previous chapters, with a parallelism of 0.25.

We chose these parameters to represent a realistic and challenging scenario with a relatively short deadline, but a sufficient maximum utility to allow the agent to afford a number of failed service invocations in uncertain environments. The workflows we test here are small, because related work that relies on integer programming techniques was unable to deal with larger cases.

6.3.2 Strategies

In our experiments, we evaluate the performance of four strategies. The first three are based closely on the work presented in Zeng et al. (2004), and are broadly similar to those described in the previous chapter (Section 5.2). The fourth is the *dynamic flexible* provisioning strategy proposed in this chapter. We briefly describe each below.

6.3.2.1 Local Strategy

This strategy is similar to the *adaptive local* strategies described in Section 5.2.1. Again, it selects an offer to maximise a weighted sum of performance parameters. However, as services are not always immediately available, we include a lookahead parameter l, which determines a window of future time steps that the strategy will consider when selecting the best available

[10] We use $\mathcal{U}_h(m)$ to refer to a uniform distribution with mean m that varies around m by a proportion of at most h, i.e., $\mathcal{U}_h(m)$ is a uniform distribution on the interval $[(1-h)\cdot m, (1+h)\cdot m]$. We use $h = 0.2$ in all our experiments, indicating a fairly high heterogeneity of offers.

offer (we set $l = 20$ in our experiments as this produces good results for the environments we consider). Also, in line with the modified system model, we now use $q_1(o) = c_e(o) + c_r(o)$ (the combined total cost), $q_2(o) = 1 - P_s(o)$ (the success probability) and $q_3(o) = t(o) + d(o)$ (the end time of the offer) as the performance parameters to consider.

As the completion time is an explicit part of each service contract, we do not need to include time-out values. Instead, the strategy re-provisions offers immediately when they are still unsuccessful after the promised duration. Furthermore, we do not currently consider parallel redundancy for this strategy, as the existing approaches we base this upon do not employ this technique (and unlike our work in the previous chapter, there is no immediately obvious technique for including parallel redundancy, as different offers may not start at the same time step or even overlap at all).

6.3.2.2 Global Weighted Optimisation

This is mostly identical to the non-adaptive global strategy described in Section 5.2.2 (using the updated performance parameters).

6.3.2.3 Adaptive Global Weighted Optimisation

This strategy behaves as the above strategy, but also re-provisions offers once they have failed. In so doing, it takes into account the money spent on offers to that point and the time that has passed, and it decides whether to keep any already provisioned offers or re-provision those too (taking into account the additional reservation cost required).

6.3.2.4 Flexible Provisioning

This is the *dynamic flexible* provisioning approach as presented in this chapter. For this strategy, we build a task strategy library by taking 2000 independent observations of the market over time and recording the predicted outcomes of each of a set of possible strategies, which we generate by considering the combinations of the advance times $t_a(\omega) \in \{0, 10, 20, 30, \ldots, 250\}$, the provisioning intervals $t_w \in \{1, 10, 20, 30, \ldots, 100\}$, the number of parallel providers $n(\omega) \in \{1, 2, 3, \ldots, 10\}$ and all selection strategies. Considering that each strategy may be repeated, this means there are 22880 possible high-level strategies for each service type. Due to the time required to build the library, we do this once for every environment in this section and then re-use the same library when repeating our experiments (we have verified that there is no significant difference in our results when using different libraries).

6.3.3 Hypotheses

In this chapter, we are interested in four hypotheses. The first two consider environments where providers fail maliciously without paying compensation, the next one considers cases where providers offer full refunds for failures, and the final hypothesis looks at environments where providers offer better services when provisioned with varying advance notice periods (e.g., where there are discounts for either early or late provisioning).

Hypothesis 12. In environments where the performance of services does not depend on the time of provisioning and where they fail maliciously (i.e., do not pay any penalties), the ***flexible*** strategy results in a higher profit than any of the other examined strategies, averaged over all cases.

Hypothesis 13. In the above environments, the ***flexible*** strategy successfully completes a higher proportion of workflows than any of the other examined strategies, averaged over all cases.

Hypothesis 14. Hypotheses 12 and 13 also hold when services offer full refunds for failures.

Hypothesis 15. Hypotheses 12 and 13 also hold in environments where the performance of services is dependent on the time of provisioning.

In the following, we discuss the results of our experiments. More specifically, we examine Hypotheses 12 and 13 in Section 6.3.4, then we discuss Hypothesis 14 in Section 6.3.5, and finally look at Hypothesis 15 in Section 6.3.6. Where appropriate, we have carried out ANOVA, followed by pairwise t-tests to ascertain the statistical significance of the results (at the $p = 0.005$ level) and we give 95% confidence intervals for all data.

6.3.4 Malicious Providers (Hypotheses 12 and 13)

During our first set of experiments, we evaluated the performance of the four strategies in environments where service providers are increasingly unreliable. To this end, we varied an overall average defection probability $\bar{d}$ across several experiments and used this to generate the defection probability of offers[11]. We also assume that services either succeed or defect (and so any penalties are irrelevant). This case is challenging for consumers, because they do not get compensation for failures, but it is realistic in highly dynamic distributed systems, where some providers may act maliciously and never perform the service they were paid to do. Examples of such systems include peer-to-peer systems, where providers may frequently leave the system and where it is difficult to enforce contracts.

The results of our experiments are shown in Figure 6.10, which plots the average failure probability of an environment against the average profit (as a proportion of $u_{\max}$) that each strategy gains[12]. To complement this, Figure 6.11 shows the associated proportion of workflows that the

[11] Again, we draw from a distribution $U_h(\bar{d})$, where $h = min(0.2, h')$ and h' is the largest real number with $(1 - h') \cdot \bar{d} \geq 0 \wedge (1 + h') \cdot \bar{d} \leq 1$.

[12] We average the profit over 750 runs for the flexible and the local approaches, while we average it over 250 runs for the global optimisation approaches due to their more time-intensive nature.

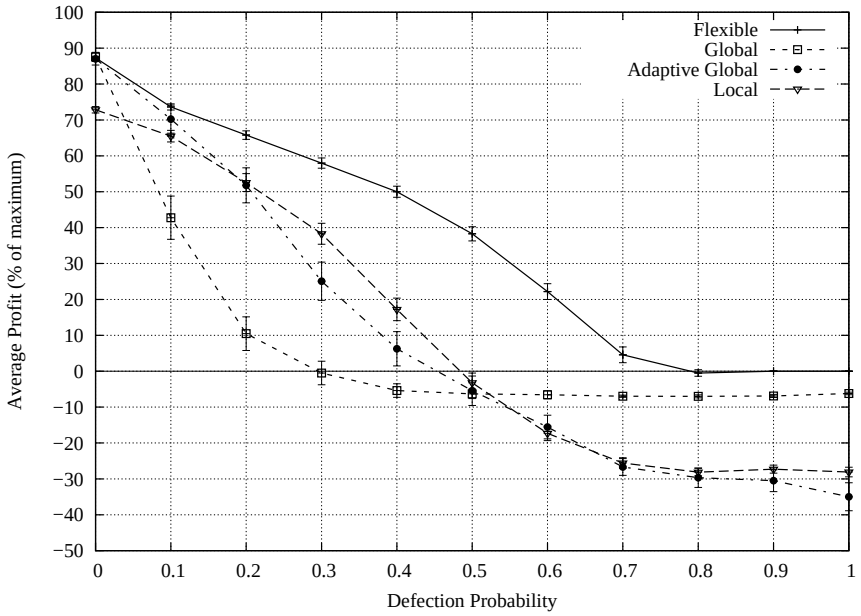


FIGURE 6.10: Performance of strategies in environments where providers increasingly defect.

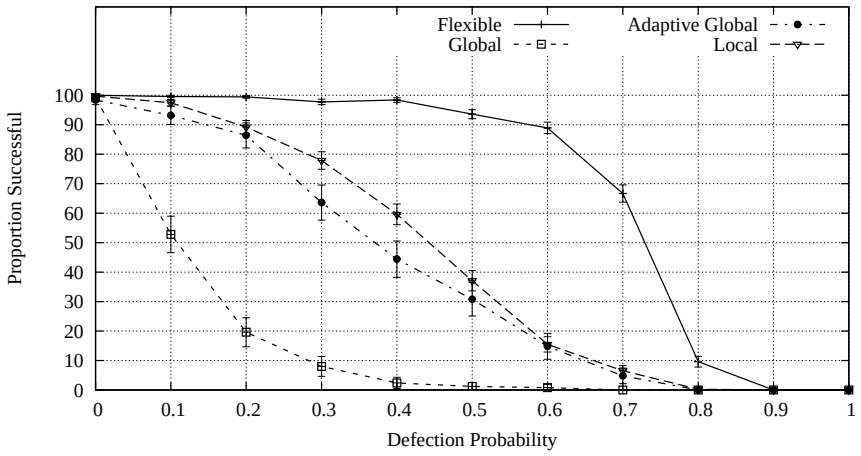


FIGURE 6.11: Proportion of successful workflows in environments where providers increasingly defect.

strategy managed to complete with a positive payoff. When providers never defect ($\bar{d} = 0$), all strategies perform well, achieving between 70–90% of the maximum reward, and there is no significant difference between either of the global optimisation approaches and the flexible strategy. Intuitively, both global strategies are equivalent here, because there is no need to re-provision failed tasks, and they both perform well due to the certain information they have about the cost and duration of the complete workflow. The flexible strategy similarly performs well — although it does not provision the complete workflow in advance, it makes accurate predictions at the start (with little uncertainty) and provisions services as it proceeds through the workflow. The local optimisation approach performs worse than the other strategies, as it takes myopic decisions and therefore occasionally exceeds t_{max} or even t_{zero}.

As $\bar{d}$ increases, all strategies generally perform worse, because they increasingly have to pay

for services that do not perform as promised. The non-adaptive global optimisation strategy is most affected as $\bar{d}$ begins to rise, due to it only attempting one execution of the workflow before giving up. If it succeeds, it gains a relatively high reward, but if it fails, it loses its initial investment. Hence, the performance trend follows closely the average success probability of a single execution, i.e., the probability that all eight workflow tasks succeed: $(1 - \bar{d})^8$. For example, when $\bar{d} = 0.1$, we expect the average success probability to be around $0.9^8 = 0.43$, while at $\bar{d} = 0.2$, it drops to $0.8^8 = 0.17$, and this is reflected closely by the shape of the graph. At $\bar{d} = 0.3$ and beyond, the strategy no longer makes a profit, as it begins to fail most workflows and consistently lose its investments.

In contrast to this, the adaptive optimisation strategy performs considerably better than the non-adaptive one as the defection probability begins to rise, up to $\bar{d} = 0.4$. On this interval, failures occur occasionally and the adaptive consumer is generally able to re-provision the workflow to meet its deadline. However, at $\bar{d} = 0.5$, failures become too numerous (the consumer now fails to complete 69.0% of its workflows before t_{zero}) and the consumer begins to make an overall loss. As the defection probability rises further, this loss increases, eventually levelling off towards $\bar{d} = 1.0$. This considerable loss occurs because the consumer lacks the capability of predicting the overall cost it will incur by re-provisioning and whether this investment is rational, given the defection probabilities of services. Rather, it will persist in retrying more services and making further investments, despite a high probability of failure (at $\bar{d} = 0.8$ and beyond, the consumer completes no workflows successfully).

Next, the average profit of the local strategy initially drops less quickly than the global strategies. This occurs because it is less affected by a small a number of failures than the global approach, which may need to re-provision its workflow completely upon a single failure. In some environments, when the defection probability is $\bar{d} = 0.2$ and $\bar{d} = 0.3$, it even outperforms the adaptive global approach for that reason. Beyond that, it drops more quickly and follows a broadly similar trend to the adaptive global strategy, as it also invests heavily in services without ever completing the workflow.

It is interesting to note here that none of the non-flexible approaches consistently outperforms the others. When service outcomes are certain, the global approaches outperform the local strategy, but as the defection probability exceeds 0.2, the local approach begins to dominate. Beyond $\bar{d} = 0.5$, the non-adaptive global approach dominates, but only because it makes the smallest loss. Also, none of the non-flexible strategies are able to deal effectively with environments where the defection probability is 0.5 or higher. Specifically, at $\bar{d} = 0.5$, they all make a loss and complete less than 40% of their workflows before t_{zero}. At $\bar{d} = 0.6$, this drops to 20%.

Finally, we consider the performance of the flexible strategy. At low defection probabilities, it achieves a similar performance as the global approaches. However, at $\bar{d} = 0.2$, it begins to clearly dominate all other strategies. Unlike the other strategies, it reasons explicitly about failures and their impact on the workflow cost and execution time, and so at these higher failure probabilities, the flexible strategy is able to deal proactively with failures, for example by

provisioning them redundantly or by favouring more reliable providers. In more detail, this means that the flexible approach is able to achieve an almost 200% improvement over the best-performing non-flexible strategy at $\bar{d} = 0.4$ and it still makes a positive profit at $\bar{d} = 0.5$, $\bar{d} = 0.6$ and $\bar{d} = 0.7$, when all other strategies make a loss (in fact, the flexible strategy successfully completes over 98%, 93%, 88% and 66% of its workflows before t_{zero} in these environments, respectively).

At $\bar{d} = 0.8$, we notice that the flexible strategy makes a small net loss of -9.97 ± 18.25. However, this is clearly not a significant loss in this case. Averaged over all values for $\bar{d}$ we tested, the flexible approach achieves a profit of 735.83 ± 15.86, while the non-adaptive and adaptive global approaches achieve only 173.80 ± 26.85 and 183.60 ± 37.83, respectively. The local strategy achieves an average profit of 212.30 ± 20.58. Similarly, the flexible strategy successfully completes $68.55 \pm 0.51\%$ of workflows, while the remaining strategies complete only $16.87 \pm 1.39\%$, $40.21 \pm 1.85\%$ and $43.93 \pm 1.07\%$, respectively. These results support[13] Hypotheses 12 and 13.

6.3.5 Failures with Refunds (Hypothesis 14)

In our next experiments, we are interested in environments where providers are not malicious, but offer full refunds to the consumer in case of failure. Hence, the setup is similar to the previous sub-section, but we now assume that when providers fail, they immediately refund both the reservation and the execution cost of the service. This is a more realistic scenario when services are offered by reputable companies, when some central entity monitors the system or when contracts are easily enforceable. Examples of such systems may include Web services or scientific Grids.

The results are shown in Figures 6.12 (average profit) and 6.13 (proportion of successful workflows) and clearly highlight mostly the same trends as in the previous experiments for the non-flexible strategies (all achieve slightly higher profits and tolerate higher failure probabilities). The local strategy now performs better than before as it will pay at most once for each task in the workflow, and it even achieves a small positive average profit when the failure probability is $\bar{f} = 0.6$.

The flexible strategy performs significantly better in this environment, achieving a high positive profit even at failure probabilities of up to $\bar{f} = 0.9$. More specifically, at $\bar{f} = 0.6$, our strategy achieves an average profit of 1071.22, with 96.5% of workflows executed successfully before t_{zero}, compared to the best non-flexible profit of only 34.34 with 19.1% of workflows successful (an approximately 35-fold improvement in average utility). At $\bar{f} = 0.8$, the flexible approach still completes 86.1% of workflows successfully, while the most successful non-flexible strategy

[13] An ANOVA of the profits, averaged over all environments, rejects H_0 that they are equal ($F = 400.16$ and $p < 0.001$). Pairwise t-tests confirm that the flexible strategy is significantly better than any of the others (all with $p < 0.001$). To test Hypothesis 13, we compared each proportion of successful workflows using Fisher's exact test, confirming that the flexible strategy is significantly more successful than any others (all with $p < 0.001$).

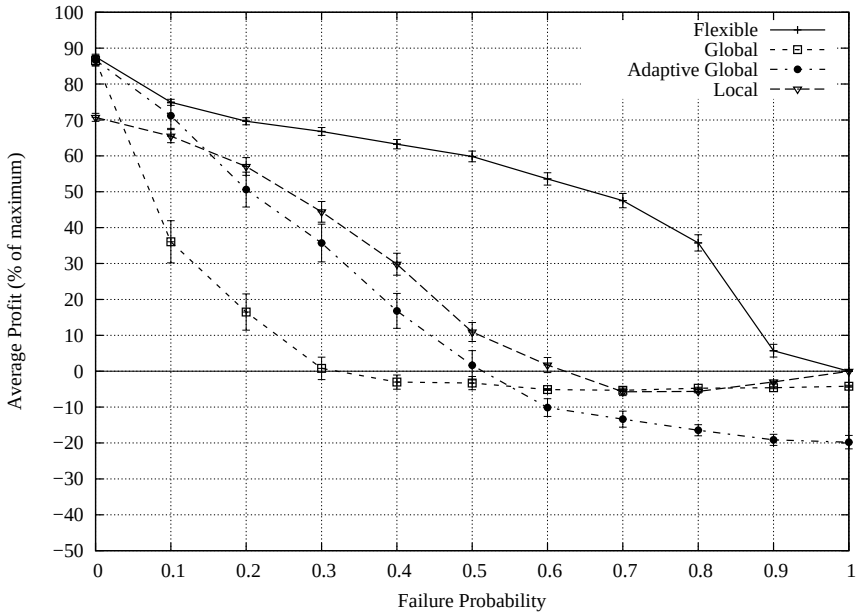


FIGURE 6.12: Performance of strategies in environments where providers give refunds.

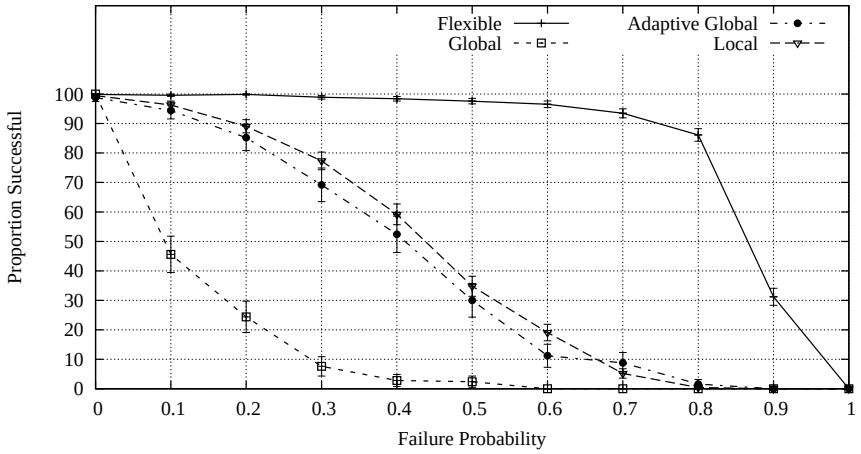


FIGURE 6.13: Proportion of successful workflows in environments where providers give refunds.

completes 1.6%. This good performance is due to the considerably lower cost of invoking services redundantly, as now the consumer effectively pays for only those services that succeeded rather than all invoked services. Even at $\bar{f} = 0.9$, the flexible approach still achieves a positive profit of 114.35 and completes 31.2% of workflows successfully.

For all values for $\bar{f}$ tested, the flexible approach achieves an average profit of 1026.53 ± 14.23, the global approaches achieve 200.60 ± 26.07 (non-adaptive) and 335.80 ± 34.30 (adaptive), while the local approach achieves 473.68 ± 17.13. The respective proportions of successful workflows are $81.96 \pm 0.42\%$, $16.74 \pm 1.38\%$, $41.57 \pm 1.86\%$ and $43.30 \pm 1.07\%$, which supports[14] Hypothesis 14.

[14] An ANOVA of the profits, averaged over all environments, rejects H_0 that they are equal ($F = 873.93$ and $p < 0.001$). Pairwise t-tests confirm that the flexible strategy is significantly better than any of the others (all with

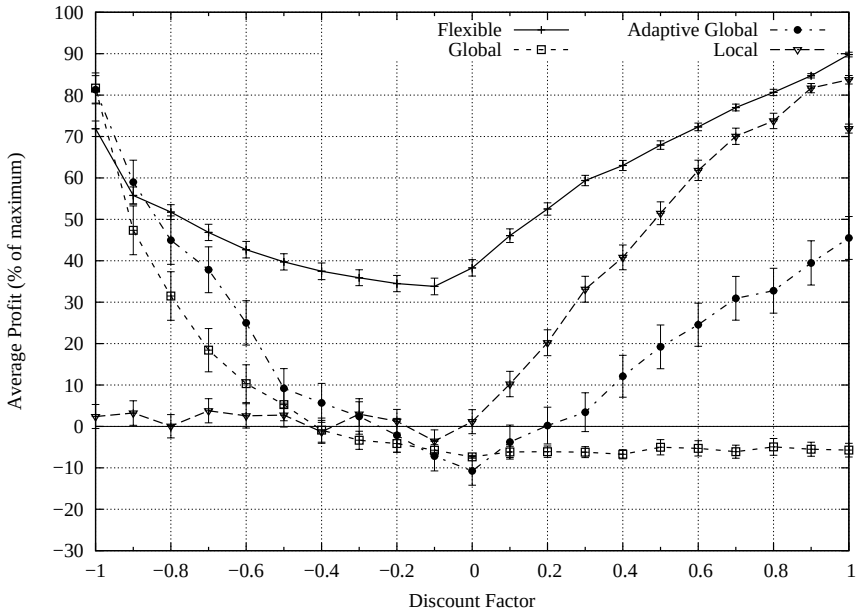


FIGURE 6.14: Performance of strategies when advance provisioning is preferred (negative adjustment) and when on demand is preferred (positive adjustment).

6.3.6 Different Market Conditions (Hypothesis 15)

Next, we tested the performance of the strategies in environments where either advance or on demand provisioning is preferred and given a discount in execution cost and a higher reliability. Such conditions might occur, respectively, when providers prefer to be given early notice by consumers, so that they can plan their resource availability in advance, or when they find their resources under-utilised and therefore offer discounted services at the last minute. To express this preference, we vary a discount factor, d, from -1 to 1. When negative, this indicates a preference for early (advance) provisioning and when positive, on demand provisioning is preferred. In more detail, we use it during offer generation to adjust the distribution means for the execution cost and failure probability by a proportion given by $|d|$. We consider all offers generated for the current time step, $\hat{t}_i$, as provisioned on demand, and any offers generated for $\hat{t}_{i+40}$ and beyond as provisioned in advance. Between these two, we vary the discount factor linearly. For example, when $d = -0.6$, $\bar{f} = 0.5$ and we generate an offer for t_{i+30}, then the corresponding mean failure probability is $(1-3/4 \cdot 0.6) \cdot 0.5 = 0.275$. We use all other experimental parameters as in our first experimental setup, but keep $\bar{f}$ at 0.5, and now set $b = 0.5$ and $d = 5$, to ensure that discounted offers are available only at their respective time steps.

Figure 6.14 shows the average profit of the strategies in these environments, while Figure 6.15 shows the proportion of successfully completed workflows. Here, we note that the non-flexible strategies perform well only in extreme conditions — the global approaches excel when advance provisioning is preferred, while the local strategy performs well as d tends to 1. When neither advance nor on demand provisioning is strongly preferred, none of the non-flexible strategies

$p < 0.001$). Next, we compared each proportion of successful workflows using Fisher's exact test, confirming that the flexible strategy is significantly more successful than any others (all with $p < 0.001$).

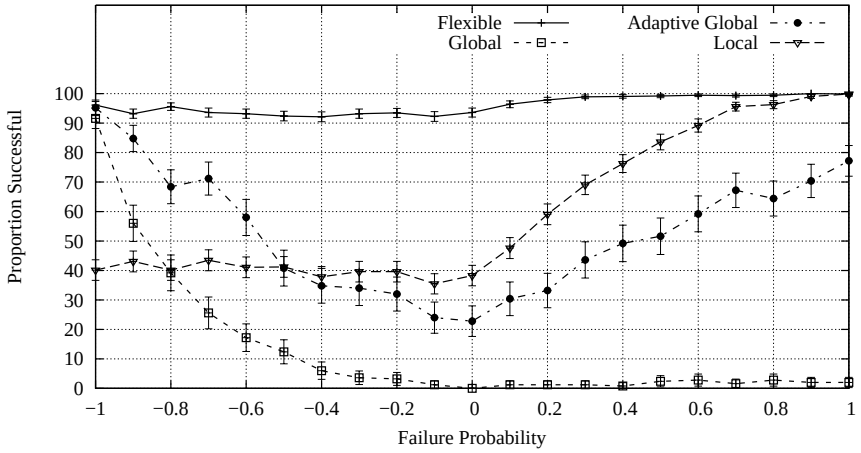


FIGURE 6.15: Proportion of successful workflows when advance provisioning is preferred (negative adjustment) and when on demand is preferred (positive adjustment).

does well, because most services in the market are unreliable. In fact, at $d = -0.1$, these strategies all make a net loss. In contrast to this, the flexible strategy manages to achieve a high profit over all environments, and, in most cases, significantly outperforms all other strategies. This is because the flexible strategy adjusts its provisioning strategies to the environment — at $d = -1$, it provisions services, on average, 43.05 ± 0.72 time steps in advance, at $d = 0$, this drops to 14.71 ± 0.36 and at $d = 1$, it provisions only 3.57 ± 0.12 time steps ahead. However, we also note that the flexible strategy is now outperformed at $d = -1$ (at $d = -0.9$, there is no significant difference). In this cases, it suffers from not provisioning all offers in advance (and thereby producing a tight-fitting but reliable schedule). Instead, the strategy continues to provision only parts of the workflow (although now provisioning further ahead) and hence sometimes exceeds $t_{\max}$. Nevertheless, when averaging over all values for d considered here, the flexible strategy achieves an average utility of 1143.61 ± 12.12, while the global approaches achieve only 109.69 ± 17.78 (non-adaptive) and 428.40 ± 24.70 (adaptive), and the local approach achieves 516.37 ± 15.17. The corresponding proportions of successfully completed workflows are $95.83 \pm 0.22\%$, $13.11 \pm 0.90\%$, $53.20 \pm 1.36\%$ and $59.78 \pm 0.77\%$, which supports[15] Hypothesis 15.

To summarise our empirical evaluation, Table 6.8 shows the average utility each strategy gained in the various environments discussed in this chapter. It is clear that the flexible strategy outperforms all other strategies we tested here. In the following section, we briefly show that these trends also hold for larger, more complex workflows.

[15]Again, an ANOVA rejects H_0 that all mean profits are equal ($F = 1825.08$ and $p < 0.001$). Pairwise t-tests confirm that the flexible strategy outperforms all others (all with $p < 0.001$), and Fisher's exact test confirms that the flexible strategy is more successful than the others (all with $p < 0.001$).

Strategy	Environment	Utility	Success %
flexible	malicious	725.83 ± 15.86	68.55 ± 0.51
global	malicious	173.80 ± 26.85	16.87 ± 1.39
adaptive global	malicious	183.60 ± 37.83	40.21 ± 1.85
local	malicious	212.30 ± 20.58	43.93 ± 1.07
flexible	refunds	1026.53 ± 14.23	81.96 ± 0.42
global	refunds	200.60 ± 26.07	16.74 ± 1.38
adaptive global	refunds	335.80 ± 34.30	41.57 ± 1.86
local	refunds	473.68 ± 17.13	43.30 ± 1.07
flexible	discounts	1143.61 ± 12.12	95.83 ± 0.22
global	discounts	109.69 ± 17.78	13.11 ± 0.90
adaptive global	discounts	428.40 ± 24.70	53.20 ± 1.36
local	discounts	516.37 ± 15.17	59.78 ± 0.77

TABLE 6.8: Summary of empirical results.

6.4 Performance in Complex Environments

In this section, we consider large workflows with 50 tasks, a parallelism of 0.25, $u_{\max} = 6000$, deadline $t_{\max} = 500$ and penalty $\delta = 50$. We assume that tasks belong to one of ten types, and we generate offers randomly in a similar manner as described in Section 6.2.9, but now use the characteristics given in Table 6.10 and the corresponding distributions[16] in Table 6.9. Table 6.10 now includes two rows for each service type, in order to generate more varied offers (e.g., early offers generated by the first row are generally cheaper but also take longer and are less reliable than those generated by the second row). Furthermore, these service populations have been chosen to represent a setting where some services are better when provisioned in advance (such as the first row for service type 2), others are better when provisioned on demand (the second row for service type 5), but most offer various trade-offs between the different qualities when provisioned earlier or later. For all types, we assume that they can be provisioned up to 300 time steps in advance.

	Costs	Duration	Reliability	Availability (Birth/Death Rates)
low	$\mathcal{U}_c(1,3)$	$\mathcal{U}_c(1,5)$	$\mathcal{U}_c(f,1.5f)$	0.05 / 1
medium	$\mathcal{U}_c(3,10)$	$\mathcal{U}_c(5,20)$	$\mathcal{U}_c(0.8f,1.2f)$	0.5 / 0.5
high	$\mathcal{U}_c(10,25)$	$\mathcal{U}_c(20,60)$	$\mathcal{U}_c(0.5f,f)$	2 / 2
v.high	$\mathcal{U}_c(25,100)$	$\mathcal{U}_c(60,240)$	$\mathcal{U}_c(0,0)$	—

TABLE 6.9: Distributions used in Table 6.10.

Figure 6.16 shows the performance of the *dynamic flexible* and the *local* strategies (it was impossible to provide results for either of the two *global* strategies, as they were unable to deal with the larger number of offers and tasks in these settings[17]). The figure shows that the *flexible* strategy

[16] The variable f used to define the reliability distributions is the average failure probability in a given setting. Where necessary, we assume that upper bounds are adjusted to be at most 1.

[17] We attempted to solve the associated integer programming problem, but CPLEX ran out of its allocated memory (1.5 GB) after two hours.

Type	Time		Reserv. Cost		Exec. Cost		Duration		Availability		Reliability		Refunds
	early	late	early	late	early	late	early	late	early	late	early	late	
1	≥ 20	≤ 5	low	low	low	medium	medium	high	high	low	high	medium	no
1	≥ 30	≤ 10	high	medium	medium	medium	low	high	medium	medium	v.high	medium	yes
2	≥ 5	≤ 0	low	medium	low	medium	low	medium	high	low	high	low	yes
2	≥ 40	≤ 10	high	medium	high	low	medium	medium	high	low	medium	medium	no
3	≥ 100	≤ 0	low	v.high	low	high	low	high	low	low	v.high	low	no
3	≥ 1	≤ 0	high	high	high	high	medium	medium	medium	medium	medium	medium	no
4	≥ 150	≤ 0	high	low	low	v.high	low	medium	medium	medium	v.high	low	yes
4	≥ 150	≤ 0	low	high	v.high	low	medium	low	medium	medium	low	medium	yes
5	≥ 60	≤ 0	medium	low	medium	low	low	medium	medium	low	medium	medium	no
5	≥ 40	≤ 10	high	medium	high	medium	low	low	low	low	high	high	no
6	≥ 50	≤ 5	high	low	low	high	low	v.high	low	medium	medium	low	no
6	≥ 1	≤ 0	low	low	high	high	high	high	medium	medium	high	high	yes
7	≥ 70	≤ 30	medium	low	low	low	low	high	medium	medium	high	low	no
7	≥ 30	≤ 10	medium	medium	low	high	high	low	high	low	medium	high	yes
8	≥ 30	≤ 0	high	low	medium	low	high	medium	medium	high	high	low	yes
8	≥ 100	≤ 50	low	high	low	high	low	low	medium	low	v.high	low	no
9	≥ 50	≤ 10	medium	high	medium	low	medium	high	medium	high	medium	high	no
9	≥ 30	≤ 0	high	medium	low	low	medium	low	high	low	medium	medium	no
10	≥ 200	≤ 50	low	medium	low	high	medium	low	high	medium	high	medium	yes
10	≥ 200	≤ 50	medium	medium	low	v.high	low	medium	medium	medium	medium	high	yes

TABLE 6.10: Service types used to evaluate complex environments.

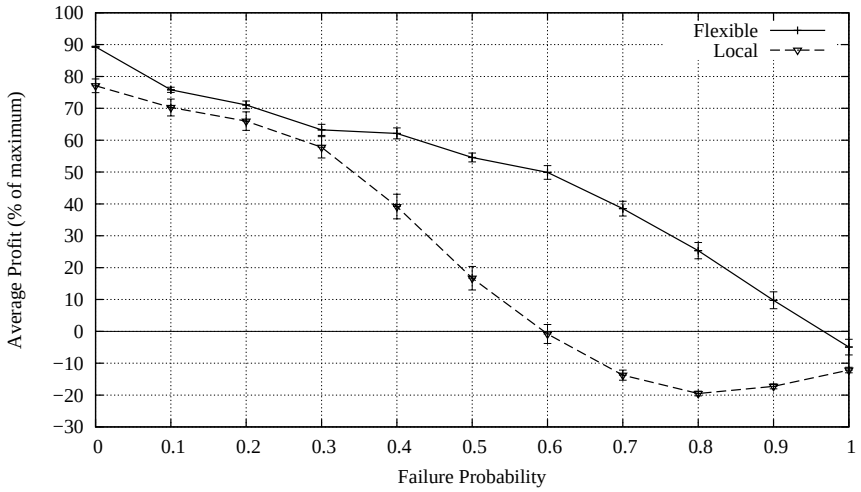


FIGURE 6.16: Performance of strategies in more complex environments.

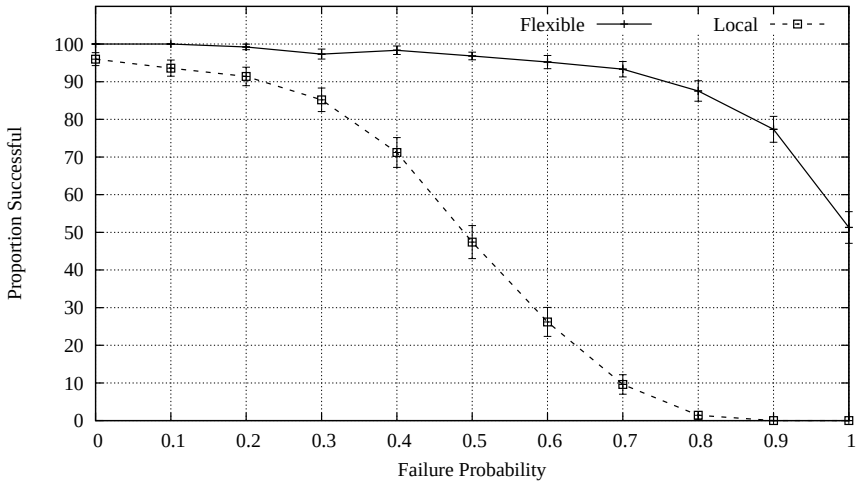


FIGURE 6.17: Proportion of successful workflows in more complex environments.

again achieves a high average profit over most environments, even when the failure probability is high. It also performs significantly better than the *local* strategy over all environments considered here.

However, we also note that the *flexible* strategy makes a small loss when the failure probability is 1. Here, it still attempts some workflows (as there are usually some offers with a non-zero success probability when provisioned at a specific time), but then often takes slightly longer to complete than anticipated. We believe that this due to our heuristic workflow duration estimation technique, which is inherently optimistic. Furthermore, in contrast to the strategies presented in previous chapters, we also use this heuristic to predict the probability that tasks will conflict with each other. As discussed in Section 6.2.4, this introduces further inaccuracies, which we believe contributes to the overall loss incurred in this particular environment. Despite this, the general trends of the strategy are promising and it still completes most workflows successfully even when the failure probability is high (as shown in Figure 6.17).

Having discussed the experimental results of the *dynamic flexible* strategy, we now summarise the work presented in this chapter.

6.5 Summary

In this chapter, we considered environments with dynamic service markets, where the prices, availability and other parameters of services change over time, and where the consumer enters explicit contracts with providers. To deal with such environments, we use high-level provisioning strategies that are based on statistical market observations, and only gradually provision workflows, in order to retain flexibility and deal with failures. By addressing this type of system model, we covered the remaining model requirements: M.2.b, M.3.b and M.5. Furthermore, as the strategy iteratively provisions the workflow over time, using new information about failures and service availability as it is observed, the extended flexible strategy of this chapter refines its decisions adaptively, thus addressing Requirement A.4.

By carrying out an experimental study, we have shown that our strategy performs significantly better than the current state of the art, and over a range of environments. Furthermore, we demonstrated that the strategy adapts well to prevailing market conditions, provisioning services earlier when providers offer discounts for advance provisioning and leaving provisioning to the last moment, when this is favoured by the providers.

Chapter 7

Conclusions and Future Work

This final chapter concludes the thesis by reviewing its contributions to the field of service provisioning and by outlining opportunities for future work. To this end, in Section 7.1, we look back at the research problem that has motivated this thesis and provide a high-level overview of the techniques we have proposed to address it. Then, in Section 7.2, we discuss in more detail our research contributions and relate these back to our original requirements from Section 1.4. In Section 7.3, we compare and contrast the flexible strategies developed throughout this thesis. Finally, in Section 7.4, we propose several ways in which our work can be extended in the future.

7.1 Research Summary

Today's computer systems are increasingly distributed and inter-connected in nature, thus allowing organisations to share expensive computational resources and to sell a wide range of services online — from running complex data processing tasks, providing credit checks and travel reservations, to selling physical goods. In this context, service-oriented computing is emerging as a powerful methodology for allowing heterogeneous and distributed software applications to discover and interact with each other automatically. Clearly, employing such a flexible systems engineering approach promises tremendous benefits, as users can automate their business processes and workflows, dynamically outsourcing complex services to those providers that best suit their needs.

However, as we have argued in this thesis, it is necessary to view participants in open service-oriented systems as autonomous agents that have their own goals and objectives. Thus, they cannot be assumed to follow service requests blindly, adhere to their advertised functionality, or even honour pre-negotiated contracts. This uncertainty poses considerable challenges to consumers that rely on external service providers to meet their own objectives, and it is a particularly pressing concern in scenarios where consumers execute large workflows, have strict deadlines, and where providers demand remuneration for their services.

Reviewing the current literature on service provisioning, we found that existing approaches do not address this uncertainty in a satisfactory manner. Most view service failures or contract violations as exceptional and rare events that are handled purely reactively either by manually specified error handling procedures, or by re-provisioning the failed task. Other approaches impose strict constraints to ensure that only highly reliable services are provisioned, but this is infeasible when all providers are unreliable or when workflows consist of hundreds of tasks. Some existing work employs redundancy to deal proactively with highly unreliable providers, but this is typically done in an ad hoc manner and does not explicitly balance the cost of introducing this redundancy with its benefit.

Against this background, we examined work in the field of multi-agent systems and identified a number of techniques that we believe are vital for addressing the above shortcomings. In particular, we built on work on trust and reputation to model the uncertain behaviour of service providers using probability theory, and we placed the interactions of consumers and providers within the context of a service market, where providers are financially remunerated for their services (and possibly allow consumers to reserve resources in advance). Based on this, we developed a decision-theoretic approach that enables a consumer agent to take appropriate decisions on behalf of its user with minimal human intervention. In particular, this approach reasons explicitly about the uncertain behaviour of service providers to decide how many services to provision for each task in a workflow, which ones to choose from a set of heterogeneous services, how to deal with services that do not return explicit failure messages and also when to start negotiating service contracts in advance.

In the following section, we provide a more detailed summary of our approach, highlighting the novel contributions we have made to the state of the art and relating our work back to our original requirements from Section 1.4.

7.2 Research Contributions

In this thesis, we set out to design a set of methods for building a computational agent that is capable of executing complex workflows in highly dynamic and uncertain service-oriented environments. We achieved this by adopting decision theory as a principled framework that not only allows the service consumer agent to make decisions autonomously on a user's behalf, but that also builds naturally on top of work in the area of trust and reputation that models the behaviour of service-providing agents probabilistically. Employing this framework, we made a number of significant contributions that allow a software agent to execute its workflows effectively even in environments where service providers are highly unreliable. In the following sections, we outline each of the main contributions of this thesis (Sections 7.2.1–7.2.5) before summarising how we have addressed our original requirements (Section 7.2.6).

7.2.1 Redundant Provisioning

In this thesis, we proposed the use of redundant services as a fundamental tool for addressing uncertainty. This approach is based on similar techniques in the area of reliability engineering and offers two key advantages: provisioning redundant services in parallel for a particular task allows the consumer to both increase the overall probability of success as well as decrease the expected task duration. However, introducing such redundancy clearly also leads to a higher overall cost, as the consumer may need to pay for all provisioned services.

Previous work has employed redundancy using a static approach, usually by provisioning a fixed number of parallel services for each task. However, our approach is the first that reasons explicitly and fully automatically about the level of redundancy that is appropriate for each workflow task, using a principled decision-theoretic framework. In doing so, we take into consideration several important factors. First, we use the performance characteristics of services, e.g., to use greater redundancy when provisioning particularly unreliable services. Second, we explicitly balance the cost of redundancy with its benefit (e.g., when they are expensive, we rely on fewer services, but when they are cheap, we provision more in parallel). Third, we consider the importance and time-constraints of workflows to decide how much to spend on services and whether redundancy is justified (e.g., when a workflow is of high importance to the consumer, it may be appropriate to use redundancy even when services are reliable). Finally, our approach also takes into account the structure of workflows, e.g., to rely on higher levels of redundancy towards the end of the workflow, in order to ensure that the high investment in earlier services is not lost.

7.2.2 Flexible Re-Provisioning

Current approaches for handling failures in service-oriented systems typically assume that services return timely and truthful error messages to inform consumers when they are unable to provide a requested service. Clearly, such an assumption is unrealistic in open systems, where providers may crash randomly without notice or even defect maliciously after receiving the payment for their services. To address such *crash* failures, previous approaches have relied on manually specified time-out periods after which it is assumed that a service has failed. However, this requires significant human intervention and it is an inflexible approach that does not exploit knowledge about heterogeneous services (some of which may be faster than others) or that can easily be adapted to the structure or time-constraints of a workflow.

To address these shortcomings, we are the first to propose a flexible provisioning mechanism that determines automatically when to stop waiting for apparently failed services and start re-provisioning new services for the task. This mechanism uses probabilistic information about the duration and reliability of services and thereby explicity balances the need to give services sufficient time to complete their task with the possibility that they have already silently failed and will never return a result. Furthermore, it again takes into account a range of factors, similar to those described in the section above. For example, when workflows have strict deadlines, the

consumer may start to wait shorter amounts of time. Similarly, it may allocate longer waiting times to tasks that are less likely to have a significant impact on the duration of the workflow (i.e., those that are not on the critical path).

7.2.3 Limited Information Availability

Related work that has considered unreliable service providers typically assumes that the service consumer has access to detailed performance information about each available service. In contrast to this, we are the first to explicitly consider a spectrum of cases where different amounts of information about services may be available:

- **Full Information:** Complete performance information about each service in the system is available to the consumer. This may be the case where consumers share information about providers using an effective reputation system, or where there is some centralised observer. We believe that such a case is actually rare in open multi-agent systems, as new providers may enter at any time, about whom no specific information is available.
- **Moderate Information:** Specific information about some individual services may be known, but other information is often generalised to larger groups of providers. As an example, this may include open systems, where consumers will have interacted frequently with some providers and so collected accurate information about their services, but where there are also groups of providers with whom few interactions have taken place (e.g., new entrants to the system). In the latter case, the consumer can only use generalised information, perhaps inferred from previous experience with members of such a group.
- **Highly Limited Information:** There is no specific information to distinguish one service provider from another. This may be the case in service-oriented systems that do not offer any form of reliable reputation mechanism and where the consumer's experience with providers is severely limited. Similarly, it is applicable in systems, where the service population changes rapidly and where it is not feasible to track and verify the identities of the providers (such as in a peer-to-peer system). In these systems, the consumer has to rely on information only about the whole population of service providers, possibly based on knowledge about the complexity of the task itself or previous experience with such providers.

In this context, we developed several techniques that exploit the characteristics of these different cases. Specifically, in Chapter 4, we considered the latter case where only limited information about the whole service population is known. Here, we showed that even when all services are highly unreliable, the consumer can use redundancy to proactively influence the expected performance of the workflow to suit its value, structure and time constraints. This is a novel contribution, as existing work on service provisioning relies on differences in the performance

of individual services, in order to select a single service that best meets the given constraints. Furthermore, our approach allows us to efficiently calculate a number of performance parameters, regardless of how many service providers there are.

In Chapter 5, we then considered the first two cases, where more (possibly complete) information about individual heterogeneous services is known. For such cases, we suggested a model that groups the services for a given task into heterogeneous populations, which allowed us to significantly speed up our proposed algorithm.

7.2.4 Gradual Provisioning with Reservations

Existing work on service provisioning has usually made the assumption that services are invoked purely on demand, i.e., that service consumers contact providers only at the time a given service is required. This model is commonly supported by current Web services, and it is an approach we adopted in Chapters 4 and 5. However, there is a growing trend towards advance agreements, in order to provide consumers with higher reliability and some assurance that services will be available when they are needed. While some work on service provisioning uses such advance agreements, their strategies rely on provisioning entire workflows in advance. Clearly, this produces brittle workflow when there is uncertainty, as a single failed service may mean that the consumer misses all subsequent reservations, and this is particularly critical when the consumer has had to make an advance payment for these.

To address these shortcomings, we are the first to propose a more flexible approach to handling advance agreements. Rather than provision an entire workflow at once, our strategy provisions only some tasks in advance and delays the provisioning of other tasks until a later time. This allows the consumer to reduce the risk of missing reserved services, as it can wait until it is more certain about the time the service will actually be needed. Furthermore, our approach naturally allows a mixed system model, where services may be provisioned either on demand or in advance, but with potentially significantly different characteristics (e.g., services provisioned on demand may take much longer and may be more failure-prone than reserved ones, but may also be far cheaper). We believe that such a model may become common in large and open service-oriented systems, but this aspect has not been considered by work on service provisioning so far.

7.2.5 Adaptive Provisioning

While some work on service provisioning has proposed adaptive approaches that monitor the execution of a workflow, these react only to breaches of service agreements or of the overall workflow constraints (and at this time, it might be too late or very costly to recover the workflow). In this thesis, we proposed a novel, more proactive approach to the monitoring and run-time adaptation of workflows. Specifically, by using probabilistic service models, we can

predict when services are likely to cause problems even if no constraints have been breached yet (e.g., we might find that a task is completed slightly later than predicted, resulting in a higher probability that already provisioned services later in the workflow will not be executable). This allows us to react earlier and in a more appropriate manner to potential problems.

In such cases, we then take corrective actions to minimise the disruption to the workflow. These might include re-provisioning tasks of the workflow, but also adding additional redundant services to already provisioned tasks or preparing contingent plans that are only activated when problems occur. Furthermore, our approach will detect infeasible workflows early during execution, as it makes probabilistic predictions and so anticipates failures or deadline violations. In these cases, the consumer may abandon the workflow to avoid wasting resources, while existing work will typically continue to re-provision the workflow until it eventually breaches one of its constraints (e.g., budget or time constraints). Finally, unlike other work, our adaptive mechanism also provides for the case when services perform better than expected. Specifically, our approach may provision less services than originally planned when the workflow is ahead of schedule or when it discovers particularly promising offers at run-time.

Now, having briefly summarised the five principal contributions of this thesis, we return to the research requirements we originally set out to address in Section 1.4, and we discuss how they have been covered by the work presented in this thesis.

7.2.6 Review of Requirements

In the following discussion, we summarise how we have addressed our original model requirements (Section 7.2.6.1), workflow requirements (Section 7.2.6.2) and agent requirements (Section 7.2.6.3).

7.2.6.1 Model Requirements

We begin by discussing how the model we have adopted in this thesis meets our original requirements.

M.1. **Uncertain Service Behaviour**
Building on work in the area of trust and reputation, we decided to model uncertain behaviour using a probabilistic framework. We described this in detail in Chapter 3 and used it as the foundation for our decision-theoretic techniques in all subsequent chapters. More specifically, this has allowed us to express uncertainty along the following dimensions:

a. *Service Success*
Throughout the thesis, we have assumed that providers may fail to provide their services and represented this using a failure probability $f(s_i)$. In Chapter 6, we

extended this model to include cases where the provider fails, but offers a compensation to the consumer.

b. *Service Duration*
Similarly, we modelled uncertain service duration using a probability distribution over possible service completion times ($d(s_i, x)$). In particular, in Chapters 4 and 5, we assumed that the actual duration was not known to the consumer until the service was completed. In Chapter 6, we used a slightly different model and assumed that the duration was an explicit part of the contract (as this is a common term in such contracts).

M.2. **Remuneration for Service Provision**
Services generally require financial remuneration in our model, and this expense has been a central consideration in our decision-making algorithms. In particular, we used both of the following two models:

a. *Fixed Pricing*
In Chapters 4 and 5, we used a fixed pricing model, which applies to systems where providers publish their prices in advance to all customers.

b. *Flexible Pricing*
In Chapter 6, we considered a more flexible approach, where a new quote was produced for each service request, thus resulting in a more uncertain environment.

M.3. **Service Interaction Models**
While most current service-oriented technologies support on demand invocation as the prevalent interaction model, the need for advance agreements is emerging in a number of application areas. For this reason, we decided to cover both in our research:

a. *On Demand Invocation*
In Chapters 4 and 5, we developed techniques for a model that relies solely on on demand invocation.

b. *Advance Provisioning*
In Chapter 6, we considered an extended model, where services may be provisioned in advance, possibly resulting in different performance characteristics. As a special case, this model includes on demand provisioning, when the consumer attempts to reserve a service for immediate use.

M.4. **Provider Heterogeneity**
In Chapters 5 and 6, we explicitly model providers that differ in the quality of their services.

M.5. **Dynamism**
In Chapter 6, we model availability of service offers using a stochastic process, where offers are created and removed from the market according to a fixed birth and death rate,

and where the terms of these contracts are drawn from probability distributions. This leads to a market with considerable dynamism, where the availability of a type of service changes over time and where the qualities are similarly uncertain. However, we currently do not consider systems where the underlying parameters of these processes also change over time.

7.2.6.2 Workflow Requirements

In this section, we summarise how our workflow model meets our original requirements.

W.1. **Workflow Expressivity**
As described in Chapter 3, we model workflows using directed acyclic graphs. This is consistent with much related work in the area and allows us to represent the following:

a. *Parallel Task Ordering*
Two tasks may be executed in parallel when there is no path from one task to the other.

b. *Sequential Task Ordering*
Otherwise, the structure of the graph dictates dependencies between tasks and the sequence they must be executed in.

W.2. **Use of Appropriate Reward Model**
We use a simple utility function u to represent the value of completing a workflow. Importantly, this depends on the time of completion, such that a workflow completed earlier may be more valuable than one that is completed later.

7.2.6.3 Agent Requirements

In this section, we review how we have addressed our original agent requirements.

A.1. **Principled Decision Framework**
We adopted decision theory as a principled framework for building a service consuming agent. As this builds on probability theory, it was a natural choice for dealing with provider uncertainty.

A.2. **Failure Handling**
Our proposed algorithms deal with failures in the following ways:

a. *Reactive Failure Handling*
Throughout this thesis, we consider the appropriate re-provisioning of services when

failures occur. This allows the consumer to recover quickly from failures and reasoning about this explicitly before execution enables the agent to predict the overall utility and feasibility of the workflow. In Chapter 6, we additionally introduce contingent re-provisioning plans for various types of failures that might occur during execution.

b. *Proactive Failure Avoidance*
Another important technique we employ in our work is the redundant provisioning of multiple services in parallel. This proactively addresses failures, as it decreases the probability that a task will not be completed by at least one service.

A.3. **Scalability**
As we rely on heuristic techniques to estimate some probabilities and to find good provisioning allocations, our approach is scalable to large workflows and service-oriented systems, as we have shown in our empirical evaluations. In particular, throughout the thesis, we have verified that our strategies work both on small workflows and on larger instances with 50–100 tasks and thousands of providers (furthermore, in Appendix B, we show that our approach can deal with workflows that consist of thousands of tasks).

A.4. **Adaptivity**
In Chapter 6, we proposed a novel adaptive provisioning approach that deals quickly with unexpected failures, but also exploits new opportunities when services perform better than expected.

Having reviewed our original research requirements, we now discuss how the various flexible provisioning strategies proposed throughout this thesis relate to each other.

7.3 Comparison of Flexible Strategies

In Chapters 4, 5 and 6, we have introduced three flexible provisioning strategies, the *flexible*, *fast flexible* and *dynamic flexible* strategies (we do not discuss the *full flexible* strategy here, as it is broadly similar to the *fast flexible*). As we described in those chapters in detail, they each make different assumptions about the information available about service instances and about how services are provisioned, and so we envisage that each will be applicable to different sets of scenarios. For example, the *flexible* strategy is suitable for environments where no specific information about service providers is available and the *fast flexible* deals with cases where previous observations or a trust model are used to differentiate between heterogeneous providers. Finally, the *dynamic flexible* strategy addresses more dynamic systems, where provisioning agreements are made in advance.

Generally, our approach for devising these three strategies has been similar. Most importantly, they each aim to maximise the service consumer's expected utility and they use the same global

utility estimation technique. However, to best suit the respective system models, the three strategies differ in their local task calculations, the decision spaces they consider and the search algorithms they employ. In the following, we briefly highlight the main distinguishing features of each strategy.

First, the *flexible* strategy outlined in Chapter 4 exploits the limited information available and uses efficient, closed-form equations to calculate local task characteristics. These can be quickly evaluated, even when there are many service providers. Due to these fast calculations and the limited decision space of n_i and w_i for each task t_i, the local search mechanism of the *flexible* strategy is simple — it carries out a steepest-ascent hill-climb until no more improvements can be made (considering neighbours of every workflow task during each iteration).

Next, the *fast flexible* strategy discussed in Chapter 5 considers a more complex problem. In this case, heterogeneous providers may be provisioned in parallel or in sequence, resulting in less efficient local task calculations. Furthermore, the decision space is larger than considered previously, and these factors have prompted us to adopt a faster, greedy hill-climb. This modifies the first task that offers any improvement to the current allocation, terminates after a fixed number of iterations and also carries out random restarts.

Both the *flexible* and the *fast flexible* cover similar system models, and, as explained in Section 5.1, systems with homogeneous providers are subsumed by the model used in Chapter 5. Nevertheless, we believe that the *flexible* strategy is more suitable for such scenarios, because it employs more efficient techniques.

Finally, the *dynamic flexible* strategy addresses a very different system model and also differs significantly in its provisioning approach. Most importantly, the strategy relies on initial high-level provisioning decisions that do not consider concrete service providers, but rather use statistical information about service offers available in the past. This approach was necessitated by the dynamic setting, where the availability of offers changes constantly and where it may be undesirable to provision an entire workflow in advance.

However, such an approach also means that initial provisioning decisions include more uncertainty that is only reduced during execution, when concrete service offers become known. For this reason, the *dynamic flexible* contains an adaptive component that revises the initial provisioning decisions as offers are provisioned at run-time. Finally, the decision space considered by the *dynamic flexible* strategy contains more allocations that result in infeasible workflows (which often lead to a local maximum). Hence, we have had to adopt a stochastic local search algorithm and a modified utility function to specifically avoid such maxima.

In conclusion, none of our proposed strategies is intended to represent a definite solution for all types of service-oriented systems. Rather, they constitute a set of approaches, each of which is best suited for a particular type of environment. Taken together, they cover a wide range of scenarios that are emerging in current service-oriented architectures.

In the following section, we examine a number of promising directions for future research.

7.4 Future Work

The work in this thesis can be extended in a variety of ways. First, some potential application areas may require certain assumptions of our current model to be relaxed. We discuss how this might be done in Section 7.4.1. Second, there are a number of extensions that can be added to our work to make it more applicable to a wider range of scenarios and we detail these in Section 7.4.2.

7.4.1 Addressing Model Assumptions

Throughout this thesis, we have built on a simple, abstract model of a service-oriented system, as outlined in Chapter 3. This allowed us to develop generic techniques that we believe are applicable in a wide range of real world scenarios. However, in devising such a general model, we have had to make a number of potentially limiting assumptions that may not hold in *all* application areas and which we listed in Section 3.6. Here, we return to these assumptions and briefly describe how they may be relaxed in future work.

- **Failure Model:** While we concentrated on silent crash failures in our work (particularly in Chapters 4 and 5), it is easy to extend our model to include explicit failure messages, e.g., by including a new mode of failure, where the provider notifies the consumer of its failure some time after invocation. This would generally reduce the expected duration of tasks as the consumer does not necessarily need to wait for the specified time-out or pre-negotiated service duration, but would not alter our overall strategy.

  Considering Byzantine failures is more challenging, but our approach forms a solid basis for tackling this problem. As we already rely on redundancy, it is possible to include voting schemes that select the majority of several different service outcomes (Lamport et al. (1982); Barborak et al. (1993)). Dealing with correlated failures also poses new challenges, but there are a number of existing techniques for modelling and learning such correlations and for avoiding services that are prone to correlated failures (Nicola and Goyal (1990); Weatherspoon et al. (2002); Townend et al. (2005)). These could be adopted in our work to calculate more accurate, correlated failure probabilities.

- **Performance Information:** The problem of obtaining accurate performance information about unreliable service providers will most likely be addressed by parallel efforts in the areas of trust and reputation. However, in systems where the service consumer relies solely on its own experience rather than a reputation mechanism, it may be interesting to consider the process of gathering such experiences as part of service provisioning. As such, the consumer might explicitly balance the higher certainty in provisioning a known and trusted provider with the potential benefit of provisioning an unknown (but possibly far cheaper) provider. Dealing with such trade-offs between exploration and exploitation are common in many areas of decision-making and could be incorporated into our model.

Furthermore, our model could be extended by considering more complex joint probability distributions that depend on the current time or on other variables. As we described earlier, taking these into consideration might be critical for applications where the quality of services fluctuates over time or with changing levels of network traffic. This would require appropriate modifications of our task calculations, but we believe that our overall framework would apply similarly.

- **Payment Model:** It is straight-forward to modify our model to include additional charges for the disposal of redundant service invocations (e.g., when dealing with the procurement of physical goods). However, adopting subscription schemes for allowing multiple service invocations at a fixed price would require some revisions of our calculations and optimisation algorithms. We believe that this is an interesting future extension for our work.

- **Reward Model:** As we use generic local search algorithms in our work, we believe that it is possible to consider more complex utility functions, including those that depend on multiple criteria (such as the timeliness and the quality of the overall workflow output). This would make our work more applicable for scenarios such as video rendering, streaming or compression tasks, where the output quality might have a significant impact on the user's satisfaction.

- **Model Scope:** In future work, we will cover more extensive workflow models that may occur in practice, and which will require small modifications to the way we aggregate performance parameters over the workflow. We envisage that a large number of other domain-specific requirements can be easily incorporated into our approach by placing appropriate constraints on the hill-climbing algorithm. For example, when it is impossible to provision multiple services in parallel for a particular task, the corresponding parameter n can be held constant at 1. Similarly, as mentioned in Section 3.1, when there are close dependencies between several services offered by a single provider, these can be aggregated and viewed at a higher level of abstraction as a single unit (e.g., a book vendor's *submitOrder* and *payOrder* services might be aggregated, as they only produce the desired result of ordering a book when used in conjunction).

7.4.2 Future Extensions

To conclude our discussion of future work, we now turn our attention towards other improvements and extensions to our work that we believe are interesting to pursue in the future.

One immediate area of further research is the development of improved heuristics and estimation techniques for aggregating the global performance parameters of workflows. In particular, our approach currently estimates the duration of a workflow using a normal distribution along the critical path of the workflow. However, such an approach generally results in an optimistic

estimate, as it uses the mean task durations to find this critical path, without considering the possibility that other tasks outside this path may in fact take longer at run-time.

To improve this, it may be possible to find an analytical solution to the overall probability distribution when certain assumptions about the workflow structure are made (e.g., that the graph is a tree or that it is reducible). When these assumptions do not hold, there are a number of existing techniques that can improve the accuracy of the critical path technique. For example, these include techniques that also take into consideration the variance of tasks to calculate the most critical path (Soroush (1994)), that identify a number of candidate critical paths (Dodin (1984)) or that use simulation to obtain distribution estimates (Cook and Jennings (1979)). Any of these techniques would require few modifications to our proposed model.

Next, it will be interesting to adapt our approach to a range of negotiation mechanisms. Currently, we use the contract net protocol in Chapter 6, which is a common and simple mechanism for multi-agent systems. However, there are many others that have been proposed in the literature and which we summarised briefly in Chapter 2. We believe that our current model can be adapted for these strategies with only few modifications — for example, the high-level strategies we use in our work can be adapted to refer to the use of different negotiation protocols and possibly for bidding strategies on these protocols (e.g., how fast to concede in bilateral negotiation or what service requests to post in a reverse auction).

Moreover, our work can be extended to consider systems that display a higher level of dynamism than considered thus far. As described above, in Section 7.2.6.3, we currently consider that the availability of offers and their performance characteristics vary according to probability distributions and a stochastic birth-death process. However, we do not currently assume that the underlying parameters of these distributions change over time. Clearly, this shortcoming should be addressed in future work, to enable us to model systems where significant changes may take place (e.g., where demand for particular services suddenly rises dramatically, or where new providers with significantly higher reliability enter the system).

Generally, such dynamism will most likely be addressed by work on trust and reputation, which has already considered how to track changes in the performance of agents (see Chapter 2). In this case, the updated values could simply be used in our existing algorithms and provisioning could be adapted at run-time in a similar manner as described in Chapter 6. However, even when dynamic trust and reputation information is available, our work on high-level strategies may need to be revised, as it depends on derived performance information that the consumer has accumulated itself. This might be addressed by constantly observing the market during execution and updating the strategy library accordingly, possibly by considering only offers over a limited time-frame.

Finally, the work in this thesis has been concerned with proposing a generic decision-making procedure for flexible service provisioning in distributed systems. As such, we have concentrated on abstract, high-level concepts when referring to services and workflows, without grounding our techniques in specific technologies and applications. While this has allowed us

to take a general approach, more work will be required to apply our work directly to a particular application and we intend to consider this in future work.

In more detail, we believe that our techniques will fit naturally on top of existing workflow execution engines (discussed in Section 2.4.1). As an additional decision-making layer in these applications, our algorithm can automate the provisioning of services, given an abstract workflow and a suitable service index (which could range from simple manually specified lists of services to sophisticated semantic matchmaking mechanisms based on OWL-S or SAWSDL). Furthermore, we believe we can build on and extend work that has already proposed dynamic provisioning techniques for established technologies, such as Web services and WS-BPEL (see Chapter 2). This might include work by Friese et al. (2005) on self-healing WS-BPEL workflows or work by Mandell and McIlraith (2003) on using semantically annotated services to provision abstract WS-BPEL workflows.

This concludes the summary of our research contributions and future work. To give further background information, the following appendices provide some supplementary material that extends the main work presented in this thesis. Specifically, Appendix A shows that our work is robust to inaccurate service information, Appendix B investigates the scalability of our approach, Appendix C provides results regarding the hardness of the provisioning problem, Appendix D discusses in more detail some of the equations presented in our work and Appendix E lists the acronyms used throughout the thesis.

Appendix A

Sensitivity Analysis

Throughout this thesis, we have assumed the service consumer to have access to accurate performance information about the services offered by provider agents. However, obtaining such information is clearly a challenging task, as we saw in Chapter 2, and often the consumer will have to rely on estimated and slightly inaccurate performance information. This is particularly the case in open and dynamic systems, where new providers may enter the system at any time and where little prior information may be available about their behaviour.

Although the design of appropriate trust and reputation mechanisms has not been the focus of this thesis, we briefly show empirically in this appendix that our flexible approach is robust to moderate inaccuracies in the performance information of services. This is not a surprising result as our work already relies on heuristic methods to estimate some of the global workflow parameters. For the sake of this discussion, we focus on the *flexible* strategy presented in Chapter 4, as we believe this strategy to be the most vulnerable to inaccurate information. In particular, this strategy assumes all providers to be homogeneous and so small inaccuracies may result in significantly biased overall estimates.

In order to evaluate the performance of this strategy in the presence of inaccurate information, we follow the same experimental setup as in Section 4.5.1, but now systematically introduce errors into the information that is available to a service consumer following the *flexible* strategy. To this end, we first evaluate the effect of relying on inaccurate failure probabilities, and then examine the impact of inaccurate service duration information. In both cases, we expect the performance of our strategy to decrease as the information becomes less accurate.

In our first set of experiments, we consider the case where the consumer *underestimates* the failure probability of service providers. Hence, we multiply the actual values for the failure probabilities $f(s_i)$ by a scalar $\epsilon_f < 1$ to provide an inaccurate input to the *flexible* strategy. The results for various values of ϵ_f are shown in Figure A.1. In most cases, the average net profit gained by the strategy degrades gracefully as the performance information becomes more inaccurate. In fact, when the (true) failure probability is low in the environment (up to around 0.3),

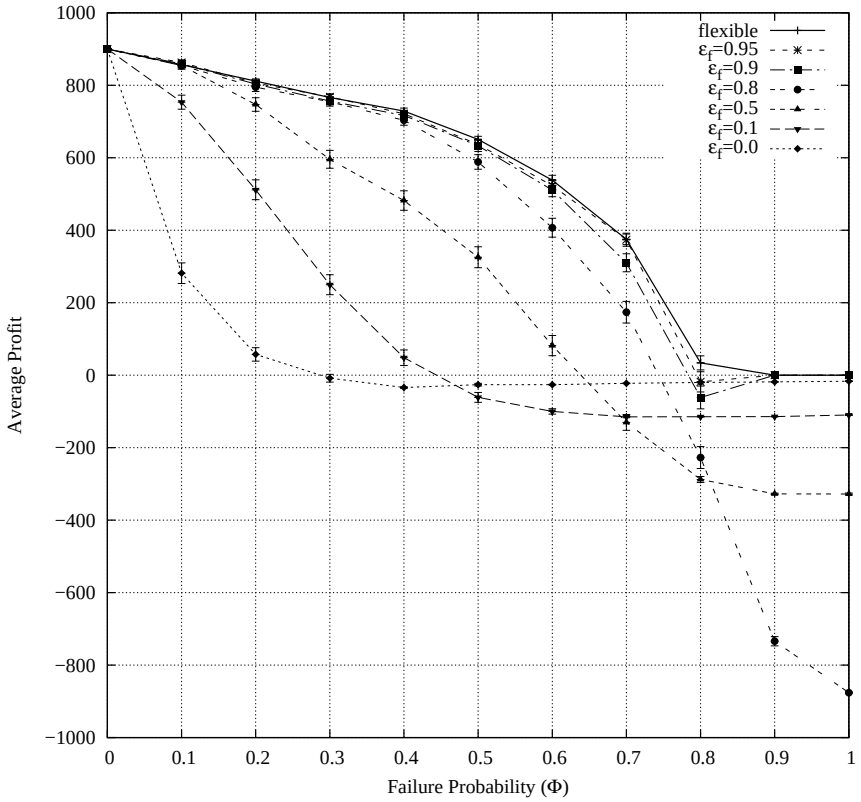


FIGURE A.1: Effect of underestimating the failure probability of providers ($\epsilon_f < 1$).

the strategy does well even if the information is up to 90% inaccurate (i.e., $\epsilon_f = 0.1$). However, when the failure probability rises to 0.7 and beyond, the impact of inaccurate information becomes more detrimental to the performance of the strategy. This is particularly evident when $\epsilon_f = 0.8$, which results in a large net loss at high failure probabilities. This is because the strategy provisions a large number of providers in parallel without detecting that the workflow is infeasible (and thus, it loses its high investment). Perhaps surprisingly, when information becomes even more inaccurate at high failure probabilities, the consumer begins to make smaller losses again. This is due to the strategy provisioning less providers in parallel and therefore losing less of its investment when the workflow eventually fails. Despite the special case when $\epsilon_f = 0.8$, the results are promising and show that small inaccuracies in the information (up to 10%) have little or no effect on our strategy. In most other cases, performance simply degrades gracefully as the information becomes more inaccurate.

Next, we are interested in the trends resulting from *overestimating* the failure probability of service providers. Hence, we now multiply the failure probabilities by a scalar $\epsilon_f > 1$ to provide an inaccurate input to our strategy (using a failure probability of 1 whenever $f(s_i) \cdot \epsilon_f > 1$). The results of this are shown in Figure A.2. Not surprisingly, the performance of the strategy simply degrades as the perceived failure probability rises. Because its behaviour is more conservative when it overestimates the failure probability of providers (it will provision unnecessarily many providers), it never makes a long-term loss. These results show that our strategy performs well, even when it significantly overestimates failure probabilities. In fact, the overall performance

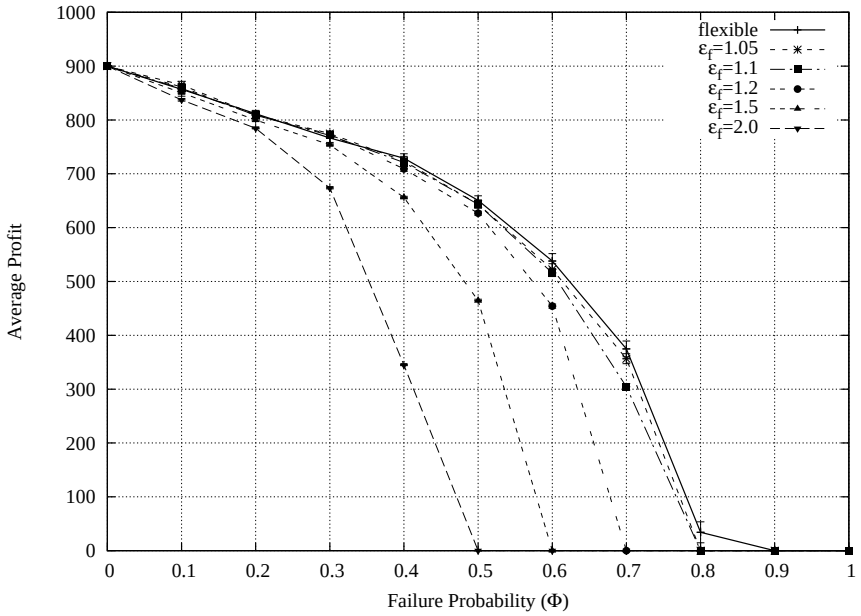


FIGURE A.2: Effect of overestimating the failure probability of providers ($\epsilon_f > 1$).

degrades only slightly when the failure probability is overestimated by 10% ($\epsilon_f = 1.1$). Even at 20% ($\epsilon_f = 1.2$), the performance is extremely good, and at 50% the strategy still performs reasonably well compared to the case with accurate information.

Apart from the failure probabilities, the *flexible* strategy also relies on probability density functions for the duration of a service execution. Because these will most likely be based on past observations and can be subject to noise, we now examine the effect of inaccurate information about these functions. Here, we multiply the scale parameter θ of the underlying gamma distribution by a scalar ϵ_d to yield an inaccurate duration distribution. By varying the scale parameter, we ensure that the mean of the distribution is varied proportionally with ϵ_d (e.g., when $\epsilon_d = 0.5$, the consumer estimates the mean service execution time to be half of the true value), while the overall shape of the distribution stays the same.

As before, we first consider the case of *underestimating* the duration of service providers ($\epsilon_d < 1$). The results are shown in Figure A.3. Here, the strategy handles an error of up to 20% ($\epsilon_d = 0.8$) very well with only a marginal performance decrease. Even when the error rises to 30% ($\epsilon_d = 0.7$), the performance comes close to the case with accurate information. However, as the information becomes even more inaccurate, the strategy performs increasingly badly. Also, it is evident that the strategy behaves more erratically at the same time — occasionally, the average net profit at a given level of inaccuracy increases as the failure probability rises (this is because the strategy constantly varies the balance between parallel and serial invocations, the latter of which is more susceptible to wrong duration estimates).

Finally, Figure A.4 shows the corresponding results when the consumer *overestimates* the service duration. Here, the performance again degrades slowly as the error rises. This is because the agent allocates unnecessarily long waiting times to the providers or provisions parallel providers

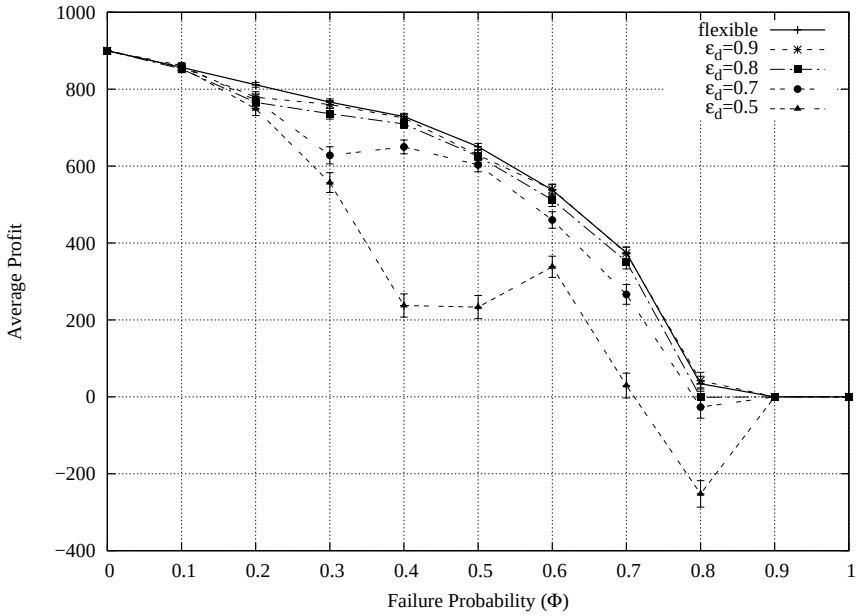


FIGURE A.3: Effect of underestimating the service duration of providers ($\epsilon_d < 1$).

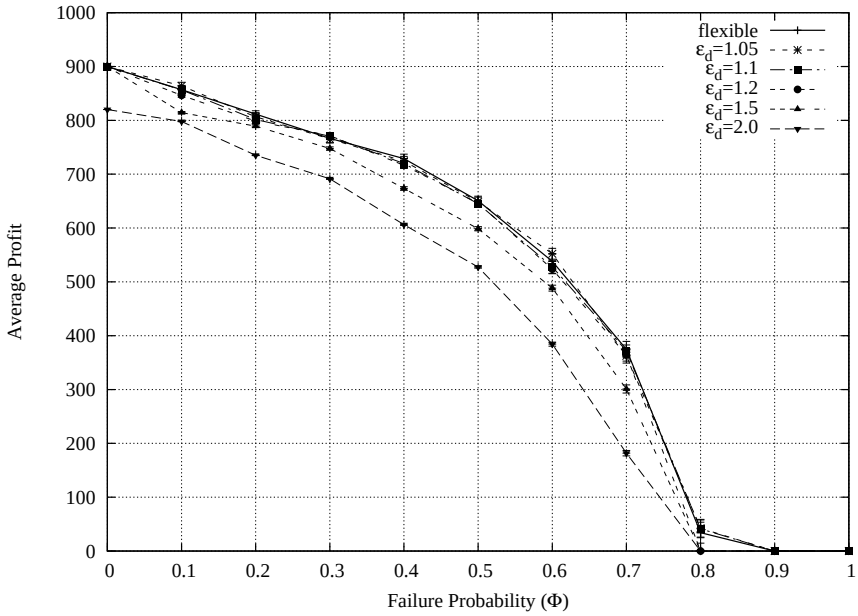


FIGURE A.4: Effect of overestimating the service duration of providers ($\epsilon_d > 1$).

when this is not needed. However, the loss in performance is clearly very small. This is because the consumer will occasionally wait longer than required or incur extra expenditures by provisioning parallel providers, but in many cases, the providers will simply complete their services earlier than anticipated and the consumer will be able to continue the workflow immediately and without penalty.

To conclude the sensitivity analysis, the results presented in this section show that our strategy is robust to small and moderate inaccuracies. In all cases, it performs well when the information provided is within 10% of the true value, and often errors up to 20% and 30% lead to only

marginal decreases in performance, especially when the consumer is overly pessimistic (i.e., when it overestimates the failure probability or duration of services). Overall, performance generally degrades gracefully as larger errors are introduced into the information that is known about providers (until they are too large to be of any value to the consumer — e.g., as ϵ_d reaches 0.5).

We also identified one case where underestimating the failure probability of providers can lead to poor performance. However, this only occurs in very specific scenarios when providers are highly unreliable and when the error in information is a significant 20%. Hence, our strategy may benefit from identifying these conditions in advance (e.g., by observing that the expected utility of a provisioned workflow is very low compared to the expected cost). Nevertheless, the overall results presented here are promising, showing that our strategy is applicable even in environments where completely accurate performance information is unavailable (as will be typical in any large dynamic multi-agent system).

Appendix B

Scalability of Flexible Provisioning

In order to address our Requirement A.3 for scalable techniques, we have concentrated in this thesis on designing heuristics that are suitable for complex environments with large workflows and many service providers. In particular, our proposed algorithms use utility estimates that can be computed in polynomial time, and we employ local search techniques with anytime properties, i.e., that can be interrupted after any amount of time to yield a candidate provisioning solution (the quality of which depends on the time of interruption). Hence, our techniques can be applied in scenarios where provisioning allocations for complex problems must be calculated within a reasonable amount of time.

Now, to convey a better understanding of the scalability of our techniques, we investigate in more detail the time it takes them to find a good solution when confronted with complex workflows. As all our strategies proposed in Chapters 4–6 are based on a similar technique for estimating the overall workflow utility $\tilde{u}$, we concentrate here on the *fast flexible* strategy outlined in Chapter 5, and examine how well it copes with workflows of varying sizes[1]. As discussed in Section 4.4.3.2, we have already seen that the time complexity of our estimation technique, with respect to the workflow size, is in $O(|T|^2)$ when run initially and $O(|T| + |\mathcal{E}|)$ for each subsequent iteration. Furthermore, we carry out up to $10 \cdot |T|$ iterations of the the main local search routine, each of which may examine every single task in T (see Algorithm 5.10). Hence, the complexity of the *fast flexible* strategy is polynomial (in $O(|T|^2 \cdot (|T| + |\mathcal{E}|))$). Furthermore, we expect it to perform better in practice, as it will usually complete each iteration after only considering a small number of tasks.

To measure the provisioning time of the *fast flexible* strategy in practice, we adopt the same experimental parameters as in the second half of Section 5.4.9 that considers a highly heterogeneous environment (in particular those shown in Table 5.7). We then consider $\Phi = 0.5$ and vary the number of workflow tasks n_T. Furthermore, we scale both the maximum workflow utility $u_{\max}$ and the deadline $t_{\max}$ by a factor $\frac{n_T}{50}$. This is done to adjust the problem to the workflow

[1]This particular strategy is chosen here simply as a representative strategy. Due to the similar estimation techniques, all strategies display the same general trends.

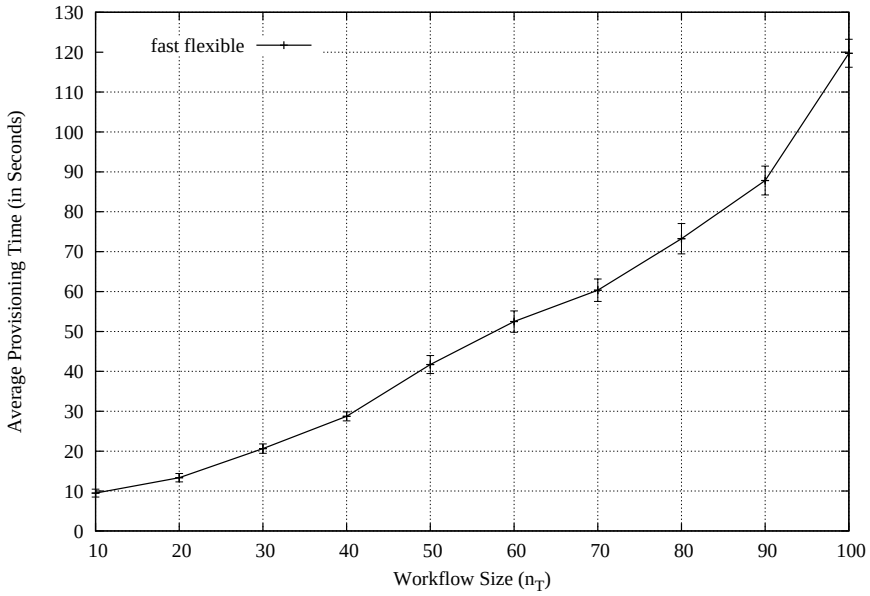


FIGURE B.1: Average time required by the *fast flexible* strategy to find a provisioning allocation.

FIGURE B.2: An example workflow with $n_T = 100$.

size, as larger workflows will incur higher costs and take longer to complete. All experiments reported in this appendix were conducted on a PC with an Intel Core 2 Duo 2.2Ghz CPU, 4 GB RAM and running Windows Vista. To obtain 95% confidence intervals for all results, the experiments were repeated 30 times for each workflow size.

Figure B.1 shows the time required by the *fast flexible* strategy to provision workflows as we increase the workflow size n_T from $n_T = 10$ to $n_T = 100$. Here, the strategy initially takes about 9.48 ± 0.98 seconds to complete a workflow with 10 tasks. This time then rises gradually as the workflow size is increased — by $n_T = 100$, it has risen to about 2 minutes (119.72 ± 3.50 seconds). We believe that this is reasonable, considering that such workflows are highly complex

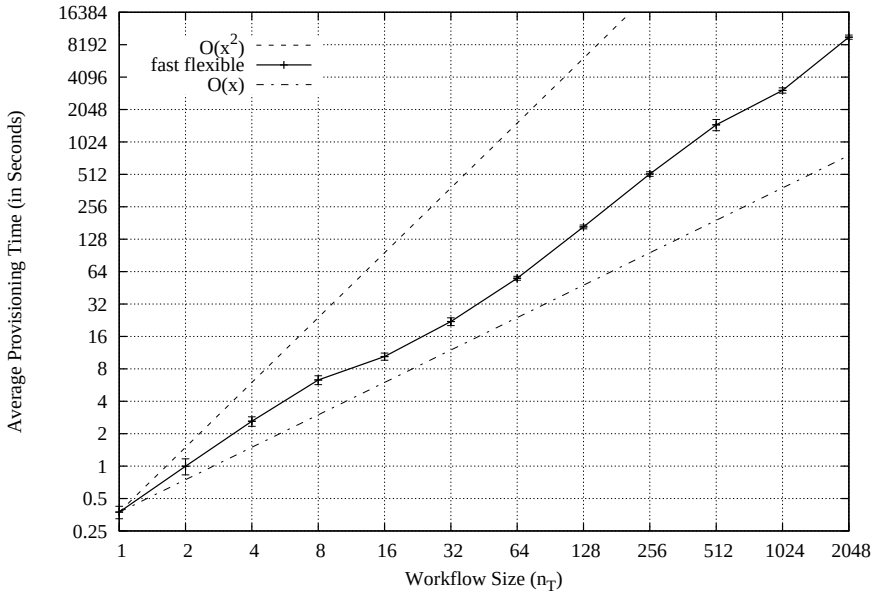


FIGURE B.3: Average time required by the *fast flexible* strategy for larger workflows.

and that there are typically hundreds of service providers for each workflow task (in fact, up to 2000, given the distributions in Table 5.7). To illustrate this complexity, Figure B.2 shows an example workflow with $n_T = 100$, as used in these experiments.

Next, Figure B.3 plots the trends of the *fast flexible* strategy over a different range of workflow sizes. Here, we consider the provisioning times of $n_T = 2^k$ tasks, where $k = 0, 1, \ldots, 10$. These experiments demonstrate how the algorithm copes with larger workflows as n_T is successively doubled in size (for this reason, both axes are shown in logarithmic scale with base 2). The overall trend in the graph is promising, highlighting a running time that grows only slightly more quickly than n_T (for reference, the graph also displays a function that grows linearly with n_T and one that grows quadratically). Although the algorithm begins to take a considerable time to find a solution as n_T becomes larger (requiring 1478.56 ± 175.72 seconds when $n_T = 512$, 3081.37 ± 173.96 seconds when $n_T = 1024$ and 9621.48 ± 433.22 seconds when $n_T = 2048$), the problem still remains tractable when considering such complex environments (again, for illustration of this, Figures B.4 and B.5 show workflows with $n_T = 512$ and $n_T = 1024$, respectively).

Furthermore, we believe that there is ample scope for refining and speeding up the *fast flexible* strategy in practice, as we have not so far concentrated on optimising the implementations of our algorithms. Such optimisation could be achieved, for example, by considering faster, approximate methods of calculating local task characteristics or by using faster programming languages (we have used Java for all our simulations). Not least, significant parts of the algorithm can be distributed to several parallel processors, including the utility calculations of neighbour allocations and the restarts of the local search.

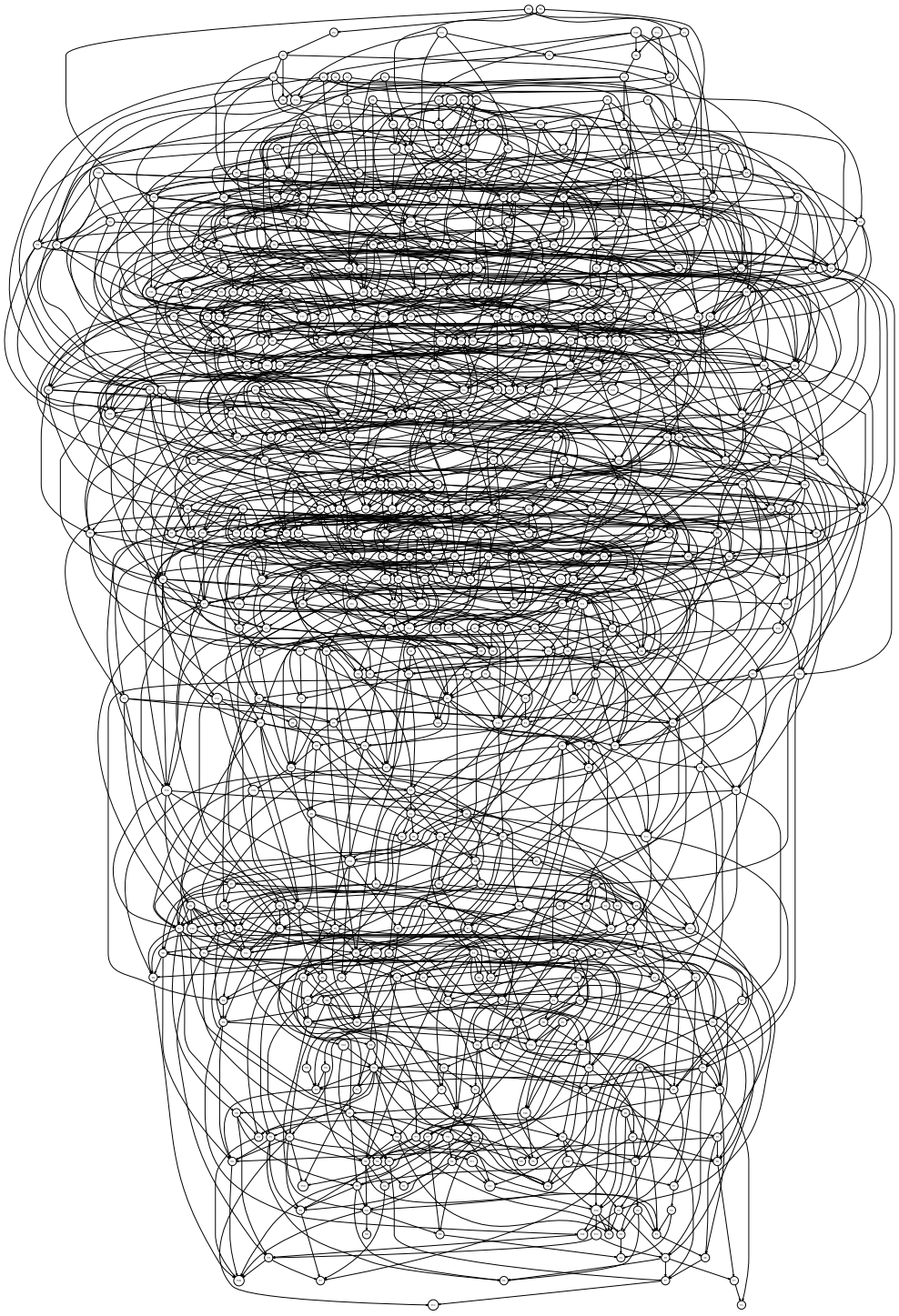


FIGURE B.4: An example workflow with $n_T = 512$.

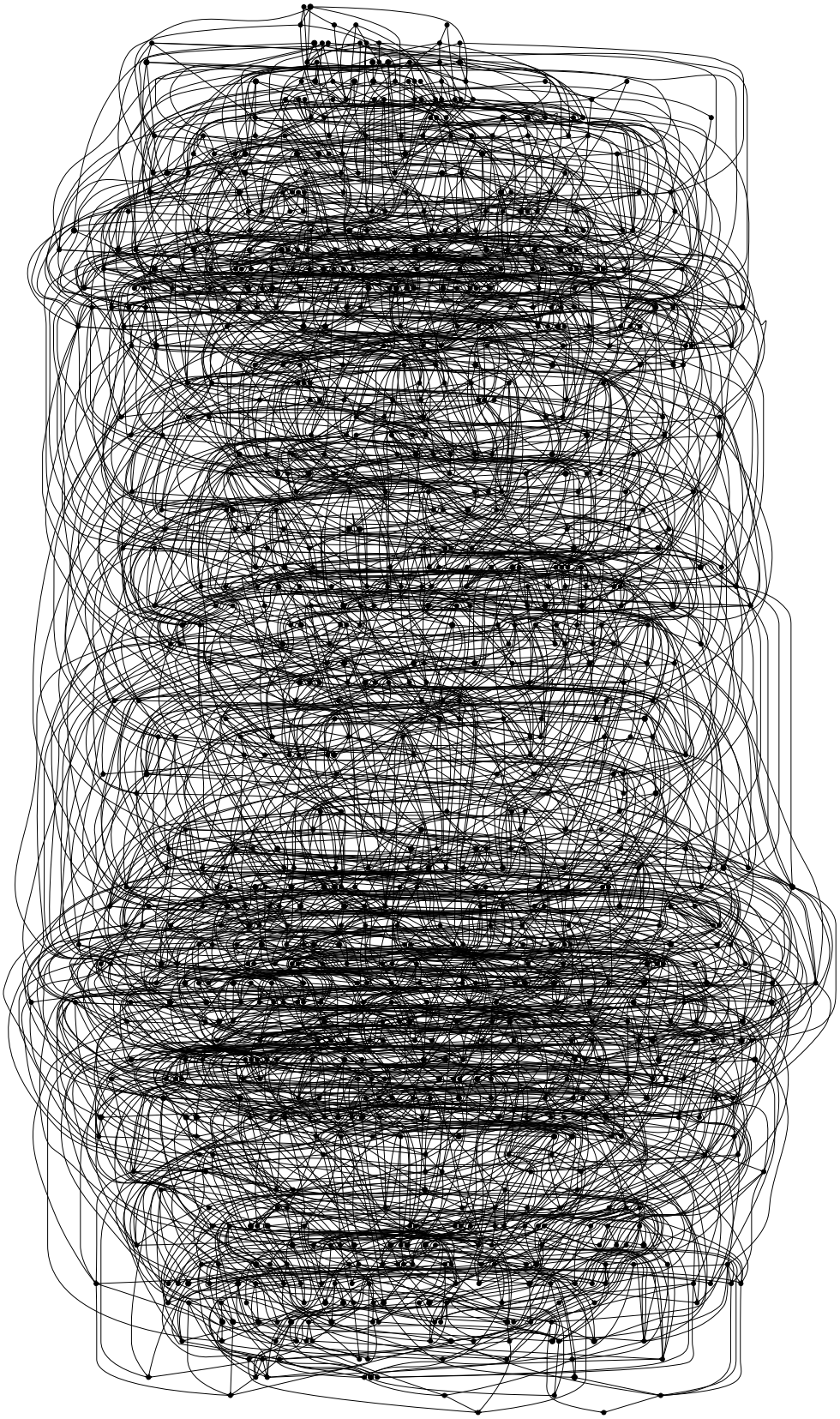


FIGURE B.5: An example workflow with $n_T = 1024$.

Appendix C

NP-Hardness of Provisioning Problem

In this thesis, we have concentrated on providing fast heuristics for a problem that is inherently difficult to solve optimally. To justify this, we have so far referred to results on the complexity of computing duration distributions of workflows, which is known to be a #P-complete problem and therefore also NP-hard (Hagstrom (1988)). In this appendix, we show more formally that the provisioning problem considered in our work is also NP-hard. To do this, we demonstrate how instances of two well-known NP-hard problems can be reduced, in polynomial time, to instances of the provisioning problem. We decided to show two such reductions, because they highlight two different sources of complexity inherent in our problem — first, the uncertainty of service durations we consider in Chapters 4 and 5 and, secondly, the combinatorial problem of dealing with highly heterogeneous service providers, as considered in Chapters 5 and 6.

Throughout this appendix, we consider the following, formal definition of the service provisioning problem:

Definition 17. (PROVISIONING): Given a workflow W, a set of service instances $\mathcal{S}$, matching function μ and quality functions f, D and c, find a (possibly empty) detailed provisioning allocation α^* that maximises the expected net profit of a consumer following it[1].

Now, we want to show the following:

Theorem C.1. PROVISIONING *is NP-hard.*

More specifically, we recall that a problem X is NP-hard if we can solve every problem in NP in polynomial time by solving instances of X in unit time (Garey and Johnson (1979)). Furthermore, as mentioned above, the following problem[2] is NP-hard (Hagstrom (1988)):

[1] All input parameters (W, $\mathcal{S}$, μ, f, D and c) are defined in more detail in Chapter 3. The detailed provisioning allocation α is defined in Section 5.3.1.

[2] For simplicity, the representation of this problem has been adapted for our problem. In its original form, edges represent tasks and nodes represent states that enable further tasks, but this can be quickly converted to our notation. Furthermore, the author considers task durations of length 0 and 1, but our model specifically excludes instantaneous services. However, their hardness result holds when considering non-zero durations.

Definition 18. (PERT CDF): Given a directed acyclic graph $G = (N, E)$ (whose nodes and edges represent tasks and dependencies, respectively), a rational probability p_i for each $n_i \in N$, such that node n_i will have a duration 1 with probability p_i and duration 2 with probability $1 - p_i$, and an input time t, compute the probability that the overall duration of the graph will not be more than t.

The above problem is NP-hard even when approximating the probability to a given error bound ϵ. Now, to prove Theorem C.1, we show that we could solve an instance of PERT CDF in polynomial time by solving instances of PROVISIONING in unit time.

Proof. First, construct a set of workflow tasks, T, and a set of edges, E', directly from G. Then construct μ and τ, so that each task t_i is mapped to exactly one service instance s_i, with $f(s_i) = 0$, $c(s_i) = 0$ and define $D(s_i, t)$, so that $D(s_i, t) = 0$ for $t < 1$, $D(s_i, t) = p_i$ for $1 \leq t < 2$ and $D(s_i, t) = 1$ for $t \geq 2$.

Next, create a utility function u with $t_{\max} = t + 1$, $u_{\max} = 1$ and $\delta = u_{\max}$. Furthermore, add an additional task t_{start} to T, which precedes all other tasks and is mapped to a single service instance s_{start} with $f(s_{\text{start}}) = 0$ and $D(s_{\text{start}}, t) = 0$ for $t < 1$ and $D(s_{\text{start}}, t) = 1$ otherwise. Now, setting $c(s_{\text{start}}) = p - \epsilon'$, where $0 < \epsilon' < \epsilon$, we perform a binary search for the largest possible value $p \in [0, 1]$ (dividing this interval in steps of ϵ), such that the corresponding PROVISIONING instance returns a non-empty provisioning allocation. As the expected reward of the workflow is equal to the probability that the duration of the original graph is t or less, and an empty provisioning allocation will be returned if $c(s_{\text{start}})$ is greater than this reward, the final value for p is now the required probability (within the error bound ϵ).

As this transformation and the binary search can be performed in polynomial time (as the number of values to consider for p is restricted by ϵ), we can thus solve instances of PERT CDF in polynomial time if we can solve PROVISIONING in unit time. This proves that PROVISIONING is NP-hard. □

As the proof uses only single providers for each workflow task, it applies equally to the problem discussed in Chapters 4 and 5. However, in Chapter 5, we also introduce the possibility of choosing between multiple heterogeneous service providers. This gives rise to another source of complexity, which applies similarly to the problem described in Chapter 6. Hence, we show in the following that the provisioning problem is still NP-hard, even when services always complete within a certain amount of time.

Theorem C.2. PROVISIONING *is NP-hard even when service durations are deterministic.*

Here, we recall a well-known NP-complete problem (Garey and Johnson (1979)):

Definition 19. (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 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 $\sum_{i \in I'} v(i) \geq V$.

To prove Theorem C.2, we show how an instance $\mathcal{K}$ of KNAPSACK can be reduced in polynomial time to an instance $\mathcal{P}$ of PROVISIONING.

Proof. Let $v_{\max}$ be the highest value of any item in I. Then, for every item $i \in I$, create a service instance 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 $D(s_i, t)$, so that the service duration is always exactly 1. Create workflow $W = (T, E, \tau, u)$ with $T = \{t_1, t_2, \ldots t_N\}$ and let E be any total order on T. Furthermore, define τ and μ, so that $\mu(\tau(t_i)) = \{s_i, s_0\}$. Also, create utility function u with deadline $t_{\max} = N + C$, maximum utility $u_{\max} = N(v_{\max} + 1) - V + \frac{1}{2}$ and penalty $\delta = u_{\max}$. This transformation is performed in $O(N)$.

Next, we show that the solution α^* to this new PROVISIONING instance $\mathcal{P}$ is sufficient to answer the original KNAPSACK instance $\mathcal{K}$. More specifically, we show that α^* is empty (no services are provisioned) if and only if the answer to $\mathcal{K}$ is "no". We prove this by contradiction in two steps:

1. Assume that α^* is empty and the answer to $\mathcal{K}$ is "yes". Then we can use the solution to $\mathcal{K}$ to find a provisioning allocation which is guaranteed to complete the workflow in time $t \leq N + C$, and whose cost is $c \leq N v_{\max} - V + N$. Since this would result in a net profit of at least $\frac{1}{2}$, α^* cannot be empty, and this is a contradiction.

2. Assume that α^* is non-empty and the answer to $\mathcal{K}$ is "no". Now, each task in α^* has exactly one provisioned service provider, as any other choice would be non-optimal[3]. Let T' be the set of tasks t_i for which service s_i has been provisioned[4]. The time for the workflow cannot be more than the deadline: $\sum_{t_i \in T'}(w(i) + 1) + \sum_{t_i \in T/T'} 1 \leq N + C$. This implies $\sum_{t_i \in T'} w(i) \leq C$ (the first constraint of $\mathcal{K}$). Furthermore, the total cost incurred must not be more than $u_{\max}$: $\sum_{t_i \in T'}(v_{\max} - v(i) + 1) + \sum_{t_i \in T/T'}(v_{\max} + 1) \leq N(v_{\max} + 1) - V - \frac{1}{2}$. This implies $\sum_{t_i \in T'} v(i) \geq V$. As both constraints of $\mathcal{K}$ are now shown to be satisfied, the answer to $\mathcal{K}$ cannot be "no", and this is a contradiction.

We conclude that there is a polynomial time decision procedure for KNAPSACK if instances of PROVISIONING can be solved in unit time. As KNAPSACK is NP-hard, so is PROVISIONING. □

Although the model used in Chapter 6 is different from the PROVISIONING problem described above, the proof of Theorem C.2 can be adapted for that chapter. In more detail, we can consider

[3] An optimal solution may contain unnecessary service providers that are never invoked. We ignore these here as they have no effect on the net profit or the following discussion.

[4] $T' = \{t_i \in T \mid \exists t \in \mathbb{Z}_0^+ \cdot \alpha^*(t_i) = \{(s_i, t)\}\}$.

a static market, where the returned offers for any call for proposal always correspond to the service instances outlined above, regardless of the time step that is requested. An agent following an optimal strategy will then start to buy offers from the market if and only if the KNAPSACK instance on which it is based is satisfiable.

In conclusion, the results in this appendix demonstrate that the provisioning problem is inherently hard and that there is no polynomial time algorithm to solve it optimally, unless P=NP.

Appendix D

Derivations of Equations

This appendix contains detailed derivations of some of the equations presented in Chapter 4. To this end, each of the following sections outlines and references relevant equations from that chapter.

D.1 Expected Task Cost (Equation 4.9)

Based on Figure 4.2, we first write the expected cost as a sum:

$$\begin{aligned}
\bar{c}_i &= \underbrace{n_i c_i + \hat{f}_i \cdot \Big(n_i c_i + \hat{f}_i \cdot \Big(n_i c_i + \hat{f}_i \cdot \Big(\ldots + \hat{f}_i \cdot (n_i c_i) \ldots \Big) \Big) \Big)}_{m \text{ instances of } n_i c_i} \\
&= n_i c_i + \hat{f}_i \cdot n_i c_i + \hat{f}_i^2 \cdot n_i c_i + \hat{f}_i^3 \cdot n_i c_i + \ldots + \hat{f}_i^{m-1} \cdot n_i c_i \\
&= n_i c_i \cdot \Big(1 + \hat{f}_i + \hat{f}_i^2 + \ldots + \hat{f}_i^{m-1} \Big) \\
&= n_i c_i \sum_{k=0}^{m-1} \hat{f}_i^k
\end{aligned} \tag{D.1}$$

Unfortunately, this sum grows with the number of available providers, v_i. To make it more tractable, we note that it is a geometric series and multiply Equation D.1 by $\hat{f}_i$:

$$\hat{f}_i \cdot \bar{c}_i = n_i c_i \cdot \Big(\hat{f}_i + \hat{f}_i^2 + \ldots + \hat{f}_i^{m-1} + \hat{f}_i^m \Big) \tag{D.2}$$

Then, we deduct Equation D.2 from D.1:

$$\begin{aligned}
\Big(1 - \hat{f}_i \Big) \cdot \bar{c}_i &= n_i c_i \Big(\Big(1 + \hat{f}_i + \hat{f}_i^2 + \ldots + \hat{f}_i^{m-1} \Big) \\
&\quad - \Big(\hat{f}_i + \hat{f}_i^2 + \ldots + \hat{f}_i^{m-1} + \hat{f}_i^m \Big) \Big) \\
&= n_i c_i \cdot \Big(1 - \hat{f}_i^m \Big)
\end{aligned} \tag{D.3}$$

Rewriting this, and assuming that $\hat{f}_i < 1$, we have:

$$\bar{c}_i = n_i c_i \cdot \frac{1 - \hat{f}_i^m}{1 - \hat{f}_i} \tag{D.4}$$

D.2 Expected Task Duration (Equation 4.14)

As before, we write the expected task duration as a weighted sum of all possible outcomes:

$$\begin{aligned}
\bar{t}_i &= \frac{1}{p_i} \cdot \sum_{k=1}^{m} \bar{d}_k \hat{f}_i^{k-1} \left(1 - \hat{f}_i\right) \\
&= \frac{1}{p_i} \cdot \sum_{k=1}^{m} ((k-1) \cdot w_i + \mu_i) \cdot \hat{f}_i^{k-1} \left(1 - \hat{f}_i\right) \\
&= \frac{1}{p_i} \cdot \sum_{k=0}^{m-1} (k \cdot w_i + \mu_i) \cdot \hat{f}_i^{k} \left(1 - \hat{f}_i\right)
\end{aligned} \tag{D.5}$$

Again, it is possible to rearrange this and rewrite it in closed form. In particular, we assume that $\hat{f}_i < 1$ and note that $\sum_{k=1}^{\infty} \hat{f}_i^k k = \hat{f}_i / (\hat{f}_i - 1)^2$.

$$\begin{aligned}
\bar{t}_i p_i &= \sum_{k=0}^{m-1} (k \cdot w_i + \mu_i) \cdot \hat{f}_i^{k} \left(1 - \hat{f}_i\right) \\
&= (1 - \hat{f}_i) \sum_{k=0}^{m-1} \hat{f}_i^k (\mu_i + k w_i) \\
&= (1 - \hat{f}_i) \left(\sum_{k=0}^{m-1} \hat{f}_i^k \mu_i + \sum_{k=1}^{m-1} \hat{f}_i^k k w_i \right) \\
&= (1 - \hat{f}_i) \left(\mu_i \left(\sum_{k=0}^{\infty} \hat{f}_i^k - \hat{f}_i^m \sum_{k=0}^{\infty} \hat{f}_i^k \right) \right. \\
&\quad \left. + w_i \left(\sum_{k=1}^{\infty} \hat{f}_i^k k - \hat{f}_i^{m-1} \sum_{k=1}^{\infty} \hat{f}_i^k k - (m-1) \hat{f}_i^m \sum_{k=0}^{\infty} \hat{f}_i^k \right) \right) \\
&= (1 - \hat{f}_i) \left(\mu_i \frac{1 - \hat{f}_i^m}{1 - \hat{f}_i} + w_i \left(\frac{\hat{f}_i - \hat{f}_i^m}{(1 - \hat{f}_i)^2} - \frac{(m-1) \hat{f}_i^m}{1 - \hat{f}_i} \right) \right) \\
&= \mu_i (1 - \hat{f}_i^m) + w_i \frac{\hat{f}_i - m \hat{f}_i^m + (m-1) \hat{f}_i^{m+1}}{1 - \hat{f}_i}
\end{aligned} \tag{D.6}$$

D.3 Expected Squared Waiting Time (Equation 4.20)

First, we express the expected squared waiting time by considering all possible outcomes:

$$\mathrm{E}(A_{Wi}^2) = \frac{(1-\hat{f}_i)w_i^2}{1-\hat{f}_i^m} \sum_{k=0}^{m-1} k^2 \hat{f}_i^k \tag{D.7}$$

In order to express this in closed form, we consider only the summation and re-use an intermediate result from Equation D.6 (as before assuming $\hat{f}_i < 1$):

$$\sum_{k=0}^{m-1} \hat{f}_i^k = \frac{1-\hat{f}_i^m}{1-\hat{f}_i} \tag{D.8}$$

Differentiating this with respect to $\hat{f}_i$ yields:

$$\sum_{k=0}^{m-1} k\hat{f}_i^{k-1} = \frac{1-\hat{f}_i^m}{(1-\hat{f}_i)^2} - \frac{m\hat{f}_i^{m-1}}{1-\hat{f}_i} \tag{D.9}$$

This can be multiplied by $\hat{f}_i$ to obtain:

$$\sum_{k=0}^{m-1} k\hat{f}_i^{k} = \frac{\hat{f}_i-\hat{f}_i^{m+1}}{(1-\hat{f}_i)^2} - \frac{m\hat{f}_i^{m}}{1-\hat{f}_i} \tag{D.10}$$

Differentiating and multiplying again finally yields the following:

$$\sum_{k=0}^{m-1} k^2 \hat{f}_i^k = \frac{1}{(1-\hat{f}_i)^3}(\hat{f}_i+\hat{f}_i^2-m^2\hat{f}_i^m-(2m+1-2m^2)\hat{f}_i^{m+1}+(2m-1-m^2)\hat{f}_i^{m+2}) \tag{D.11}$$

Combining this with Equation D.7, we obtain:

$$\begin{aligned} \mathrm{E}(A_{Wi}^2) \quad = \quad & \frac{w_i^2}{(1-\hat{f}_i^m)(1-\hat{f}_i)^2}(\hat{f}_i + \hat{f}_i^2 - m^2\hat{f}_i^m - \\ & (2m+1-2m^2)\hat{f}_i^{m+1} + (2m-1-m^2)\hat{f}_i^{m+2}) \end{aligned} \tag{D.12}$$

Appendix E

Acronyms

ADEPT Advanced Decision Environment for Process Tasks

HTTP Hypertext Transfer Protocol

MAGNET Multi Agent Negotiation Testbed

OGSA Open Grid Services Architecture

OWL Web Ontology Language

P2P Peer-to-Peer

PDDL Planning Domain Definition Language

QoS Quality-of-Service

SAWSDL Semantic Annotations for WSDL and XML Schema

SLA Service Level Agreement

SOAP Simple Object Access Protocol

SOC Service-Oriented Computing

UDDI Universal Description, Discovery and Integration

VO Virtual Organisation

WSAF Web Services Agent Framework

WS-BPEL Web Services Business Process Execution Language

WSDL Web Service Description Language

WSLA Web Service Level Agreement

WSMO Web Service Modeling Ontology

XML Extensible Markup Language

Bibliography

Abdul-Rahman, A. and Hailes, S. 1997. A distributed trust model. In *Proceedings of the 1997 Workshop on New Security Paradigms*, pages 48–60.

Aggarwal, R., Verma, K., Miller, J., and Milnor, W. 2004. Constraint driven web service composition in METEOR-S. In *Proceedings of the IEEE International Conference on Services Computing 2004 (SCC 2004), Shanghai, China*, pages 23–30. IEEE Computer Society Press.

Aghdaie, N. and Tamir, Y. 2003. Fast transparent failover for reliable web service. In *Proceedings of the 15th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS), Marina del Rey, USA*, pages 757–762. ACTA Press.

Agrawal, R., Bayardo, R. J., Gruhl, D., and Papadimitriou, S. 2001. Vinci: a service-oriented architecture for rapid development of web applications. In *Proceedings of the 10th International World Wide Web Conference*, pages 355–365. IEEE Computer Society Press.

Anderson, D. P. 2004. BOINC: a system for public-resource computing and storage. In *Proceedings of Fifth IEEE/ACM International Workshop on Grid Computing*, pages 4 – 10.

Anderson, D. P., Cobb, J., Korpela, E., Lebofsky, M., and Werthimer, D. 2002. SETI@home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61.

Andrieux, A., Czajkowski, K., Dan, A., Keahey, K., Ludwig, H., Nakata, T., Pruyne, J., Rofrano, J., Tuecke, S., and Xu, M. 2007. Web services agreement specification (ws-agreement). Technical Report GFD-R-P.107, Open Grid Forum. (available at `http://www.ogf.org/documents/GFD.107.pdf`).

Ardagna, D. and Pernici, B. 2007. Adaptive service composition in flexible processes. *IEEE Transactions on Software Engineering*, 33:369–384.

Arunachalam, R. and Sadeh, N. 2004. The 2003 supply chain management trading agent competition. In *Proceedings of the 6th International Conference on Electronic Commerce (ICEC '04), Delft, The Netherlands*, pages 113–120. ACM Press.

Avery, V., Chamberlaine, E., Summerfield, C., and Zealey, L., editors 2007. *Focus on the Digital Age*. The Office for National Statistics.

Avižienis, A. 1995. *Software Fault Tolerance*, chapter The Methodology of N-Version Programming, pages 23–46. Wiley.

Babanov, A., Collins, J., and Gini, M. 2004. Harnessing the search for rational bid schedules with stochastic search and domain-specific heuristics. In *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 269–276, Washington, DC, USA. IEEE Computer Society.

Baccelli, F., Jean-Marie, A., and Liu, Z. 1993. A survey on solution methods for task graph models. In Götz, N., Herzog, U., and Rettelbach, M., editors, *Arbeitsberichte der IMMD*, volume 26 (14), chapter Second QMIPS Workshop, pages 163–183. Universität Erlangen-Nürnberg, Erlangen.

Baker, M. A., Buyya, R., and Laforenza, D. 2002. The grid: International efforts in global computing. *International Journal of Software Practice and Experience*, 32(15).

Barborak, M., Dahbura, A., and Malek, M. 1993. The consensus problem in fault-tolerant computing. *ACM Computing Surveys*, 25(2):171–220.

Belecheanu, R. A., Munroe, S., Luck, M., Payne, T. R., Miller, T., McBurney, P., and Pechoucek, M. 2006. Commercial applications of agents: Lessons, experiences and challenges. In *Proceedings of the Fifth International Conference on Autonomous Agents and Multiagent Systems*.

Berners-Lee, T., Cailliau, R., Luotonen, A., Nielsen, H. F., and Secret, A. 1994. The world-wide web. *Communications of the ACM*, 37(8):76 – 82.

Berners-Lee, T., Hendler, J., and Lassila, O. 2001. The Semantic Web. *Scientific American*, 284(5):34–43.

Bernholdt, D., Bharathi, S., Brown, D., Chanchio, K., Chen, M., Chervenak, A., Cinquini, L., Drach, B., Foster, I., Fox, P., Garcia, J., Kesselman, C., Markel, R., Middleton, D., Nefedova, V., Pouchard, L., Shoshani, A., Sim, A., Strand, G., and Williams, D. 2005. The earth system grid: supporting the next generation of climate modeling research. *Proceedings of the IEEE*, 93(3):485–495.

Boddy, M. and Dean, T. L. 1994. Deliberation scheduling for problem solving in time-constrained environments. *Artificial Intelligence*, 67(2):245–285.

Botelho, S. and Alami, R. 1999. M+: a scheme for multi-robot cooperation through negotiated task allocation and achievement. In *Proceedings of the 1999 IEEE International Conference on Robotics and Automation*, volume 2, pages 1234–1239.

Brassard, G. and Bratley, P. 1996. *Fundamentals of Algorithmics*. Prentice-Hall.

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. 2004. Extensible Markup Language (XML) 1.0 (Third Edition). Technical report, W3C.

Brooks, R. 1986. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23.

Burstein, M., Bussler, C., Finin, T., Huhns, M., Paolucci, M., Sheth, A., Williams, S., and Zaremba, M. 2005. A semantic web services architecture. *Internet Computing, IEEE*, 9(5):72 – 81.

Butler, D. 1999. Computing 2010: from black holes to biology. *Nature*, 402:C67–C70.

Buyya, R., Abramson, D., and Giddy, J. 2001. A case for economy grid architecture for service oriented grid computing. In *10th Heterogeneous Computing Workshop HCW 2001*.

Buyya, R., Abramson, D., Giddy, J., and Stockinger, H. 2002. Economic models for resource management and scheduling in grid computing. *Concurrency and Computation: Practice and Experience*, 14(13–15):1507–1542.

Byde, A. 2006. A comparison between mechanisms for sequential compute resource auctions. In *Proceedings of the Fifth International Conference on Autonomous Agents and Multiagent Systems*.

Bylander, T. 1994. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2):165–204.

Canfora, G., Penta, M. D., Esposito, R., and Villani, M. L. 2005. QoS-aware replanning of composite web services. In *Proceedings of the IEEE International Conference on Web Services (ICWS'05), Orlando, USA*, pages 121–129. IEEE Computer Society.

Cardoso, J., Sheth, A., Millerb, J., Arnoldc, J., and Kochutb, K. 2004. Quality of service for workflows and web service processes. *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, 1(3):281–308.

Casati, F., Ceri, S., Paraboschi, S., and Pozzi, G. 1999. Specification and implementation of exceptions in workflow management systems. *ACM Transactions on Database Systems*, 24(3):405–451.

Casati, F., Dayal, U., and Shan, M.-C. 2001. E-business applications for supply chain management: challenges and solutions. In *Proceedings of the 17th International Conference on Data Engineering*, pages 71 – 78.

Chawathe, Y., Ratnasamy, S., Breslau, L., Lanham, N., and Shenker, S. 2003. Making gnutella-like P2P systems scalable. In *SIGCOMM '03: Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 407–418, New York, NY, USA. ACM Press.

Chern, M.-S. 1992. On the computational complexity of reliability redundancy allocation in a series system. *Operations Research Letters*, 11(5):309–315.

Christensen, E., Curbera, F., Meredith, G., and Weerawarana, S. 2001. Web Services Description Language (WSDL) 1.1. Technical report, W3C.

Cohen, P. R. 1995. *Empirical methods for artificial intelligence*. MIT Press, USA.

Coles, S., Frey, J. G., Hursthouse, M. B., Light, M. E., Meacham, K. E., Marvin, D. J., and Surridge, M. 2005. ECSES - examining crystal structures using 'e-science': a demonstrator employing web and grid services to enhance user participation in crystallographic experiments. *Journal of Applied Crystallography*, 38(5):819–826.

Collins, J., Bilot, C., Gini, M., and Mobasher, B. 2001. Decision Processes in Agent-Based Automated Contracting. *IEEE Internet Computing*, 5(2):61–72.

Collins, J., Ketter, W., Gini, M., and Mobasher, B. 2002. A multi-agent negotiation testbed for contracting tasks with temporal and precedence constraints. *International Journal of Electronic Commerce*, 7(1):35–57.

Collins, J., Tsvetovas, M., Sundareswara, R., van Tonder, J., Gini, M., and Mobasher, B. 1999. Evaluating risk: flexibility and feasibility in multi-agent contracting. In Etzioni, O., Müller, J. P., and Bradshaw, J. M., editors, *Proceedings of the Third International Conference on Autonomous Agents (Agents'99)*, pages 350–351, Seattle, WA, USA. ACM Press.

Cook, T. M. and Jennings, R. H. 1979. Estimating a project's completion time distribution using intelligent simulation methods. *The Journal of the Operational Research Society*, 30(12):1103–1108.

Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and Polk, W. 2008. Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile. RFC 5280, IETF.

Corson, M. S., Macker, J., and Cirincione, G. 1999. Internet-based mobile ad hoc networking. *IEEE Internet Computing*, 3(4):63–70.

Coyle, F. 2001. Breathing life into legacy. *IT Professional*, 3(5):17 – 24.

Cristian, F. 1991. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78.

Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N., and Weerawarana, S. 2002. Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6(2):86–93.

Curbera, F., Khalaf, R., Mukhi, N., Tai, S., and Weerawarana, S. 2003. The next step in Web services. *Communications of the ACM*, 46(10):29–34.

Czajkowski, K., Foster, I., and Kesselman, C. 2005. Agreement-based resource management. *Proceedings of the IEEE*, 93(3):631–643.

D'Ambrogio, A. 2006. A model-driven WSDL extension for describing the QoS of web services. *International Conference on Web Services 2006 (ICWS '06)*, pages 789–796.

Dan, A., Davis, D., Kearney, R., King, R., Keller, A., Kuebler, D., Ludwig, H., Polan, M., Spreitzer, M., and Youssef, A. 2004. Web services on demand: WSLA-driven automated management. *IBM Systems Journal*, 43(1):136–158.

Dash, R. K., Parkes, D., and Jennings, N. R. 2003. Computational mechanism design : A call to arms. *IEEE Intelligent Systems*, 18(6):40–47.

De Roure, D., Baker, M., Jennings, N., and Shadbolt, N. 2003. The Evolution of the Grid. In Berman, F., Fox, G., and Hey, A., editors, *Grid Computing: Making The Global Infrastructure a Reality*, pages 65–100. John Wiley & Sons.

Dean, T. and Boddy, M. 1988. An analysis of time-dependent planning problems. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 49–54.

Dearden, R., Friedman, N., and Andre, D. 1999. Model based bayesian exploration. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence (UAI'99)*, pages 150–159.

Deelman, E., Blythe, J., Gil, Y., Kesselman, C., Koranda, S., Lazzarini, A., Mehta, G., Papa, M. A., and Vahi, K. 2003a. Pegasus and the Pulsar Search: From Metadata to Execution on the Grid. In *Parallel Processing and Applied Mathematics: 5th International Conference, PPAM 2003, Czestochowa, Poland, September 7-10, 2003*, volume 3019 / 2004.

Deelman, E., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Blackburn, K., Lazzarini, A., Arbree, A., Cavanaugh, R., and Koranda, S. 2003b. Mapping Abstract Complex Workflows onto Grid Environments. *Journal of Grid Computing*, 1(1):25–39.

DeGroot, M. H. and Shervish, M. J. 2002. *Probability and Statistics*. Addison-Wesley, third edition edition.

Dodin, B. 1984. Determining the k most critical paths in pert networks. *Operations Research*, 32(4):859–877.

Dodin, B. 1985. Bounding the project completion time distribution in PERT networks. *Operations Research*, 33(4):862–881.

Edelkamp, S. and Hoffmann, J. 2003. PDDL2.2: The Language for the Classical Part of the 4th International Planning Competition. Technical Report 195, Institut für Informatik, Freiburg, Germany.

Eder, J. and Liebhart, W. 1995. The workflow activity model WAMO. In *Proceedings of the 3rd International Conference on Cooperative Information Systems, Vienna, Austria*, pages 87–98.

Erol, K., Nau, D. S., and Subrahmanian, V. S. 1995. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence*, 76(1–2):75–88.

Erradi, A., Maheshwari, P., and Tosic, V. 2006. Recovery policies for enhancing web services reliability. In *Proceedings of the IEEE International Conference on Web Services (ICWS'06), Chicago, USA*, pages 189–196. IEEE Computer Society.

Ewing, B., Hillier, L., Wendl, M. C., and Green, P. 1998. Base-calling of automated sequencer traces using phred. i. accuracy assessment. *Genome Research*, 8(3):175–185.

Faratin, P., Sierra, C., and Jennings, N. R. 1998. Negotiation decision functions for autonomous agents. *Int. Journal of Robotics and Autonomous Systems*, 24(3-4):159–182.

Foster, I. 2005. Globus toolkit version 4: Software for service-oriented systems. In Jin, H., Reed, D. A., and Jiang, W., editors, *IFIP International Conference on Network and Parallel Computing*, volume 3779 of *Lecture Notes in Computer Science*, pages 2–13. Springer.

Foster, I. and Iamnitchi, A. 2003. On death, taxes, and the convergence of peer-to-peer and grid computing. In *Peer-to-Peer Systems II: Second International Workshop, IPTPS 2003 Berkeley, CA, USA, February 21-22, 2003 Revised Papers*, volume 2735 of *Lecture Notes in Computer Science*, pages 118 – 128. Springer.

Foster, I., Jennings, N. R., and Kesselman, C. 2004. Brain meets brawn: Why Grid and agents need each other. In *Proceedings of the 3rd International Conference on Autonomous Agents and Multi-Agent Systems*, pages 8–15.

Foster, I. and Kesselman, C., editors 1999. *The grid: blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Foster, I., Kesselman, C., Nick, J., and Tuecke, S. 2002. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Technical report, Open Grid Service Infrastructure WG.

Foster, I., Kesselman, C., and Tuecke, S. 2001. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*, 15(3):200–222.

Frey, J., Tannenbaum, T., Livny, M., Foster, I., and Tuecke, S. 2001. Condor-G: A computation management agent for multi-institutional grids. In *HPDC '01: Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10'01)*, page 55, Washington, DC, USA. IEEE Computer Society.

Friese, T., Müller, J. P., and Freisleben, B. 2005. Self-healing execution of business processes based on a peer-to-peer service architecture. In *Proceedings of the 18th International Conference on Architecture of Computing Systems (ARCS '05), System Aspects in Organic and Pervasive Computing, Innsbruck, Austria*, volume 3432 of *Lecture Notes in Computer Science*, pages 108–123. Springer-Verlag.

Garcia-Molina, H. and Salem, K. 1987. Sagas. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data (SIGMOD '87), San Francisco, USA*, pages 249–259. ACM Press.

Garey, M. R. and Johnson, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA.

Geddes, N. 2006. The national grid service of the uk. In *Second IEEE International Conference on e-Science and Grid Computing (e-Science'06)*, pages 94–100.

Gentzsch, W. 2006. D-grid, an e-science framework for german scientists. In *The Fifth International Symposium on Parallel and Distributed Computing (ISPDC'06)*, pages 12–13.

Georgakopoulos, D., Hornick, M. F., and Sheth, A. P. 1995. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3(2):119–153.

Georgeff, M., Pell, B., Pollack, M., Tambe, M., and Wooldridge, M. 1999. The belief-desire-intention model of agency. In Müller, J., Singh, M. P., and Rao, A. S., editors, *Proceedings of the 5th International Workshop on Intelligent Agents V : Agent Theories, Architectures, and Languages (ATAL-98)*, volume 1555, pages 1–10. Springer-Verlag: Heidelberg, Germany.

Ghallab, M., Nau, D., and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Ghare, P. M. and Taylor, R. E. 1969. Optimal redundancy for reliability in series systems. *Operations Research*, 17(5):838–847.

Gibbins, N., Harris, S., and Shadbolt, N. 2003. Agent-based semantic web services. In *WWW '03: Proceedings of the Twelfth International Conference on World Wide Web*, pages 710–717, New York, NY, USA. ACM Press.

Golden, B., Bodin, L., Doyle, T., and Jr., W. S. 1980. Approximate traveling salesman algorithms. *Operations Research*, 28(3):694–711.

Golle, P., Leyton-Brown, K., Mironov, I., and Lillibridge, M. 2001. Incentives for sharing in peer-to-peer networks. In Fiege, L., Mühl, G., and Wilhelm, U., editors, *Electronic Commerce : Second International Workshop, WELCOM 2001 Heidelberg, Germany, November 16-17, 2001*, volume 2232 of *Lecture Notes in Computer Science*, page 75. Springer.

Gong, L. 2001. JXTA: a network programming environment. *IEEE Internet Computing*, 5(3):88–95.

Gopal, K., Aggarwal, K. K., and Gupta, J. S. 1978. An improved algorithm for reliability optimization. *IEEE Transactions on Reliability*, R-27(5):325–328.

Gruber, T. R. 1993. A translation approach to portable ontologies. *Knowledge Acquisition*, 5(2):199–220.

Gu, X. and Nahrstedt, K. 2002. A scalable QoS-aware service aggregation model for peer-to-peer computing grids. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, pages 73– 82.

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

He, M., Jennings, N., and Leung, H. 2003. On agent-mediated electronic commerce. *IEEE Transactions On Knowledge And Data Engineering*, 15(4):985–1003.

Hendler, J. 2001. Agents and the semantic web. *Intelligent Systems, IEEE*, 16(2):30 – 37.

Hollingsworth, D. 1995. The workflow reference model. Technical Report TC00-1003, Workflow Management Coalition.

Horvitz, E. J. 1988. Reasoning under varying and uncertain resource constraints. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI 1988)*, pages 111–116. Morgan Kaufmann.

Huang, G., Zhou, L., Liu, X.-Z., Mei, H., and Cheung, S.-C. 2006. Performance aware service pool in dependable service oriented architecture. *Journal of Computer Science and Technology*, 21(4):565–573.

Huhns, M. N., Holderfield, V. T., and Gutierrez, R. L. Z. 2003. Achieving software robustness via large-scale multiagent systems. In *Software Engineering for Large-Scale Multi-Agent Systems*, pages 171–210.

Huhns, M. N. and Singh, M. P. 2005. Service-Oriented Computing: Key Concepts and Principles. *IEEE Internet Computing*, 9(1):75–81.

Ingham, D. B., Panzieri, F., and Shrivastava, S. K. 1999. Constructing dependable web services. In *Advances in Distributed Systems, Advanced Distributed Computing: From Algorithms to Systems*, pages 277–294. Springer-Verlag.

Irwin, D. E., Grit, L. E., and Chase, J. S. 2004. Balancing risk and reward in a market-based task service. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC-13 '04), Honolulu, USA*, pages 160–169. IEEE Computer Society.

Ismail, R. and Jøsang, A. 2002. The beta reputation system. In *Proceedings of the 15th Bled Conference on Electronic Commerce*, Bled, Slovenia.

Jaeger, M., Rojec-Goldmann, G., and Mühl, G. 2004. QoS aggregation for web service composition using workflow patterns. In *Proceedings of the Eighth IEEE Enterprise Distributed Object Computing Conference (EDOC 2004)*, pages 149–159.

Jaeger, M. C. and Ladner, H. 2005. Improving the QoS of WS compositions based on redundant services. In *Proceedings of the 2005 International Conference on Next Generation Web Services Practices (NWeSP 2005), Seoul, Korea*, pages 189–194. IEEE Computer Society.

Jaeger, M. C. and Mühl, G. 2007. QoS-based selection of services: The implementation of a genetic algorithm. In *KiVS 2007 Workshop: Service-Oriented Architectures und Service-Oriented Computing (SOA/SOC), Bern, Switzerland*, pages 359–370. VDE Verlag.

Jennings, N. R. 2000. On Agent-Based Software Engineering. *Artificial Intelligence*, 117(2):277–296.

Jennings, N. R., Faratin, P., Johnson, M. J., Norman, T. J., O'Brien, P., and Wiegand, M. E. 1996. Agent-based business process management. *International Journal of Cooperative Information Systems*, 5(2–3):105–130.

Jennings, N. R., Faratin, P., Lomuscio, A. R., Parsons, S., Sierra, C., and Wooldridge, M. 2001. Automated negotiation: prospects, methods and challenges. *International Journal of Group Decision and Negotiation*, 10(2):199–215.

Johnson, M. E. and Whang, S. 2002. E-business and supply chain management: An overview and framework. *Production and Operations Management*, 11(4):413–423.

Jøsang, A., Ismail, R., and Boyd, C. 2007. A survey of trust and reputation systems for online service provision. *Decision Support Systems*, 43(2):618–644.

Kahn, R. 1972. Resource-sharing computer communications networks. *Proceedings of the IEEE*, 60(11):1397 – 1407.

Kang, Y., Herzog, J., and Spragins, J. 1988. FISHNET: a distributed architecture for high-performance local computer networks. *IEEE Transactions on Computers*, 37(1):119 – 123.

Keidl, M., Seltzsam, S., and Kemper, A. 2003. Reliable web service execution and deployment in dynamic environments. In *Proceedings of the International Workshop on Technologies for E-Services (TES)*, volume 2819 of *Lecture Notes in Computer Science*, pages 104–118.

Kifer, M., Lausen, G., and Wu, J. 1995. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42(4):741–843.

Kirkpatrick, S., C. D. Gelatt, J., and Vecchi, M. P. 1983. Optimization by simulated annealing. *Science*, 220(4598):671–680.

Klusch, M., Gerber, A., and Schmidt, M. 2005. Semantic web service composition planning with OWLS-XPlan. In *Proceedings of the 1st International AAAI Fall Symposium on Agents and the Semantic Web, Arlington, USA*, pages 55–62. AAAI Press.

Knight, J. 1972. A case study: Airlines reservations systems. *Proceedings of the IEEE*, 60(11):1423 – 1431.

Kochut, K., Arnold, J., Sheth, A., Miller, J., Kraemer, E., Arpinar, B., and Cardoso, J. 2003. IntelliGEN: A distributed workflow system for discovering protein-protein interactions. *Distributed and Parallel Databases*, 13(1):43–72.

Kopecký, J., Vitvar, T., Bournez, C., and Farrell, J. 2007. SAWSDL: Semantic annotations for WSDL and XML schema. *IEEE Internet Computing*, 11(6):60–67.

Kreger, H. 2003. Fulfilling the Web services promise. *Communications of the ACM*, 46(6):29–ff.

Krishna, V. 2002. *Auction Theory*. Academic Press.

Kuhn, N., Müller, H., and Müller, J. 1993. Task decomposition in dynamic agent societies. In *Proceedings of the International Symposium on Autonomous Decentralized Systems 1993 (ISADS 93)*, pages 165–171.

Kuo, W. 2000. An annotated overview of system-reliability optimization. *IEEE Transactions on Reliability*, 49(2):176–187.

Lamport, L., Shostak, R., and Pease, M. 1982. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401.

Laszewski, G. and Hategan, M. 2005. Workflow concepts of the java CoG kit. *Journal of Grid Computing*, 3(3–4):239–258.

Li, P., Hayward, K., Jennings, C., Owen, K., Oinn, T., Stevens, R., Pearce, S., and Wipat, A. 2004. Association of variations in i kappa b-epsilon with graves disease using classical and mygrid methodologies. In *Proceedings of UK e-Science Programme All Hands Meeting*, pages 832–839.

Li, W., He, J., Ma, Q., Yen, I.-L., Bastani, F., and Paul, R. 2005. A framework to support survivable web services. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers, Denver, USA*, page 93.2. IEEE Computer Society.

Liang, Y.-C. and Smith, A. 2004. An ant colony optimization algorithm for the redundancy allocation problem (RAP). *IEEE Transactions on Reliability*, 53(3):417–423.

Lindley, D. V. 1971. *Making Decisions*. John Wiley & Sons Ltd.

Luck, M., McBurney, P., Shehory, O., and Willmott, S. 2006. *Agent Technology: Computing as Interaction (A Roadmap for Agent-Based Computing)*. AgentLink.

Ludwig, H., Keller, A., Dan, A., King, R. P., and Franck, R. 2003. Web service level agreement (WSLA) language specification. Technical Report 1.0, IBM Corporation. (available at `http://www.research.ibm.com/wsla/WSLASpecV1-20030128.pdf`).

Lyu, M. and He, Y.-T. 1993. Improving the n-version programming process through the evolution of a design paradigm. *IEEE Transactions on Reliability*, 42(2):179–189.

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.

Mandell, D. and McIlraith, S. 2003. Adapting BPEL4WS for the Semantic Web: The Bottom-Up Approach to Web Service Interoperation. In *Proceedings of the Second International Semantic Web Conference, Sanibel Island, USA*, volume 2870 of *Lecture Notes in Computer Science*, pages 227–241. Springer-Verlag.

Martin, D., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., McIlraith, S., Narayanan, S., Paolucci, M., Parsia, B., Payne, T., Sirin, E., Srinivasan, N., and Sycara, K. 2004a. OWL-S: Semantic Markup for Web Services. Technical report, OWL-S Coalition.

Martin, D., Paolucci, M., McIlraith, S., Burstein, M., McDermott, D., McGuinness, D., Parsia, B., Payne, T., Sabou, M., Solanki, M., Srinivasan, N., and Sycara, K. 2004b. Bringing Semantics to Web Services: The OWL-S Approach. In *Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004), San Diego, USA*, volume 3387 of *Lecture Notes in Computer Science*, pages 26–42. Springer-Verlag.

Matei, R., Iamnitchi, A., and Foster, P. 2002. Mapping the Gnutella network. *IEEE Internet Computing*, 6(1):50 – 57.

Maturana, F. P. and Norrie, D. H. 1997. Distributed decision-making using the contract net within a mediator architecture. *Decision Support Systems*, 20(1):53–64.

Maximilien, E. and Singh, M. 2005. Multiagent System for Dynamic Web Services Selection. In *Proceedings of the AAMAS Workshop on Service-Oriented Computing and Agent-Based Engineering (SOCABE), Utrecht, The Netherlands*, pages 294–301.

Maximilien, E. M. and Singh, M. P. 2004. A framework and ontology for dynamic web services selection. *IEEE Internet Computing*, 8(5):84–93.

McDermott, D. 2002. Estimated-regression planning for interactions with Web Services. In *Proceedings of the 6th International Conference on AI Planning and Scheduling (AIPS'02), Toulouse, France*, pages 204–211. AAAI Press.

McGuinness, D. and van Harmelen, F. 2004. OWL Web Ontology Language Overview. Recommendation, W3C. (available at `http://www.w3.org/TR/2004/REC-owl-features-20040210/`).

McIlraith, S. A. and Son, T. C. 2002. Adapting Golog for Composition of Semantic Web Services. In *Proceedings of the Eighth International Conference on Knowledge Representation and Reasoning (KR2002), Toulouse, France*, pages 482–493. Morgan Kaufmann.

McIlraith, S. A., Son, T. C., and Zeng, H. 2001. Semantic Web Services. *IEEE Intelligent Systems*, 16(2):46–53.

Menasce, D. 2002. QoS issues in web services. *IEEE Internet Computing*, 6(6):72–75.

Merideth, M. G., Iyengar, A., Mikalsen, T., Tai, S., Rouvellou, I., and Narasimhan, P. 2005. Thema: Byzantine-fault-tolerant middleware for web-service applications. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05), Orlando, USA*, pages 131–142. IEEE Computer Society.

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

Milojicic, D. S., Kalogeraki, V., Lukose, R., Nagaraja, K., Pruyne, J., Richard, B., Rollins, S., and Xu, Z. 2002. Peer-to-peer computing. Technical Report HPL-2002-57R1, Hewlett-Packard.

Mitra, N. 2003. SOAP Version 1.2 Part 0: Primer. Technical report, W3C.

Mizukami, K. 1968. Optimum redundancy for maximum system reliability by the method of convex and integer programming. *Operations Research*, 16(2):392–406.

Morris, M. and Ogan, C. 1996. The internet as mass medium. *Journal of Communication*, 46(1):39–50.

Müller, J. P. 1996. *The Design of Intelligent Agents: A Layered Approach*, volume 1177 of *Lecture Notes in Computer Science*. Springer.

Muscettola, N., Nayak, P. P., Pell, B., and Williams, B. C. 1998. Remote agent: to boldly go where no ai system has gone before. *Artificial Intelligence*, 103(1-2):5–47.

Nadalin, A., Goodner, M., Gudgin, M., Barbir, A., and Granqvist, H. 2007. Ws-trust 1.3. Technical report, OASIS.

Nadalin, A., Kaler, C., Monzillo, R., and Hallam-Baker, P. 2006. Web services security: Soap message security 1.1 (ws-security 2004). Technical report, OASIS.

Naedele, M. 2003. Standards for xml and web services security. *Computer*, 36(4):96–98.

Narayanan, S. and McIlraith, S. A. 2002. Simulation, verification and automated composition of web services. In *Proceedings of the Eleventh International Conference on World Wide Web*, pages 77–88. ACM Press.

Neches, A.-L. 1993. FAST - a research project in electronic commerce. *Electronic Markets*, 3(3):25–27.

Nelson, B. 1981. Remote procedure call. Technical report, Xerox Corp.

Neuman, B. and Ts'o, T. 1994. Kerberos: an authentication service for computer networks. *IEEE Communications Magazine*, 32(9):33–38.

Ng, K.-C. and Abramson, B. 1990. Uncertainty management in expert systems. *IEEE Expert: Intelligent Systems and Their Applications*, 5(2):29–48.

Nicola, V. F. and Goyal, A. 1990. Modeling of correlated failures and community error recovery in multiversion software. *IEEE Transactions on Software Engineering*, 16(3):350–359.

Norman, T. J., Preece, A., Chalmers, S., Jennings, N. R., Luck, M., Dang, V. D., Nguyen, T. D., Deora, V., Shao, J., Gray, A. W., and Fiddian, N. J. 2004. Agent-based formation of virtual organisations. *Knowledge-Based Systems*, 17(2–4):103–111.

Noy, F. and Musen, A. 2002. Evaluating ontology-mapping tools: Requirements and experience. In *Proceedings of OntoWeb-SIG3 Workshop at the 13th International Conference on Knowledge Engineering and Knowledge Management*, pages 1–14.

O'Brien, A., Newhouse, S., and Darlington, J. 2004. Mapping of scientific workflow within the e-protein project to distributed resources. In *Proceedings of UK e-science All Hands Meeting (AHM 2004), Nottingham, UK*, pages 404–409. EPSRC.

Oinn, T., Addis, M., Ferris, J., Marvin, D., Senger, M., Greenwood, M., Carver, T., Glover, K., Pocock, M. R., Wipat, A., and Li, P. 2004. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054.

Oinn, T., Greenwood, M., Addis, M., Alpdemir, M. N., Ferris, J., Glover, K., Goble, C., Goderis, A., Hull, D., Marvin, D., Li, P., Lord, P., Pocock, M. R., Senger, M., Stevens, R., Wipat, A., and Wroe, C. 2006. Taverna: lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100.

Oram, A. 2001. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly & Associates, Inc., Sebastopol, CA, USA.

Paolucci, M., Kawamura, T., Payne, T. R., and Sycara, K. P. 2002. Semantic matching of web services capabilities. In *Proceedings of the First International Semantic Web Conference (ISWC 2002), Sardinia, Italy*, volume 2342 of *Lecture Notes in Computer Science*, pages 333–347. Springer-Verlag.

Papazoglou, M. 2003. Service-oriented computing: concepts, characteristics and directions. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering (WISE 2003)*, pages 3–12.

Parsons, S. and Wooldridge, M. 2002. Game theory and decision theory in multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 5(3):243–254.

Patel, C., Supekar, K., and Lee, Y. 2004. Provisioning resilient, adaptive web services-based workflow: A semantic modeling approach. In *IEEE International Conference on Web Services (ICWS'04)*, pages 480–487.

Paurobally, S. and Jennings, N. R. 2005. Protocol engineering for web services conversations. *Engineering Applications of Artificial Intelligence*, 18(2):237–254.

Pautasso, C. and Alonso, G. 2005. Flexible binding for reusable composition of web services. In *Proceedings of the 4th International Workshop on Service Composition (SC 2005)*, pages 151–166.

Pearl, J. 1984. *Heuristics: Intelligent search strategies*. Addison-Wesley.

Pordes, R., Petravick, D., Kramer, B., Olson, D., Livny, M., Roy, A., Avery, P., Blackburn, K., Wenaus, T., Würthwein, F., Foster, I., Gardner, R., Wilde, M., Blatecky, A., McGee, J., and

Quick, R. 2007. The open science grid. *Journal of Physics: Conference Series*, 78:012057 (15pp).

Raiffa, H. 1968. *Decision Analysis: Introductory Lectures on Choices Under Uncertainty*. McGraw-Hill, Inc., USA.

Ramchurn, S. D., Huynh, D., and Jennings, N. R. 2004. Trust in multiagent systems. *Knowledge Engineering Review*, 19(1):1–25.

Ran, S. 2003. A model for web services discovery with QoS. *SIGecom Exch.*, 4(1):1–10.

Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Schenker, S. 2001. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 161–172, New York, NY, USA. ACM.

Richards, W. 2002. Virtual screening using grid computing: The screensave project. *Nature Reviews Drug Discovery*, 1(7):551–555.

Roman, D., Lausen, H., Keller, U., de Bruijn, J., Bussler, C., Domingue, J., Fensel, D., Kifer, M., Kopecky, J., Lara, R. Oren, E., Polleres, A., and Stollberg, M. 2005. Web Service Modeling Ontology (WSMO). Technical Report D2v1.1., DERI.

Rosenschein, J. S. and Zlotkin, G. 1994. *Rules of Encounter: Designing Conventions for Automated Negotiation Among Computers*. The MIT Press.

Rubinstein, A. 1982. Perfect equilibrium in a bargaining model. *Econometrica*, 50(1):97–109.

Russell, S. and Norvig, P. 2003. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 2nd edition.

Sabater, J. and Sierra, C. 2002. Social ReGreT, a reputation model based on social relations. *ACM SIGecom Exchanges*, 3(1):44–56.

Sandholm, T. W. 1999. Distributed rational decision making. In Weiß, G., editor, *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, pages 201–258. MIT Press, Cambridge, MA, USA.

Sandholm, T. W. 2002. Algorithm for optimal winner determination in combinatorial auctions. *Artificial Intelligence*, 135(1-2):1–54.

Sandholm, T. W. and Lesser, V. R. 1995a. Advantages of Leveled Commitment Contracting Protocol. *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, 1:126–133.

Sandholm, T. W. and Lesser, V. R. 1995b. Issues in automated negotiation and electronic commerce: Extending the contract net framework. In Lesser, V. R., editor, *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS'95)*, pages 328–335, San Francisco, CA, USA. The MIT Press: Cambridge, MA, USA.

Sandholm, T. W. and Lesser, V. R. 1996. Advantages of a leveled commitment contracting protocol. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, Portland, OR.

Saroiu, S., Gummadi, K. P., and Gribble, S. D. 2003. Measuring and analyzing the characteristics of napster and gnutella hosts. *Multimedia Systems*, 9(2):170–184.

Schoder, D. and Fischbach, K. 2003. Peer-to-peer prospects. *Communications of the ACM*, 46(2):27–29.

Scott, R., Gault, J., and McAllister, D. 1987. Fault-tolerant software reliability modeling. *IEEE Transactions on Software Engineering*, SE-13(5):582–592.

Simon, H. A. 1957. *Models of man, social and rational : mathematical essays on rational human behavior in a social setting*. Wiley.

Simon, H. A. and Newell, A. 1958. Heuristic problem solving: The next advance in operations research. *Operations Research*, 6(1):1–10.

Singh, G., Kesselman, C., and Deelman, E. 2007. A provisioning model and its comparison with best-effort for performance-cost optimization in grids. In *Proceedings of the 16th International Symposium on High Performance Distributed Computing (HPDC '07)*, pages 117–126, New York, NY, USA. ACM Press.

Singh, M. P. and Huhns, M. N. 2005. *Service-Oriented Computing : Semantics, Processes, Agents*. John Wiley & Sons, Inc., USA.

Sirin, E., Hendler, J., and Parsia, B. 2003. Semi-automatic Composition of Web Services using Semantic Descriptions. In *Web Services: Modeling, Architecture and Infrastructure workshop in ICEIS 2003*, Angers, France.

Sirin, E., Parsia, B., and Hendler, J. 2005. Template-based composition of semantic web services. In *AAAI Fall Symposium on Agents and the Semantic Web, Arlington, USA*, pages 85–92. AAAI Press.

Smith, R. G. 1980. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions On Computers*, 29(12):1104–1113.

Smith, T. M., Abajian, C., and Hood, L. 1997. Hopper: software for automating data tracking and flow in DNA sequencing. *Comput. Appl. Biosci.*, 13(2):175–182.

Soroush, H. M. 1994. The most critical path in a pert network. *The Journal of the Operational Research Society*, 45(3):287–300.

Sreenath, R. M. and Singh, M. P. 2004. Agent-based service selection. *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, 1(3):261–279.

Srivastava, B. and Koehler, J. 2003. Web Service Composition - Current Solutions and Open Problems. In *Proceedings of ICAPS'03 Workshop on Planning for Web Services*, Trento, Italy.

Stäber, F. and Müller, J. P. 2007. Evaluating peer-to-peer for loosely coupled business collaboration: A case study. In *Business Process Management*, volume 4714 of *Lecture Notes in Computer Science*, pages 141–148.

Stein, S., Jennings, N. R., and Payne, T. R. 2006. Flexible provisioning of service workflows. In *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI-06), Riva del Garda, Italy*, pages 295–299. IOS Press.

Stein, S., Jennings, N. R., and Payne, T. R. 2007a. Provisioning heterogeneous and unreliable providers for service workflows. In *Proceedings of the 6th International Conference on Autonomous Agents and Multiagent Systems (AAMAS), Honolulu, Hawai'i, USA*, pages 523–525. ACM Press.

Stein, S., Jennings, N. R., and Payne, T. R. 2007b. Provisioning heterogeneous and unreliable providers for service workflows. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, Vancouver, Canada*, pages 1452–1458. AAAI Press.

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

Stein, S., Payne, T. R., and Jennings, N. R. 2008a. Flexible provisioning of web service workflows. 8(4). (in press).

Stein, S., Payne, T. R., and Jennings, N. R. 2008b. Flexible service provisioning with advance agreements. In *Proceedings of the 7th International Conference on Autonomous Agents and Multiagent Systems (AAMAS), Estoril, Portugal*, pages 249–256. ACM Press.

Stevens, R. D., Tipney, H. J., Wroe, C. J., Oinn, T. M., Senger, M., Lord, P. W., Goble, C. A., Brass, A., and Tassabehji, M. 2004. Exploring williams-beuren syndrome using mygrid. *Bioinformatics*, 20(Suppl. 1):303–310.

Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160.

Talia, D. 2002. The Open Grid Services Architecture: where the grid meets the Web. *IEEE Internet Computing*, 6(6):67– 71.

Teacy, W. T. L. 2006. *Agent-Based Trust and Reputation in the Context of Inaccurate Information Sources*. PhD thesis, School of Electronics and Computer Science, University of Southampton.

Teacy, W. T. L., Chalkiadakis, G., Rogers, A., and Jennings, N. R. 2008. Sequential decision making with untrustworthy service providers. In *Proceedings of the 7th International Conference on Autonomous Agents and Multiagent Systems*.

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.

Thain, D., Tannenbaum, T., and Livny, M. 2003. Condor and the grid. In Berman, F., Hey, A., and Fox, G., editors, *Grid Computing: Making The Global Infrastructure a Reality*, chapter 11, pages 299–335. John Wiley and Sons, Ltd.

Tillman, F. A., Hwang, C. L., and Kuo, W. 1977. Optimization techniques for system reliability with redundancy: A review. *IEEE Transactions on Reliability*, R-26:148–155.

Tillman, F. A. and Liittschwager, J. M. 1967. Integer programming formulation of constrained reliability problems. *Management Science*, 13(11):887–899.

Timmers, P. 1999. *Electronic commerce : strategies and models for business-to-business trading*. Wiley.

Townend, P., Groth, P., and Xu, J. 2005. A provenance-aware weighted fault tolerance scheme for service-based applications. In *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05), Seattle, USA*, pages 258–266. IEEE Computer Society.

Trivedi, K. 2001. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. John Wiley & Sons, Inc., USA, 2nd edition.

van der Aalst, W. M. P. 1998. The application of petri nets to workflow management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66.

van der Aalst, W. M. P., ter Hofstede, A., Kiepuszewski, B., and Barros, A. 2003. Workflow Patterns. *Distributed and Parallel Databases*, 14(3):5–51.

Verma, D., Sahu, S., Calo, S., Beigi, M., and Chang, I. 2002. A policy service for grid computing. In Parashar, M., editor, *Grid Computing - GRID 2002 : Third International Workshop*, pages 243 – 255. Springer.

Von Neumann, J. and Morgenstern, O. 1944. *Theory of games and economic behavior*. Princenton University Press.

Vulkan, N. and Jennings, N. R. 2000. Efficient mechanisms for the supply of services in multi-agent environments. *Decision Support Systems*, 28(1–2):5–19.

Wang, X., Vitvar, T., Kerrigan, M., and Toma, I. 2006. A QoS-aware selection model for semantic web services. In *Service-Oriented Computing 2006 (ICSOC 2006)*, volume 4294 of *Lecture Notes in Computer Science*, pages 390–401.

Wang, Y. and Vassileva, J. 2003. Bayesian network-based trust model. In *Proceedings of the IEEE/WIC International Conference on Web Intelligence (WI 2003)*, pages 372–378.

Weatherspoon, H., Moscovitz, T., and Kubiatowicz, J. 2002. Introspective failure analysis: Avoiding correlated failures in peer-to-peer systems. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems, Osaka, Japan*, pages 362–367. IEEE Computer Society.

Weerawarana, S., Curbera, F., Leymann, F., and Ferguson, T. S. D. F. 2005. *Web Services Platform Architecture*. Prentice-Hall.

Weiß, G., editor 1999. *Multiagent systems: A modern approach to distributed artificial intelligence*. MIT Press.

Wittie, L. 1991. Computer networks and distributed systems. *Computer*, 24(9):67 – 76.

Wooldridge, M. 2002. *An Introduction to MultiAgent Systems*. John Wiley & Sons, Chichester, England.

Wooldridge, M. and Ciancarini, P. 2000. Agent-oriented software engineering: The state of the art. In Ciancarini, P. and Wooldridge, M., editors, *First International Workshop on Agent-Oriented Software Engineering*, pages 1–28.

Wu, D., Sirin, E., Hendler, J., Nau, D., and Parsia, B. 2003. Automatic web services composition using SHOP2. In *Workshop on Planning for Web Services*, Trento, Italy.

Xiao, J. and Boutaba, R. 2005. QoS-aware service composition and adaptation in autonomic communication. *IEEE Journal on Selected Areas in Communications*, 23(12):2344– 2360.

Yang, Z. and Duddy, K. 1996. CORBA: a platform for distributed object computing. *ACM SIGOPS Operating Systems Review*, 30(2):4–31.

Yu, T. and Lin, K. 2005. Service selection algorithms for composing complex services with multiple QoS constraints. In *Proceedings of the Third International Conference on Service Oriented Computing (ICSOC2005), Amsterdam, The Netherlands*, volume 3826 of *Lecture Notes in Computer Science*. Springer.

Zeng, L., Benatallah, B., Ngu, A. H., Dumas, M., Kalagnanam, J., and Chang, H. 2004. QoS-aware middleware for web services composition. *IEEE Trans. Softw. Eng.*, 30(5):311–327.

Zeng, L., Lei, H., jang Jeng, J., Chung, J.-Y., and Benatallah, B. 2005. Policy-driven exception-management for composite web services. In *Proceedings of the Seventh IEEE International Conference on E-Commerce Technology (CEC'05)*, pages 355–363, Washington, DC, USA. IEEE Computer Society.

Zhao, B., L. Huang; Stribling, J., Rhea, S., Joseph, A., and Kubiatowicz, J. 2004. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41 – 53.

Zhou, C., Chia, L.-T., and Lee, B.-S. 2004. DAML-QoS ontology for web services. In *Proceedings of the IEEE International Conference on Web Services (ICWS 2004), San Diego, USA*, pages 472–479. IEEE Computer Society.

Zimmermann, O., Tomlinson, M. R., and Peuser, S. 2003. *Perspectives on Web Services: Applying SOAP, WSDL and UDDI to Real-World Projects*. Springer.